INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

H4.SMR. 403/22

FIFTH COLLEGE ON MICROPROCESSORS: TECHNOLOGY AND APPLICATIONS IN PHYSICS

2 - 27 October 1989

Software Tools

P. BARTHOLDI
Observatory of Geneva, Sauverny, Switzerland

Software Tools

Paul Bartholdi

Observatory of Geneva

CH-1290 Sauverny

Switzerland

Preliminary notes - Trieste - october 1989

Fifth College on Microprocessors

Contents

1 Introduction, Which Topics ?

We are going to look at various tools in a bottom up fashion, from *RESET Button* to *File System* and *Context Sensitive Editor*, from *Hardware* towards *Abstract Machine* that are adequate for applications.

- Programming
  - Machine Code
  - Assembler
  - Compiler - Interpreter
  - Linker - Editor
- System
  - RESET/ABORT Button - Switch Register
  - Monitor – Debugger
  - Functions of an Operating System
- Input/Output
  - Hardware devices (ACIA, disc controller)
  - Device Handler
  - File System

We will not necessarily follow strictly this order, nor, and by far, give equal weight to each point above. We will also make some lateral excursions.

2 Machine Code

It allows direct interaction with hardware, which can be very useful

- acceptable for very small programmes
- may be necessary for **bootstrapping**

2.1 How to prepare *Hand coding*

use tables of
opcodes,
addressing modes,
postbytes,
instruction sizes.
(all are very much CPU dependent)

build tables of
variables and constants,
branching labels,
while you write your code on blank sheets, leaving most addresses unfiled.

fill the tables
with the real addresses

put real addresses
back in the code (instructions)

This is very similar to what a real assembler does. Although it uses your brain, hands and pencil and not electronic circuits, it may well be faster in some cases than using an editor to prepare the instructions, loading and then running an assembler to produce machine code, that has to be eventually loaded into the computer and then executed.

3 Assembler

An Assembler will automatically execute the previous steps of hand coding.

Very often, it includes macro facilities that are not really part of the assembler.

3.1 Three Phases of Processing

The input string (file) will go through three phases :

1. Macro expansion. Macro definitions are stored in the local memory, and all macro calls are replaced with the expanded code of the macro and all the parameters replaced by their local value. This phase is independent of the processor, and in fact of the assembler. In many cases it is an other programme that is called automatically by the assembler before starting its execution.
2. Builds symbol table. The input string is scanned to find all symbols used in the programme, where they are defined, what kind of object they represent. This table may be saved to be used later by the symbolic debugger.
3. Machine code generation. Now that all addresses are known, the input string can be scanned again to produce both a listing with the symbolic code and the machine code, and a second file containing the object, relocatable or executable code.

3.2 Assembler informal code

The following pseudo-code gives some idea, in a top-down fashion, about how a real assembler works.

```
Programme Assembler
    Macro_expansion
    Initialise_Tables
    Pass_1
    Pass_2
    Print_summary
end{programme}

Proc Pass_1
    for each line
    do{ get_beginning_of_line
        if label then check_label
                 else ignore
        get_instruction
        analyse_instruction }
end{pass_1}

Proc ignore
end

Proc Check_label
    search_symbol_table
    if present then error_label
```

```
            else add_label
end

Proc Analyse_instruction
    get_opcode
    search_opcode_table
    case class_0 then error_opcode
         class_1 then class_1_code
         class_2 then class_2_code
         class_3 then ...
end

Proc Class_1_code
    get_address
    if illegal then error_address
    add length_opcode + address to programme_counter
end

etc.

Proc Pass_2
    for each line
    do{ get_instruction  (ignore label part)
        analyse_instruction }
end

(modified proc, that is called both by Pass_1 and Pass_2 :)

Proc Class_1_code
    get_address
    case first_pass then  if illegal then error_address
                          add length_opcode + address to programme_counter
    case second_pass then assemble\_code (opcode & address, using symbol values)
end

etc.
```

3.3 Assembler Tables

An assembler uses at least a symbol table and an opcode (instruction) table.

3.3.1 Symbol Table

What do we need ?

1. name / symbol
2. address in memory
3. type of symbol (variable, label etc.)
4. line where the symbol is defined

5. lines where the symbol is used

Note that :

- 1 and 2 are necessary
- 3 may be useful
- 4 and 5 are necessary for errors and cross-reference
- 5 is highly variable in size
- during Pass_1 we add and search through the table
- during Pass_2 we only search through the table

3.3.2 Opcode Table

What do we need ?

1. opcode name / symbol
2. opcode value
3. instruction class
4. instruction length

Note that :

- 4 is not necessary, could be better in instruction class.

This table is fixed for a given processor, we have no need for additions or updates.

3.3.3 Opcode Table Organisation

The best organisation for the opcode table is probably a 2D table :

Symbol	value	class	cl. name
CLRB	5F	1	'cpu'
LDA	A6	2	'mem'
BGT	2E	3	'jump'

3.3.4 Pseudo Opcodes

Most assembler need instructions that help structuring the programme, but do not produce any executable code :

1. set symbol value for example : `ROSY EQU 0`
2. organise memory for example : `ORG`
3. reserve space (set to values)for example : `FCB , FCC, FDB`

4. conditional assembling for example : `IF(cond) ...ENDIF`

5. macro, for example : `MON`

- 1 and 2 produce nothing, they are used only in `Pass_1`
- 3 initialise memory, but do not produce executable code
- 4 allows conditional production of code (with or without floating point processor etc.)
- 5 help produce repetitive code (and pseudo-code !)

3.3.5 Remarks

- by modifying the opcode table, and the class_code procedure, the assembler can be updated to new hardware.
- the assembler transforms our symbolic programme into machine codes, of transforms the hardware into something more sophisticated !

Can we go further ?

Can we add an other layer to make a 'machine' which is able to understand more natural language ?

$\Longrightarrow$ **High Level Languages**

4 Symbol Tables

Remarks :
The main entries are **'symbols'** with :

- many characters (meaningful symbols may be quite long)
- very few possibilities used

For example, 4 letter words correspond to 26^4 or 456'976 possibilities,
or 6 ASCII characters corresponds to 96^6 or $7.8 \cdot 10^{11}$ possibilities.

It is therefore strictly impossible to reserve an entry for each possible symbol.

Here are some possibilities to organise these tables in a reasonable form :

- lexicographic order (as in the dictionary)
  $\Longrightarrow$ binary search
- hashing (randomized)
  $\Longrightarrow$ hash - coding, random access
- 'letter'-Tree or B-Tree (list)
  $\Longrightarrow$ list search

4.1 Hash Table

The symbols chosen by users are usually not random at all. The idea behind hashing is to transform each symbol into an integer K corresponding to the position of the symbol in the table, in such a way that :

- $0 \leq K < N$ = size of the table
- K is unique for a given symbol
- K has no memory of the characters forming the symbol, that is K is randomly distributed although the symbols are not.

For example : if the symbol is represented by $C_1C_2C_3 \ldots C_n$
then

$$Hash[symbol] = \left[\sum_{i=1}^{n} i \times C_i \right] \bmod N = \left[\sum_{i=1}^{n} [\, i \times C_i \bmod N \,] \right] \bmod N$$

- it is very fast to
  - search through the table
  - add new entries to the table
- it is slowed down by collisions (two or more different symbols with same key K)
- it contains no pointer

When the table becomes full, the risk of collision is very high.

4.1.1 Solutions

- have an overflow secondary table
- rehash with an other function
- take next free entry

Rule of thumb :

> Hash tables are very good if less than half full

4.1.2 Signal Generator Symbol Table

(see Software Technics)

Let us use the following hash function, assuming a table size of 16 :

$$h(symbol) = \sum i \times C_i \mod 16$$

Then

$$\begin{aligned} h(SAW) &= (1 \times 19 + 2 \times 1 + 3 \times 23) \mod 16 \\ &= (3 + 2 + 3 \times 7) \mod 16 \\ &= 10 \end{aligned}$$

and in a similar way :

h(END) = 13	h(EXIT) = 0	h(HZ) = 12
h(KHZ) = 9	h(MMS) = 0	h(MS) = 3
h(SAW) = 10	h(SIN) = 15	h(SQR) = 11
h(SQU) = 4	h(TRI) = 3	

They are 2 collisions (on addresses 0 and 3). We have chosen to take the next free position.

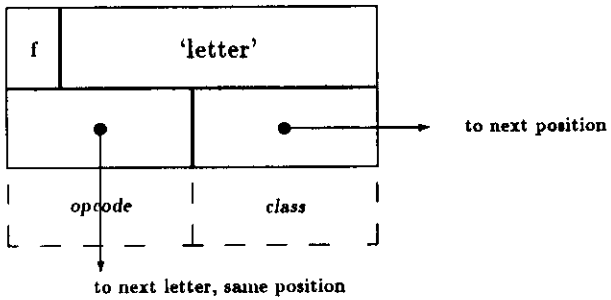
The table looks, when filled, as :

K	Symbol	address
0	EXIT	...
1	MMS	...
2		0
3	MS	...
4	SQR	...
5	TRI	...
6		0
7		0
8		0
9	KHZ	...
10	SAW	...
11	SQR	...
12	HZ	...
13	END	...
14		0
15	SIN	...

4.2 Letter - Tree

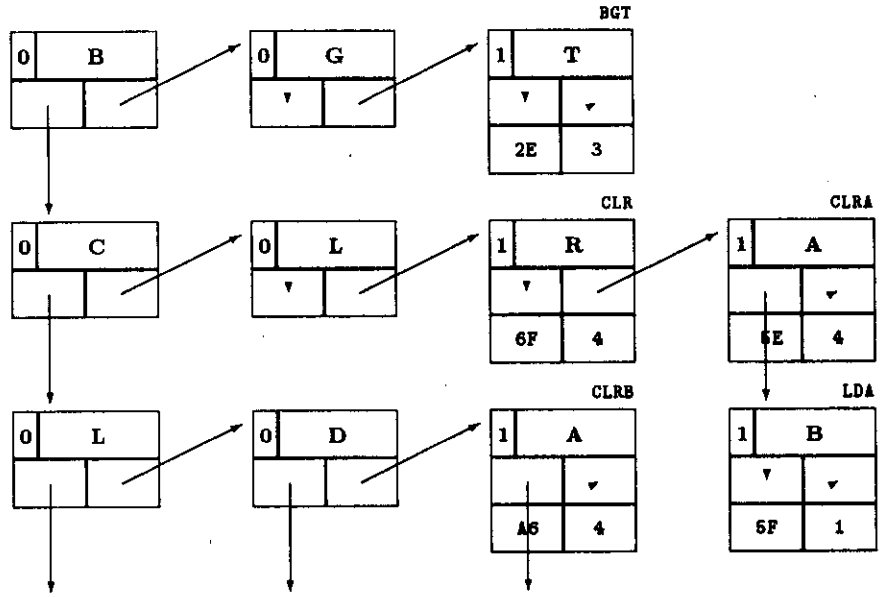
In this organisation, we build a cell for each letter used in a given position in a symbol.

4.2.1 General Form of Cell



- f is the opcode flag, 1 if the letters encountered up to now correspond to an existing opcode symbol
- a pointer with a value of zero indicates *NIL* or *END OF LIST*
- it use a lot of space (many pointers)
- it is very easy - to search for a given symbol
  - to add new symbol

4.2.2 Letter - Tree for instructions opcodes

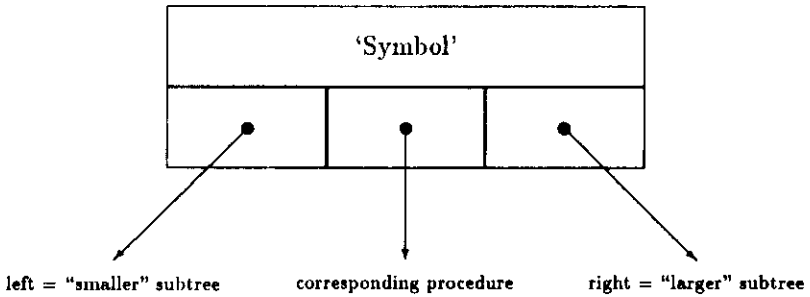


▼ ▾ indicate NIL, or nothing further in this list

long arrows point to the next element in the list

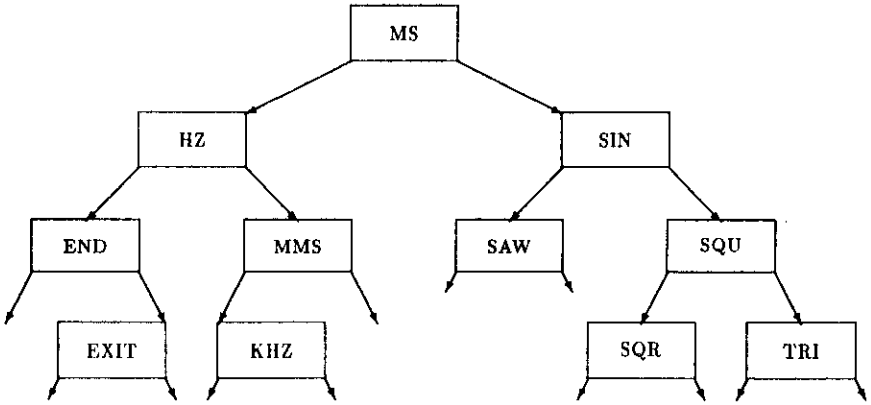
4.3 Binary Tree

4.3.1 General Form of Entry



The upper box (8 characters ?) contains the characters of the symbol. The next three boxes contain pointers, that is addresses of other boxes (left and right subtree), or of the code (procedure) associated with the symbol.

4.3.2 Signal Generator Binary Tree



4.3.3 Remarks

Many different Binary Trees can be build with the same set of symbols.

- a tree is balanced if the length of any path is $\leq \log_2$ (*number of symbols*).
- a tree can be optimised in minimising $cost = \sum path\_length \times prob(target\_symbol)$

5 High Level Languages

5.1 Comparison between Assembler and High Level Languages

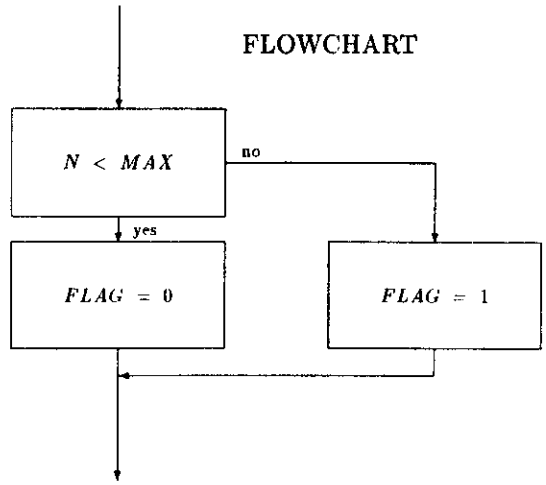
In Assembler :

- There is a strong relation between the symbolic instructions and the machine code. It is therefore *hardware dependent*
- Macro instructions and subroutine are like new instructions with no direct counter part in hardware
- There is a small relation between the symbolic instructions and the work to be done $\Longrightarrow$ too much secondary work

In High Level Languages (HLL) :

- It has no relation with hardware (almost)
- It use 'natural' expressions
- It may be well standardised
- Large libraries are available
- Execution can be slower (in some cases much slower)
- Direct access to hardware is rarely possible
- Programmer and user can be relatively well protected against errors

5.1.1 Small Example



Assembler

```
        LDA     N
        CMPA    MAX
        BLT     ZFLAG
        LDA     # 1
        STA     FLAG
        BRA     DONE
ZFLAG   CLR     FLAG
DONE    EQU     *
```

HLL

```
if N < MAX      then FLAG = 0
                else FLAG = 1
```

Remarks :

- HLL follows strict formal rules and are so 'readable' / translatable by computer
- HLL is very easy to read and understand by user
- all three notations (flowchart, assembler, HLL) are equivalent

5.1.2 HLL and Virtual Machine

HLL make available an abstract - virtual machine

- with instructions adapted to applications
  as `if` , `sqrt` , `sin` , `read_disc` etc.
- with data type (structure) also adapted to applications
  as `integer` , `real` , `complex` , `vector` , `record` , `string` etc.
- part of the virtual machine may have correspondent in the hardware,
- other part will be dealed with by software library routines (run-time libraries)

5.2 Types of Statements

They are basically 3 types of statements :

1. declaration statements
2. normal statements, execution - commands
3. control statements

5.2.1 Declaration Statements

- specify the type of variables
- specify simple or complex data structures
- specify global, local or imported variables and procedures

Operators will have different meaning depending on data type :

For example : `x = 3, y=5, z=x/y`

```
if type is integer,     then z = 0
          real,              z = 0.6
          complex            z = (0.6,0.0)
```

- in FORTRAN, (and other HLL) default data types are given to undeclared variables
  $\Longrightarrow$ less work for programmer,
  less semantic error detection
- in PASCAL (Algol etc.) all variables must be declared
  $\Longrightarrow$ strong typing (cross checking allows the detection of many errors)
- in FORTRAN all variables (except in COMMON) are local or imported, without any control of type
- some HLL allow the specification of new data types and associated operators (ex Algol 68, ADA, FORTH)

5.2.2 Structured Data

1. simple structures : complex, vector, matrix etc.
   for example :

```
dimension result(1:100),sigma(1:10),covar(1:10,1:10)
do i=1,10
sigma(i) = sqrt(covar(i,i))
enddo
```

   or :

```
counts = array[1..100] of integer
covar = array[1..10,1..10] of real
```

2. complex structures : made of simpler ones
   for example :

```
type months = (january, february, ... december) (user defined)
date = record
day :  1..31
month :  months
year :  integer
end

date.day = 25
date.month = june
```

```
date.year = 1986
⋮
print date                (will print complete date)
```

Some HLL will allow to define operators on new structure, to incorporate checking routines etc.

In FORTRAN, you will have to play tricks, using equivalence, and only with standard types.

5.3 Subroutines and Procedures

They add higher level, more specific instructions to your language

Goal : Build up a new language, in which you can naturally express your needs / applications.

- use as many layers as needed
- put them in libraries
- hide the 'how' in lower levels
- whenever possible, reuse existing routines

6 Programme controls

6.1 Programming syntax

6.1.1 Conditionals

Only one of two possible block is executed :

if *condition* **then** *body_true*
else *body_false*

Only one of many possible block is executed :

case *expression* **of**
value_1 : *body_1*
value_2 : *body_2*
⋮
end

6.1.2 Counting loops

A given block is executed an exact number of times :

for *var* = *first_value* **to** *last_value* **do** *loop_body* *loop_body* is usually executed at least once

6.1.3 Conditional loops

A given block may or may not be executed many times depending on a condition. The condition may be set inside the block.

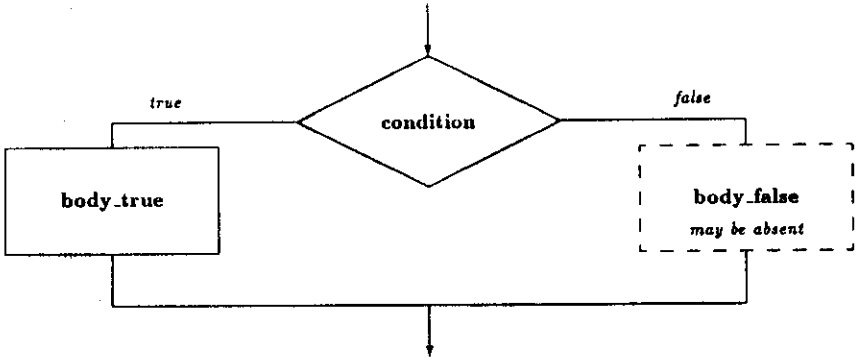
while *condition* **do** *conditional_body* check done before first execution

do *conditional_body* **until** *condition* check done after first execution

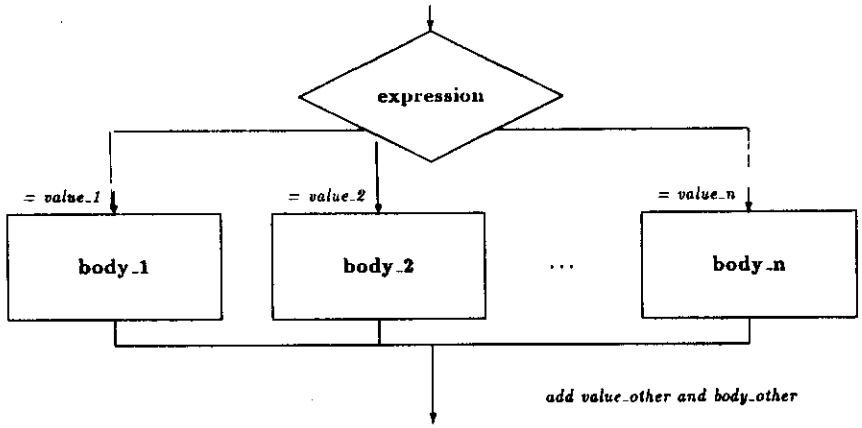
do *body_1* **if** *condition* **then_break** **else** *body_2* **repeat** check done after first execution of *body_1*, before first execution of *body_2*

6.2 Flowchart for programme controls

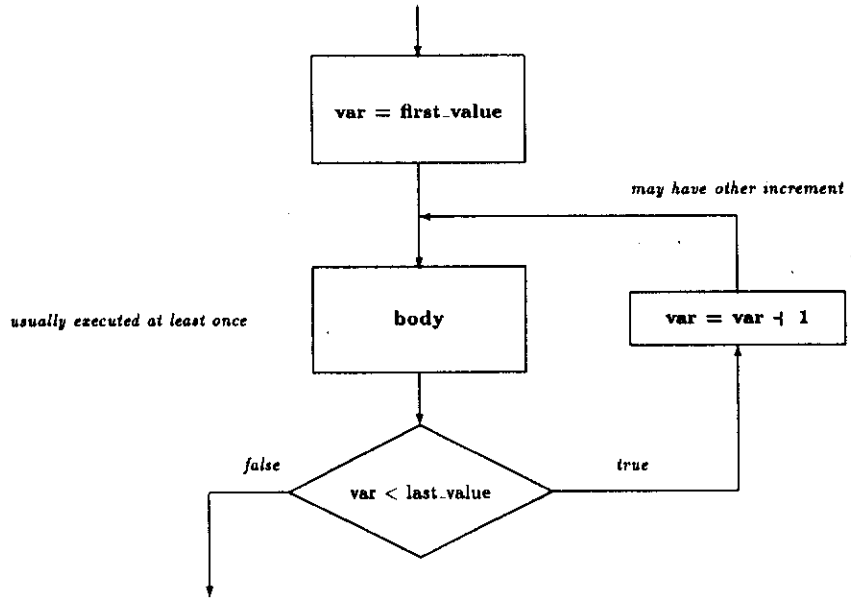
6.2.1 if ...



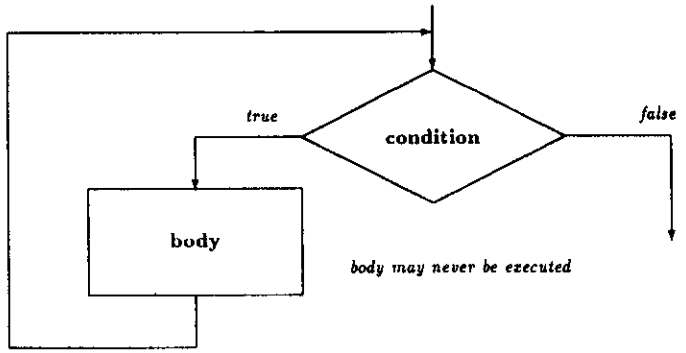
6.2.2 case ...



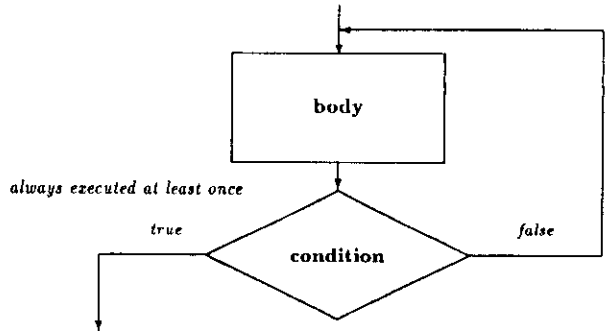
6.2.3 for ...



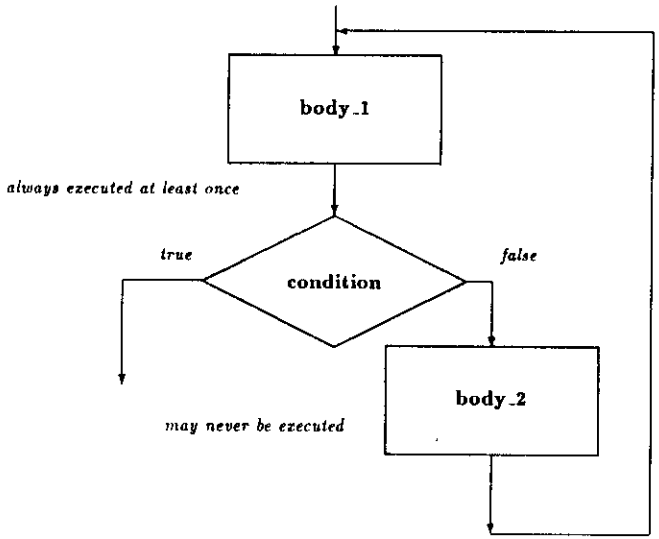
6.2.4 while ...



6.2.5 do ... until ...



6.2.6 do ... if ... repeat



Does your loop terminate in all cases?

6.3 Examples

- `if x > xmax then x = xmax`
- ```
  if a < b then min = a ; max = b
           else min = b ; max = a
  ```
- ```
  case symbol_class of
       0 :  not_a_number
       1 :  integer_length
       2 :  real_length
       3 :  2 * real_length
       otherwise unpermited_symbol_class
  end
  ```
- `for month = 1 to 12 do print(mean_value(month))`
- `while amount < reserve do payment ; reserve = reserve - amount`
- `do x = ( N + x * x ) / ( 2 * x ) until abs( N - x * x ) < 1`
- ```
  do read_tape
  if eof then break
  else print_data
  repeat
  ```

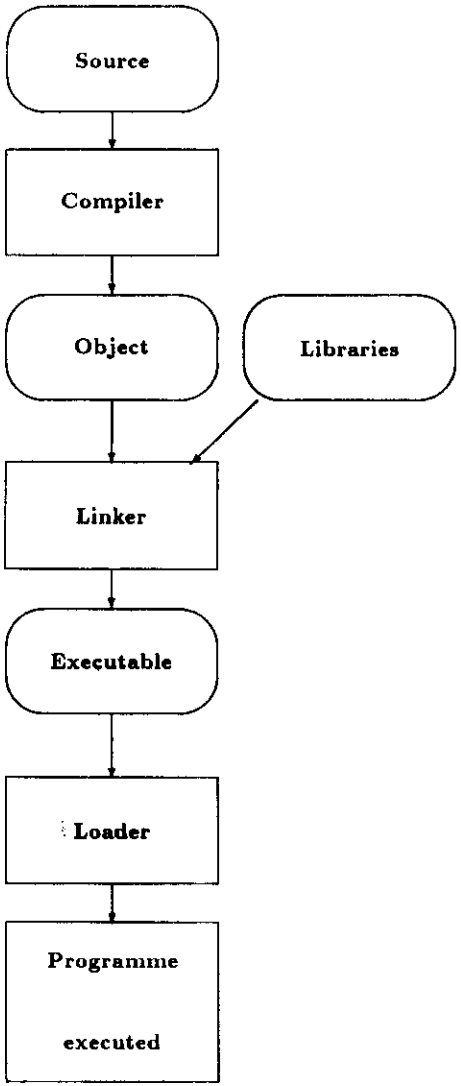
6.4 Correctness proof

This is just a very sketchy overview of a subject considered by many to be the key of any good and secure programming.

- for each block :
  - find what is **logically invariant**
  - verify that your code does keep it **invariant**
- Verify that the information is **always** available when needed
- Verify that the loops **always** terminate correctly
  ```

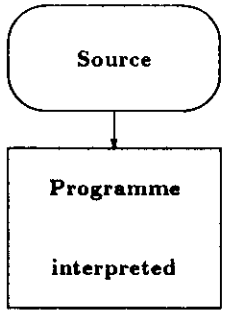
7 Compiler / Interpreter

7.1 Compiler



Slow cycle, fast execution

7.2 Interpreter



Lines of source are analysed and immediately executed

Cycle is fast, execution is slow

Memory usage is bigger

Otherwise, Compilers and Interpreters offer the same possibilities, the same degree of abstraction etc.

7.2.1 Remarks

- Many compilers produce intermediate code (ex assembler etc.)
- Both compilers and interpreters exist for some languages (ex Basic)
- Some interpreters produce code that can be reused later on
- Forth works both as an interpreter and as a compiler
- The "user interface" of an Operating System is an interpreter (usually very simple !)

7.3 multipass compiler

lexical analysis (orthograph)
recognize reserved words
build-up symbol table(s)
report unacceptable symbols

syntax analysis (grammar)
recognize logical relations between symbol "words"
report grammatical errors

code generation (translation)
generate op-codes, usually in assembler
allocate memory, registers, etc.

code optimisation
reorganise code for speed-up, locally and globally

assembler
produce definite object code

7.4 Jump tables for interpreters

Like the assembler and compilers, an interpreter needs **Symbol** and **Op-code** tables (with higher level instructions instead of hardware ones)

Then, instead of producing machine codes, it gives an offset into a table of jumps to various routines, which will eventually jump back to the analysed.

Example :

```
Jump_table    BRA    Add_integer
(+1)          BRA    Add_real
(+2)          BRA    Add_complex
  ⋮
(+74)         BRA    Sqrt
  ⋮
(+96)         BRA    Read_disc
(+97)         BRA    Write_disc
  ⋮
(offset)      BRA    Address of routine
```

7.4.1 Remarks

- Jump tables can be replaced by procedure address tables with indexed jumps.
- Jumps can be replaced by **calls to subroutine**, so that they can call other subroutines (of lower level) present in the table
- Incremental interpreter can be build if new entries can be added by the interpreter (see Forth)
- Parameter passing must be very carefully planed (use of stack, global variables etc.)

7.5 P-code, M-code and Co.

7.5.1 History of Pascal and Modula-2

Algol 58 + 60 : before computers where widely available.
(very) good theoretical ideas, but no implementation,
while IBM pushed FORTRAN, which was much worst, but was based on existing compiler.

FORTRAN = **FORM**ULA **TRAN**SLATOR

exception : Burroughs 5000 (and its successors) whose hardware is based on Algol (no assembler, pure stack architecture)

1960-1968 : Algol 68 : contains every thing you can think of
$\Longrightarrow$ very difficult to implement, to use, to teach ...

Pascal : Wirth (and Hoare) separated from the Algol 68 team, and designed Pascal as a simple, limited language for teaching good programming. They produced immediately a compiler ("Zurich" compiler for CDC)

7.5.2 The Pascal compilers

The first Pascal compilers where written in Pascal (bootstraping).

The goals where :

- check the theoretical design
- teach Pascal and use it immediately
- have a large programme to demonstrate the use of Pascal

Different versions have been build :

Pascal-S with some limitations,
produce internal S-code that can be immediately executed (interpreted, no object file). The S-code is processor independent. This version is particularly efficient for small programmes with fast turnaround, whose compilation may cost more than execution.

Pascal-P full implementation,
produce P-code in object file, for later execution (interpretation) or translation in pure machine code. The P-code is processor independent.

Optimised Pascal full implementation, but optimised for a given processor. Only the first passes of this compiler is processor independent. Very good for big programmes whose execution cost dominates.

UCSD and Turbo Pascal developed more recently for personal computers. They both offer not only a compiler (not very efficient, UCSD Pascal produces P-code that is later interpreted) but also a powerful environment with editor, interactive debugger, graphics etc.

7.5.3 Bootstraping process

Start with a very limited, unoptimised compiler, write a new better one, using the previous to compile the next, and iterate !

7.5.4 What is P-code, S-code, M-code

It is the op-codes of a non-existing processor that is :

- very near to real processors
- produce optimised translation from Pascal
- based on stack(s) architecture

Around 1982, Wirth proposed **Modula-2** (against ADA) for real-time and multitasking, and the M-code as intermediate language. With the exception of the Lilith, the M-code is usually translated into machine instructions.

At execution, a small interpreter is loaded with the P/S/M-code to realise the virtual P/S/M processor.

NB: Western Digital (ATT) has a fast μprocessor (similar to J11 from DEC) that execute directly P-code instructions. Wirth him self has build a special processor (the Lilith) that also execute directly M-code instructions. Moore has build a single chip (Novix 4000) that execute directly Forth high level instructions.

7.6 Remarks about ADA and Modula-2

Both are based on the same ideas :

- produce highly secure code for real-time, multitasking environment
- provide all facilities for tasks cooperation (semaphore, fork, 'rendez-vous'etc.)
- help abstraction and information hiding
- import libraries with invisible informations (hided) for checking
- assign new operators with new data structures (in libraries)

but they differ very much in :

ADA is huge, does every thing, took 6 years to have the first compilers (!),
is very well defined and standardised (DoD)
has no subset, no extension
ADA is now available on many computers, although most compilers are still very slow.

Modula-2 is small, perfect for small, manageable applications, use a very small, provable, kernel, had a compiler (written in Modula-2 and/or Pascal) available immediately with it's definition, is not standardised, nor supported by any major manufacturer.

Theoretically, ADA is more secure, but can we trust a monster ?

It seems that ADA will always come in a complete ADA-environment, including editor, loader, file handler, linker etc. all special purpose, necessary to make it work, but also a big help for the programmer.

8 Function of an Operating System

8.1 Goals

- provide the interface between
  - the user
  - the processor
  - the external world (I/O)
- maximise
  - security
  - standardisation
  - easy access
- control resource sharing
  - time / memory space
  - I/O
  - disc files
- provide general purpose tools
  - file system
  - assemblers
  - compilers
  - editor(s)
  - sort/merge
  - debugger
  - general facilities

8.2 Operating System layered model

4	User Application	Usercommand interpreter
3	System library routines	
2	I/O driver, System Programmes	
1	Kernel	

- the Kernel should be very small, very secure and protected
- most layers are in fact multilayered
- calls and informations can only move between adjacent layers (up or down)

8.2.1 Layered model : example

layer	programme instructions			system commands		
4	Data login prog call create call read_data call store			> list_file > delete_file > run_programme > send_mail		
3	proc create	proc store	proc delete	proc read	proc print	proc ...
2	read disc	write disc	read A/D	write D/A	lock disc	start prog.
1	handle interrupts, semaphores, hard locks etc.					

8.3 Why have layers ?

8.3.1 Disadvantages

- no direct access/control over what is happening
- slow down by multiple transfer
- difficult to change rapidly

Layered model is not always better

8.3.2 Advantages

- no needs for the user to provide OS functions
- no needs to know anything about hardware
- no needs to adapt to new or changed hardware etc.
- good protection against errors
- standard interfaces
- system as seen by users are very stable

Layered model is mostly much better

9 Monitor

Monitors are considered here as elementary Operating System, available on some small single card computers.

The set of commands available is :

- read content of memory
- put data in memory
- start execution at given address, provide procedure calls to :
  - display results
  - read in 'keys'
  - do I/O
  - get/send ASCII characters on RS-232 lines, etc.
- debugger
  - read/set memory/register, in any format (opcode, decimal, octal, character)
  - set break-point(s)
  - execute single step (one instruction at a time), or of groupes of instructions

10 UNIX

10.1 Main characteristics

Unix was designed around 1970 by Kernighan and Ritchie from the Bell Laboratories as a much simpler version of the Multics Operating system developed at MIT. For many years, Unix was used only inside Bell labs and in some universities.

- it takes some (the best) ideas from project MAC at MIT ($\Longrightarrow$ Multics) but avoids making a monster
- make things as simple, as uniform as possible
- make it processor independent, write it in a high-level language (C)
- provide tools for users, make source code available for corrections, and also to reuse part of existing code
- provide powerful yet simple multitasking kernel, with task cooperation facilities
- very simple, yet powerful and uniform file system
  - sequential character string
  - hierarchical directories
- very powerful command interpreter (similar to C) with variables, conditionnals, loops, procedures etc.
- redirected I/O (pipes) for parallel/sequential execution
  example : `acquisition | sort | print`
  print sorted results produced by the data acquisition programme

Most tools are very short programmes that do only a single operation (like sort, print, extract lines etc.)

The combination of these tools using pipes and a powerful system command language gives Unix its power.

Unix is available for many processors, from 8 bits micro to large Amdhal and Cray.

10.2 Some bad points of Unix

- it is old, has limitations dating to 1970 hardware
- was never standardised.
- main versions available :
  - ATT Unix version V.2 (some older System III)
  - Berkeley Unix 4.2 or 4.3 (mainly for Vax, Sun etc.)

- has a very poor user interface with
  - meaningless names
  - no protection against catastrophic errors
- has good C but bad Fortran compilers
- has
  - many many tools for computer scientists,
  - many tools for physicists etc.
  - very few tools for commercial applications
- is not very good for real-time applications (exception HP-UX, Modcomp)

Most modern operating systems (MS-DOS, FLEX, CP/M etc) have been inspired by Unix !

10.3 Other common Operating System for microprocessors

10.3.1 OS 9

- Developed originally for Motorola 6809, now also available for Motorola 68000.
- Support Basic O9 (almost nothing to do with Basic), Pascal, C, Forth, word processors etc.
- Can be very powerful, specially in a lab and if you can help.
- It is an open system, on which you can add new I/O handlers etc.

10.3.2 MS-DOS

- Developed for the IBM-PC with Intel 8088, 8086 or 80286 processors
- Probably now the most used OS $\Longrightarrow$ de facto standard, although very few versions are really 100 % compatible.
- Support all sorts of languages, programme, data base, word processors etc.
- Can be very good both for lab and commercial applications.

11 Editor

To manipulate text not numbers

- Most interpreters have an internal core editor (+ load and save)
- Some monitor commands implement also a primitive core editor

11.1 Typical commands

find, change, add, delete, display, copy, move, insert, repeat etc.
on character(s), word(s), line(s), paragraph(s) etc.

11.2 Environment

11.2.1 Line editor

as used in Basic and Flex, for example.

- references by line number
- line numbers remain fixed, $\Longrightarrow$ keep gaps left to insert new lines
- display current line for modification
- restricted manipulations

unacceptable for text or large programmes

11.2.2 Context editor

It is now the most commonly available editor.

- references by strings in text
- relative and absolute current line number
- allow local and global modifications
- powerful 'regular expressions' (define string by template)
- display current line for immediate use

acceptable for all purposes, can be very fast, but you do not see globally what you will get at the end

11.2.3 Full screen editor

Very common on supermini and large computers.

- references by pointing (cursor, mouse etc.) and string
- display the 'page' around last change
- needs
  - high resolution screen
  - good cursor commands
  - high speed communication with memory

very powerful, you see what you get. Implemented on modern (single and multiuser) interactive computers, but produce heavy demand on I/O

11.2.4 Language (context) sensitive editor

Are still not very common

- usually combined with full screen editor
- provides immediate lexical and syntax analysis.
- report errors immediately
- can also be combined with symbolic debugger on different windows
- some prototypes verify assertions (logical invariants) provided with blocks (structured programming) for correctness proof

good to produce syntax error free code (the easiest to correct !)

Improvements have been very slow !

Is it due to user conservatism or trade-off between speed and power ?

11.3 Remarks

No standard editor exists.
Every processor, every system has a different editor.
But, as for animals, most of them can be traced back in evolution to very few originators.
Small differences can therefore be more dangerous!

12 Debugger

Almost all programmes contain error (= bugs in relay)

- add guards while coding
- prepare simulated input, first simple (easy to trace by hand), then more complex (difficult)
- Debug each module alone, then in small integration
- chose critical point where you know what you should get if previous step are correct.
- advance by small steps
  - from input forward
  - from output backward
- analyse wrong result to see what/where this value comes from
- try all (very) improbable cases

Rules :

> if some thing can go wrong, it will !
> if an error can be damaging, it will !
> if it is very improbable, it will still exist !

12.1 Use of a non-symbolic debugger

They usually need absolute and/or relative addresses.
prepare them in advance :

- list of entry-points
- offset to them (critical points or interesting variables)
- variable addresses
- as given by
  - assembler
  - compiler
  - linker

Symbolic debugger can use symbols instead of numerical values for entry-points and variables.

12.2 Symbolic debugger

They need a file, produced by the assembler/compiler/linker with :

name, address, type for each symbol

Then names can be used in place of absolute or relative addresses.
It is much easier to use, more user friendly, but do not provide extra facilities.

13 File system

13.1 Generalities

- If correctly generalised, can cover all I/O, including internal data transfer.
- A file is a collection of records,
  - it has a name
  - a set of permissions
  - a set of special characteristics
- A record is a consecutive chain of characters (words).
  - it has a record number
  - possibly one or more access key(s)
  - a set of permissions
- Records can be of fixed or variable length
- Records can be accessed
  - sequentially
  - randomly
  - through index

13.2 Manipulation on files

13.2.1 Global operation on files

create, erase, delete, open, close, rename, copy

13.2.2 Operation on records

find, read, write, add, erase, move, copy

13.2.3 Pseudo operations on file/record/character

lock : prohibit momentarly any other access
change : read/write/execute/erase permission (passwords)

All these commands should be completely independent from the hardware on which the file resides.

**files and records are abstractions,
implementation informations are hidden !**

13.3 Special files : directories

Contains user's given and hidden implementation informations

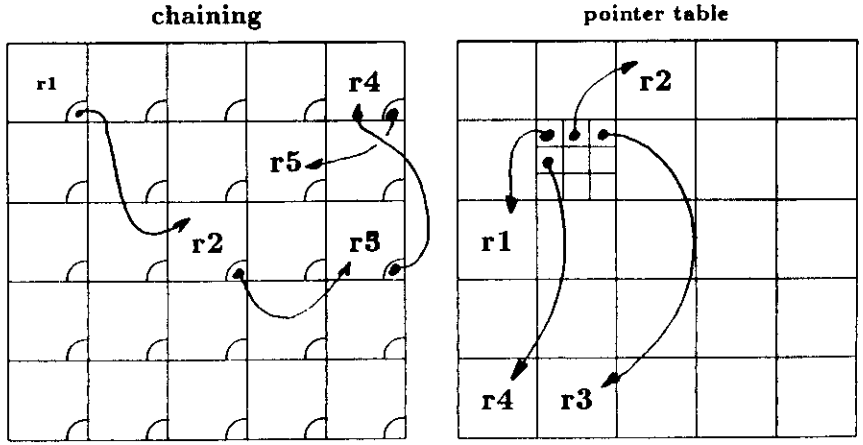
13.4 Physical support

- **magnetic tape and cartridge**
- **floppy, removable/unremovable hard disc**
- **optical disc (ROM, PROM,RAM)**
- **bulk storage**
- **size (1986) 10^5 to 10^{12} characters**
- **access time from 10^{-5} to 10^2 seconds**
- **usually divided in unit/track/sector**
- **the sector is the smallest physically accessible unit, it has no relation with records**

13.4.1 File organisation on physical support

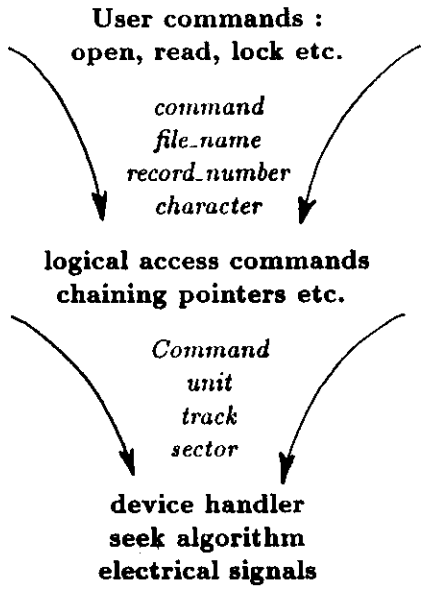
To make efficient use of support, records belonging to a given file may be scattered all over in many different physical subfiles (by sectors or even characters).
Some means are necessary to find them back rapidly :

- **chaining pointers**
- **pointer tables (map)**
- **bit maps , clusters ...**
- **etc.**



In both cases, we need a 'file' of free sectors.

13.5 Three layers of abstraction



14 Think

Think !

- think before doing
- think while doing
- think after having done
- your are responsible, you are the master
  never give it to μP
- μP must obey, not dictate

Think small !

- 'Small is beautiful'
- keep things manageable, under control
- use small modules

Think with others !

- do not reinvent the wheel
- make your work shareable
- build-up libraries
- accept help, call for help
- the others can and must think too

Think on your own !

- do not accept buzz words for granted
- adapt to your own country
- do not destroy your richness
- never accept dogma