HAC Ada Compiler User Manual

https://github.com/zertovitch/hac
https://sourceforge.net/projects/hacadacompiler
Gautier de Montmollin gdemont@hotmail.com

gdemont@hotmail.com
https://github.com/zertovitch/hac

Ed.	Release	Comments	
1	20201111	Initial release	sr
2	20201119	Add introduction and general organization	sr
6	20201121	Update document properties, HAC runtime & vector logo	sr
14	20201122	Add PDF documents properties and keywords, citations, update HAC runtime, progress review	sr
21	20201214	Add illustrations, new runtime functions, predefined types, HAC basic programming [to be translated]	sr
24	20201219	Add code examples in HAC runtime chapter, add exit code issue	sr
26	20201220	Add and populate chapter HAC firsts programs, translate first program	sr
27	20210303	Change hax to hac	sr
33	20210308	Runtime update, various improvements, reorganize chapters, fix many typos	sr
38	20210309	Final review for first public edition	sr
48	20210312	Change main title, Gautier email, replace many Hac typos, fix various typos, first release	sr
49	20210313	Add hyperlinks on summary [direct clic with PDF, ctrl-clic with ODT]	sr
53	20210409	Correct many typos, alphabetically reorder API in HAC runtime instead of hat.ads source ordering.	sr
54	20210410	Add HAC examples and HAC Advent of Code examples, links and build, reformat footers	sr
71	20210412	Add information and architecture section. Improve HAC runtime documentation	gm
75	20210418	Change Humanist 521 BT font to Airbus cockpit free font designed by Intactile[1]	sr
76	20210419	Add chapter “Programming basics”. Improve documentation, fix various typos	sr
82	20210420	Add information about Shell_Execute procedure under Linux	sr
84	20210731	Minor updates and typos	sr
86	20221012	Replace HAL by HAT, new HAC runtime introduction, fix some typos	sr
88	20230121	New standard Ada Logo	sr
89			

[1] https://b612-font.com under Open Font License, replaced the Humanist 521 BT licensed by Monotype.

❑ Author

Gautier de Montmollin - gdemont@hotmail.com

❑ Manual

Stéphane Rivière [Number Six] - stef@genesix.org
Gautier de Montmollin - gdemont@hotmail.com

The “Excuse me I’m French” speech - The main author of this manual is a Frenchman[2] with basic English skills. Frenchmen are essentially famous as frog eaters[3]. They have recently discovered that others ~~forms of communication~~ languages are widely used on earth. So, as a frog eater, I’ve tried to write some stuff in this foreign dialect loosely known here under the name of English. However, it’s a well known fact, frogs don’t really speak English. So your help is welcome to correct this bloody manual, for the sake of the wildebeests, and penguins too.

❑ Edition

1 89 - 2023-01-21
https://this-page-intentionally-left-blank.org

[2] This do not apply to Gautier, author of HAC, who is a proud multilingual Swiss citizen ;]

[3] We could be famous as designers of the Concorde, Ariane rockets, Airbus planes or even Ada computer language but, definitely, Frenchmen have to wear beret with bread baguette under their arm to go eating frogs in a smokey tavern. That’s *le cliché* :]

gdemont@hotmail.com
https://github.com/zertovitch/hac

Summary

Introduction
1 About HAC....10
2 HAC purposes....10
2.1 Thanks....11
2.2 HAC history....11
3 Syntax notation....12
4 About Ada....12
4.1 Introduction....12
4.2 Why use Ada....13
4.3 The ending word....13
Getting started
1 Distribution....14
1.1 Directories....14
1.2 Key files....14
2 Get a compiler....15
2.1 FSF GCC GNAT & AdaCore GNAT Community Edition compilers....15
2.2 Linux....15
2.3 Windows....15
3 HAC build....15
3.1 Linux....15
3.2 Windows....16
HAC language
1 Design points....17
2 Language subset....17
3 Predefined types....18
3.1 Integer....18
3.2 Real....18
3.3 Character....18
3.4 String....18
3.5 VString....18
3.6 Boolean....18
4 General capabilities....18
5 Keywords and special characters....19
5.1 Special characters....19
5.2 Reserved words....19
6 Metrics....19
HAC runtime
1 Introduction....20

2 Real functions....21
2.1 Arctan....21
2.2 Cos....21
2.3 Exp....21
2.4 Log....21
2.5 Round....22
2.6 Sin....22
2.7 Sqrt....22
2.8 Truncate....23
3 Real operators....23
3.1 **....23
4 Randomize functions....23
4.1 Rand....23
4.2 Rnd....24
5 Characters functions....24
5.1 Pred....24
5.2 Succ....24
6 VString functions....24
6.1 Element....25
6.2 Ends_With....25
6.3 Head....25
6.4 Index....26
6.5 Index_backward....26
6.6 Length....27
6.7 Slice....27
6.8 Starts_With....27
6.9 Tail....28
6.10 Tail_After_Match....28
6.11 To_Lower....29
6.12 To_Upper....29
6.13 Trim_Left....29
6.14 Trim_Right....30
6.15 Trim_Both....30
7 VString operators....30
7.1 +....30
7.2 *....31
7.3 &....31
7.4 =....31
7.5 <....32
7.6 <=....32
7.7 >....32
7.8 >=....33
8 Type conversion functions....33
8.1 Chr....33

8.2 Float_Value....33
8.3 Image....33
8.4 Image_Attribute....34
8.5 Integer_Value....34
8.6 Ord....34
8.7 To_VString....35
9 Console i/o functions....35
9.1 End_Of_File....35
9.2 End_Of_Line....36
9.3 Get_Needs_Skip_Line....36
9.4 Get....36
9.5 Get_Immediate....37
9.6 Get_Line....37
9.7 New_Line....37
9.8 Put....38
9.9 Put_Line....38
9.10 Skip_Line....39
10 File i/o functions....39
10.1 Append....39
10.2 Close....39
10.3 Create....40
10.4 New_Line....40
10.5 End_Of_Line....41
10.6 End_Of_File....41
10.7 Error handling....41
10.8 Get....41
10.9 Get_Line....42
10.10 Open....42
10.11 Put....43
10.12 Put_Line....43
10.13 Skip_Line....44
11 Time functions....44
11.1 Clock....44
11.2 Day....45
11.3 Month....45
11.4 Seconds....45
11.5 Year....46
12 Time operators....46
12.1 -....46
13 Semaphore functions....46
13.1 Signal....46
13.2 Wait....47
14 System....47
14.1 Argument....47

14.2 Argument_Count....47
14.3 Command_Name....48
14.4 Copy_File....48
14.5 Current_Directory....48
14.6 Delete_File....49
14.7 Directory_Exists....49
14.8 Directory_Separator....49
14.9 Exists....50
14.10 File_Exists....50
14.11 Get_Env....50
14.12 Rename....51
14.13 Set_Directory....51
14.14 Set_Env....51
14.15 Set_Exit_Status....52
14.16 Shell_Execute....52
HAC architecture
1 Introduction....54
2 Architecture....54
2.1 The builder....54
2.2 The compiler....55
2.3 The Virtual Machine interpreter....55
FAQ
1 Return codes....56
2 How to compile HAC programs with a “full Ada” compiler....56
2.1 HAC example....56
2.2 GNAT compilation....57
3 How fast is HAC ?....58
3.1 Timings compared to Bash....58
3.2 Conclusion....59
4 How to add a runtime procedure....59
5 How to add a runtime function....59
6 How to add bound checking to a runtime function....59
Programs examples
1 HAC firsts programs....60
1.1 A salute to the world....60
1.2 A simpler way to run HAC programs....61
1.3 A salute without use clause....61
2 HAC examples....62
3 HAC Advent of Code....63
3.1 2020 year....63
Programming basics
1 Tools....65
2 Analysis....66
2.1 Methods overview....66

2.2 Top-Down example....66
3 Modular and structured programming method....70
3.1 Introduction....70
3.2 Program's Structured Diagram....70
3.3 Pseudo-code....71
3.4 Functions....76
4 Boole algebra....76
4.1 Identities, properties and De Morgan's laws....76
4.2 Practical advises....77

Notes

1 To-do list....78
1.1 HAC....78
1.2 Doc....78
2 Issues....78
2.1 Github....78

Introduction

1 About HAC

HAC is a recursive acronym meaning HAC Ada Compiler. HAC isn't a native code compiler but a Virtual Machine compiler which comes with a very compact and monolithic run-time executor.

As the HAC author says: "HAC is perhaps the first open-source (albeit very partial) Ada compiler (and virtual machine interpreter) fully programmed in Ada itself. It wasn't written from scratch, but is based on a renovation of SmallAda, a system developed around 1990 and then abandoned."

2 HAC purposes

HAC can be used for small Ada sand-boxed prototypes, education and scripting.

As an education tool, HAC is an excellent Ada subset for programming introduction.

As a script language, thanks for its shebang handling and its useful environment functions, HAC is the most Ada compact and powerful script engine you ever dream for. The HAC compilation is straightforward. The executor is ridiculously small and fully standalone. Move the hac program to your path and voila!

Last but not the least, HAC sources are fully compatible with Ada compilers, through the compatibility package HAT !

2.1 Thanks

The authors of SmallAda, listed below, for making their work open-source.

Jean-Pierre Rosen for the free AdaControl tool which was very helpful detecting global variables stemming from SmallAda's code :

https://www.adalog.fr/en/adacontrol.html
https://sourceforge.net/projects/adacontrol

AdaCore for providing their excellent Ada compiler for free :

https://www.adacore.com/community

2.2 HAC history

Now: HAC is being made more and more usable for real applications, with Ada compatibility, modularity, a library with I/O, system subprograms...

2020: FOSDEM's Ada Developer Room: https://fosdem.org/2020/schedule/event/ada_hac

2013: January 24th: Day One of HAC: Hello World, Fibonacci and other tests work!

2009: A bit further trying to make the translation of SmallAda sources succeed (P2Ada was improved on the way, for WITH statements and modularity)...

1999: Automatic translation of Mac Pascal SmallAda sources o Ada, using P2Ada.

1989: SmallAda is derived from CoPascal; works only within two very system-dependent environments (a Mac GUI, a DOS GUI); two similar source sets in two Pascal dialects (Mac Pascal, Turbo Pascal).

1986: CoPascal (Schoening).

1975: Pascal-S (Wirth) - Reference: PASCAL-S, a subset and its implementation https://doi.org/10.3929/ethz-a-000147073

- Authors of SmallAda (in Pascal)

1990 Manuel A. Perez — Macintosh version
1990 Arthur V. Lopes — integrated environment for IBM-PC
1989 Arthur V. Lopes — window-oriented monitoring for IBM-PC
1988 Stuart Cramer and Jay Kurtz — refinement of tasking model
1987 Frederick C. Hathorn — conversion of CoPascal

- Author of CoPascal (derived from Niklaus Wirth's Pascal-S)

1986 Charles Schoening

- SmallAda sources

You can find the SmallAda sources and examples in the "archeology" folder. The Turbo Pascal sources files are smaller than the Mac Pascal ones, probably because of the memory constraints of the 16-bit DOS system. So the Turbo Pascal sources are especially cryptic and sparsely commented, full with magic numbers and 1-letter variables, many of them global.

3 Syntax notation

Inside a command line:
- A parameter between brackets [ ] is optional ;
- Two parameters separated by | are mutually exclusives.

<<<TODO>>>Add more structured syntax notation.

4 About Ada

Some general thoughts about Ada.

4.1 Introduction

This language is not known enough yet, at least not to the majority of us, much to the detriment of many potential users for that matter. Compared to the fashionable languages, Ada is more portable, more readable, allows for higher abstraction levels and has features and functionalities unseen in other languages. Ada also allows a more comfortable experience in system programming[4] and proves itself light enough to be usable on low class 8 bit processors[5].

Ada is the name of the first programmer to ever exist in humanity. And this first programmer was a woman: Augusta Ada Byron King, Countess of Lovelace, born in 1815, daughter of Byron, the great poet, Charles Babbage's assistant, she wrote programs destined to run on his famous machine.

Ada is an American military norm[6] as well as an international civil norm[7], it is the first object oriented language to be standardized at an international level. All Ada compilers must strictly adhere to the standard. There are hundreds of compilers destined to run on that many platforms but all of them will produce a code that runs identically.

Ada is used everywhere security is critical: Airbus [A3xx civil series and A400 military], Alsthom [High speed train], Boeing [777 and 787], EADS [Eurofighter, Ariane, ATV, many spaces probes], STS [line 14 Meteor], NASA [Electric power supply

[4] Thanks to it's representation clauses that obliterates the need to use bit masking for XORed for bit manipulation. This functionality *essential to system programming* is simply not there in C or even in Assembly language.

[5] Components that have at their disposal a couple dozen bytes of RAM and a couple Kilobytes of programming memory.

[6] MIL-STD-1815

[7] ISO/IEC 8652

of the International Space Station). The list goes on and on. Everywhere reliability and security must come first, Ada is the language of choice.

4.2 Why use Ada

Ada was created because software engineering is a human activity. Humans make mistakes, the Ada compiler is friend to developers. Ada is also friend to project managers for large scale development. An Ada application is written, expanded and maintained very naturally. For these reasons, Ada is also friend to executives. Ada is the language of happy programmers, managers and users.

Because Ada is a comfortable language by it's expressiveness and a restful language by it's reliability, humans involved with Ada also reflect the image of their language. The Ada community is a very comfortable community to visit and most meetings are very enlightening. Free libraries are numerous and are usually of a very high quality. Finally, the Ada community is very highly active and by now growing again.

4.3 The ending word

When Boeing decided, two decades ago, that all software for the 777[8] would be exclusively written in Ada, the corporate associates of the constructor made the remark that they were using, for a long time, languages such as C, C++ and assembly language and that they were fully satisfied with them. Boeing simply answered that only firms that could provide Ada software would be considered in contracts offerings. Therefore, the firms converted themselves to Ada.

Today, the development of software for the Boeing 777 nicknamed "The Ada Plane", has been performed and it is essentially thanks to the very big commercial success of this plane that Boeing was able to maintain the revenues created by its civil activities.

And what do the Boeing partner firms do from now on ? They continue to develop their new software in none other than... Ada, and here's why:
- They noticed that the length of time to convert developers to Ada is usually rather short. In a week, the developer is comfortable enough to write software in Ada and in less than a month, he feels totally comfortable with the language ;
- These firms did their accounting: written in Ada, software costs less, present less anomalies, are ready sooner and are easier to maintain.

[8] The Boeing 777 is the world's biggest two engines plane and the first civil Boeing having electrical flight commands, ten years later the Airbus A320.

Getting started

One can write neatly in any language, including C. One can write badly in any language, including Ada. But Ada is the only language where it is more tedious to write badly than neatly.
Jean-Pierre Rosen

1 Distribution

1.1 Directories

src	sources of HAC, plus the compatibility package HAT
exm	examples
exm-manual	sources related to this manual
test	testing

1.2 Key files

Key files are located in the main directory.

hac.pdf	this file
hac.gpr	project file for building HAC with GNAT
hac_work.xls	workbook containing, a bug list and a to-do list
save.adb	backup script, works both with HAC and GNAT!

2 Get a compiler

2.1 FSF GCC GNAT & AdaCore GNAT Community Edition compilers

Free Software Foundation GCC GNAT compiler can be used for closed sources applications. AdaCore compiler GNAT CE [Community Edition] can be used for GPL applications. For a HAC build point of view, it doesn't strictly matter.

However, for both Linux and Windows, it has to be said that today the AdaCore GNAT CE compiler is a better technical choice than the FSF compiler, for example for exception traces. This situation will change in the future.

2.2 Linux

On Debian-based Linuxes like Ubuntu, GCC GNAT FSF is part of the standard packages. You can get GNAT from AdaCore's web site: https://www.adacore.com/community which is today the better technical choice.

2.3 Windows

You can get GNAT from AdaCore's web site: https://www.adacore.com/community or from other sites, like http://www.getadanow.com providing links to FSF GCC GNAT compiler.

3 HAC build

3.1 Linux

❑ Debian 10 compiler

Install an Ada compiler. Depending of your system, replace aptitude by apt-get or apt, preceded or not by sudo:

```
user@system: aptitude install gnat-8 gprbuild git
```

❑ HAC Compilation

Assuming you wish to install HAC under /root, using it as a system tool:

Assuming you have the GNAT compiler installed, do the following from a command line interpreter. Open a terminal:

```
user@system: cd /root
user@system: git clone https://github.com/zertovitch/hac
user@system: cd hac
user@system: gprbuild -P hac
user@system: ln -s -f /root/hac/hac /usr/local/bin/hac

user@system: gnatmake -P hac
user@system: cd exm
user@system: ../hac gallery.adb
```

3.2 Windows

❑ Windows compiler

```
gnatmake -P hac
cd exm
..\hac gallery.adb
```

❑ Compilation

Assuming you have the GNAT compiler installed, do the following from a command line interpreter. Open a terminal:

```
user@system: gnatmake -P hac
user@system: cd exm
user@system: ..\hac gallery.adb
```

➪ Alternatively, with the help of a graphic file manager, you can go into the "exm" folder and double-click "e.cmd".

HAC language

Investment in C programs reliability will increase up to exceed the probable cost of errors or until someone insists on re-coding everything in Ada.
Gilb's laws synthesis

1 Design points

HAC reads Ada sources from any stream. In addition to files, it is able to read from a Zip archive (plan is to have sets with many sources like libraries in Zip files, for instance), from an internet stream, from an editor buffer (see the LEA project), from a source stored in memory, etc.

One goal is to complement or challenge GNAT, for instance in terms of compilation speed, or object code compactness, or usability on certain targets or for certain purposes (like scripting jobs).

HAC could theoretically be also used for tuning run-time performance; this would require compiling on other targets than p-code, that is, real machine code for various platforms.

2 Language subset

HAC supports a very small subset of the Ada language. On the other hand, Ada is very large, so even a small subset could already fit your needs.

There is a short (and growing) list of small programs that are working (in the meaning: the compilation succeeds and the execution gives a correct output). They are listed in the project file hac_exm.gpr in the "exm" directory.

You will see that the "Pascal subset" plus tasking is more or less supported, so you have things like subprograms (including nested and recursive subprograms), expressions, that are working, for instance.

3 Predefined types

HAC handles Integer, Real, Character, String, VString & Boolean as predefined types.

3.1 Integer

An HAC integer is always 64 bits wide, whatever the host system.

3.2 Real

18 digits accuracy floating-point number.

3.3 Character

8-bit character.

3.4 String

Literal like “Abc”. Its use is mostly limited to passing literals to HAT subprograms.

3.5 VString

Unlimited length. This type is available in the HAT package.

3.6 Boolean

True or False.

4 General capabilities

You can define your own data types: enumerations, records, arrays (and every combination of records and arrays).

Only constrained types are supported (unconstrained types are Ada-only types and not in the "Pascal subset" anyway). The Ada language has types depending on parameters whose value at compile-time or at run-time. A typical example is the predefined String type[9]. You can have s : String (1..10), a stack-allocated fixed string of length 10, but also s : String (1..n) where n is a function parameter or another dynamic value, or even s as a function parameter; the bounds are not explicit but can be queried.
But so far HAC doesn’t support unconstrained types, one of Ada most powerful constructs.

However, the String type is implemented, in a very limited way. So far you can only define variables (or types, record fields) with it, like:

[9] The definition of the Ada language minimizes the “magic items”, so String is also defined somewhere in Ada as: type String is array(Positive range <>) of Character

```
Digitz : constant String (1..10) := "0123456789";
```

... and output them with Put, Put_Line.

For general string manipulation, the most practical way with the current versions of HAC is to use the VString variable-length string type.

There are no pointers (access types) and nor heap allocation, but you will be surprised how far you can go without pointers!

Subprograms names cannot be overloaded, although some *predefined* subprograms (including operators) are overloaded many times, like "Put", "Get", "+", "&", ...

Tasks are implemented, but not yet tested.

A more systematic testing is done in the "test" directory.

5 **Key**words and special characters

5.1 Special characters

+ - * / ( ) [ ] , ; &

5.2 Reserved words

ABORT ABS ABSTRACT ACCEPT ACCESS ALIASED ALL AND ARRAY AT BEGIN BODY CASE CONSTANT DECLARE DELAY DELTA DIGITS DO ELSE ELSIF END ENTRY EXCEPTION EXIT FOR FUNCTION GENERIC GOTO IF IN INTERFACE IS LIMITED LOOP MOD NEW NOT NULL OF OR OTHERS OUT OVERRIDING PACKAGE PRAGMA PRIVATE PROCEDURE PROTECTED RAISE RANGE RECORD REM RENAMES REQUEUE RETURN REVERSE SELECT SEPARATE SOME SUBTYPE SYNCHRONIZED TAGGED TASK TERMINATE THEN TYPE UNTIL USE WHEN WHILE WITH XOR

6 Metrics

Object code size:
Stack size: 1 000 000 elements
Identifiers: 10000

HAC runtime

There are 10 types of people in the world: those who understand binary and those who don't.
Anonymous

1 Introduction

The HAC runtime is embedded in the HAC Virtual Machine. Standard types, arithmetic operators, etc. are immediately visible (see the Ada Standard package).

Additionally, another built-in package called HAT (HAC Ada Toolbox) is available and embedded in the HAC system.

The specification of HAT is located in the ./src/hat.ads file. Note that the file hat.ads is not needed for running your programs with HAC.

It is there for two purposes:

- Informational (you know what is available in the HAT package);
- compilation with a "full Ada" system; in that case the implementation files ./src/hat.adb and ./src/hat-non_standard.adb are needed as well.

The specification file lists all the types, functions and procedures available. HAT being embedded with HAC, you will often see HAC programs having at their very top:

```
with HAT;
```

2 Real functions

Floating-point numeric type functions.

2.1 Arctan

- Description

Performs arc-tangent operation with arguments in radians.

- Usage

function Arctan (F : Real) return Real

- Example

<<<TODO>>>

2.2 Cos

- Description

Performs cosine operation with arguments in radians.

- Usage

function Cos (F : Real) return Real

- Example

<<<TODO>>>

2.3 Exp

- Description

Performs exponential operation.

- Usage

function Exp (F : Real) return Real

- Example

<<<TODO>>>

2.4 Log

- Description

Performs natural logarithm operation.

- Usage

function Log (F : Real) return Real

- Example

<<<TODO>>>

2.5 Round

- Description

Performs rounding operation from Real to an Integer.

- Usage

function Round (F : Real) return Integer

- Example

<<<TODO>>>

2.6 Sin

- Description

Performs sine operation with arguments in radians.

- Usage

function Sin (F : Real) return Real

- Example

<<<TODO>>>

2.7 Sqrt

- Description

performs square root operation.

- Usage

function Sqrt (I : Integer) return Real
function Sqrt (F : Real) return Real

- Example

<<<TODO>>>

2.8 Truncate

- Description

Performs truncating operation from Real to an Integer.

- Usage

function Truncate (F : Real) return Integer

- Example

<<<TODO>>>

3 Real operators

3.1 **

- Description

Performs F1^F2 (power of) operation.

- Usage

function "**" (F1, F2 : Real) return Real

- **Not**e

Ada (and HAC) provides also a predefined function "**" (F1 : Real; F2 : **Integer**) return Real.

4 Randomize functions

4.1 Rand

- Description

Returns random Integer number in the real range [0, I+1[, truncated to lowest integer.

- Usage

function Rand (I : Integer) return Integer

- Example

Rand (10) returns equiprobable integer values between 0 and 10 (so, there are 11 possible values).

4.2 Rnd

- Description

Returns random Real number from 0 to 1, uniform.

- Usage

function Rnd return Real

- Example

See exm/random.adb, exm/einmaleins.adb

5 Characters functions

5.1 Pred

- Description

Returns previous character in ASCII table order.

- Usage

function Pred (C : Character) return Character

- Example

<<<TODO>>>

5.2 Succ

- Description

Returns next character in ASCII table order.

- Usage

function Succ (C : Character) return Character

- Example

<<<TODO>>>

6 VString functions

Variable-size string type

Null_VString : VString

6.1 Element

- Description

Return the Character in Index position of the VString argument.
Index starts at one.

- Usage

function Element (Source : VString; Index : Positive) return Character

- Example

<<<TODO>>>

6.2 Ends_With

- Description

Check if VString Item ends with another VString or String Pattern.

- Usage

function Ends_With (Item : VString; Pattern : String) return Boolean
function Ends_With (Item : VString; Pattern : VString) return Boolean

- Example

```
- Check VString with String pattern
if Ends_With (+"package",  "age") then
  Put_Line ("Match !");
end if;

- Check VString with VString pattern
if Ends_With (+"package",  +"age") then
  Put_Line ("Match !");
end if;
```

6.3 Head

- Description

Extract a VString between the beginning to Count Value to a VString.
Count starts at one.

- Usage

function Head (Source : VString; Count : Natural) return VString

- Example

```
Put_Line (Head (+"ABCDEFGH", 4));

"ABCD"
```

6.4 Index

- Description

Returns Natural start position of String or VString Pattern in the target Vstring Source, From a starting index.
Natural is zero if not found.
Natural starts at one.

- Usage

function Index (Source : VString; Pattern : String) return Natural
function Index (Source : VString; Pattern : VString) return Natural
function Index (Source : VString; Pattern : String; From : Natural) return Natural
function Index (Source : VString; Pattern : VString; From : Natural) return Natural

- Example

```
if Index (+"ABCDABCD", +"BC") = 2 then
    Put_Line ("Match !");
end if;

if Index (+"ABCDEFGH", +"BC", 4) = 6 then
    Put_Line ("Match !");
end if;
```

6.5 Index_backward

- Description

From the end of the target Vstring Source, returns Natural start position of String or VString Pattern in the target Vstring Source, From a backward starting index.
Natural is zero if not found.
Natural starts at one.

- Usage

function Index_Backward (Source : VString; Pattern : String) return Natural
function Index_Backward (Source : VString; Pattern : VString) return Natural
function Index_Backward (Source : VString; Pattern : String; From : Natural) return Natural
function Index_Backward (Source : VString; Pattern : VString; From : Natural) return Natural

- Example

```
if Index_Backward (+"abcdefabcdef", +"cd") = 9 then
    Put_Line ("Match !");
end if;

if Index_Backward (+"abcdefabcdef", +"cd", 8) = 3 then
    Put_Line ("Match !");
end if;
```

6.6 Length

- Description

Returns the length of the VString represented by Source.

- Usage

function Length (Source : VString) return Natural

- Example

```
Put (Length (+"ABCDEFGH"));

8
```

6.7 Slice

- Description

Returns a Vstring portion of the Vstring represented by Source delimited by From and To.
From and To start at one.

- Usage

function Slice (Source : VString; From : Positive; To : Natural) return VString

- Example

```
Put_Line (Slice (+"ABCDEFGH", 2, 4));

"BCDE"
```

6.8 Starts_With

- Description

Check if Vstring Item starts with another VString or String Pattern.

- Usage

function Starts_With (Item : VString; Pattern : String) return Boolean
function Starts_With (Item : VString; Pattern : VString) return Boolean

- Example

```
- Check VString with String pattern
if Ends_With (+"package",  "pac") then
  Put_Line ("Match !");
end if;

- Check VString with VString pattern
if Ends_With (+"package",  +"pac") then
  Put_Line ("Match !");
end if;
```

6.9 Tail

- Description

Extract a VString from Source between its end to backward Count Value.
Count starts at one (backward).

- Usage

function Tail (Source : VString; Count : Natural) return VString

- Example

```
Put_Line (Tail (+"ABCDEFGH", 4));

"EFGH"
```

6.10 Tail_After_Match

- Description

Extract a VString from Source starting from Pattern+1 position to the end.

- Usage

function Tail_After_Match (Source : VString ; Pattern : VString) return VString

- Examples

```
Put_Line (Tail_After_Match (Path, '/'));
"gnx-startup"

Put_Line (Tail_After_Match (Path, "ix"));
"/gnx-startup"

Put_Line (Tail_After_Match (Path, "gene"));
```

```
"six/gnx-startup"

Put_Line (Tail_After_Match (Path, "etc/genesix/gnx-startu"));
"p"

Put_Line (Tail_After_Match (Path, "/etc/genesix/gnx-startu"));
"p"

Put_Line (Tail_After_Match (Path, "/etc/genesix/gnx-startup"));
empty string

Put_Line (Tail_After_Match (Path, +"/etc/genesix/gnx-startupp"));
empty string

Put_Line (Tail_After_Match (Path, +"/etc/geneseven"));
empty string
```

6.11 To_Lower

- Description

Convert a Character or a VString to lower case.

- Usage

function To_Lower (Item : Character) return Character
function To_Lower (Item : VString) return VString

- Example

<<<TODO>>>

6.12 To_Upper

- Description

Convert a Character or a VString to upper case.

- Usage

function To_Upper (Item : Character) return Character
function To_Upper (Item : VString) return VString

- Example

<<<TODO>>>

6.13 Trim_Left

- Description

Returns a trimmed leading spaces VString of VString Source.

- Usage

function Trim_Left (Source : VString) return VString

- Example

```
Put_Line (Trim_Left (+"   ABCD   "));
"ABCD   "
```

6.14 Trim_Right

- Description

Returns a trimmed trailing spaces VString of VString Source.

- Usage

function Trim_Right (Source : VString) return VString

- Example

```
Put_Line (Trim_Right (+"   ABCD   "));
"   ABCD"
```

6.15 Trim_Both

- Description

Returns an all trimmed spaces VString of VString Source.

- Usage

function Trim_Both (Source : VString) return VString

- Example

```
Put_Line (Trim_Right (+"   AB CD   "));
"AB CD"
```

7 VString operators

7.1 +

- Description

Cast a String to a VString.

- Usage

function "+" (S : String) return VString

7.2 *

- Description

Duplicate a Character, String or VString Num times to a VString.

- Usage

function "*" (Num : Natural; Pattern : Character) return VString
function "*" (Num : Natural; Pattern : String) return VString
function "*" (Num : Natural; Pattern : VString) return VString

- Example

```
Put_Line (3 * "0");

"000"

Put_Line (3 * +"12");

"121212"
```

7.3 &

- Description

Concatenate a VString with a VString, String, Character, Integer and Real to a VString

- Usage

function "&" (V1, V2 : VString) return VString

function "&" (V : VString; S : String) return VString
function "&" (S : String; V : VString) return VString

function "&" (V : VString; C : Character) return VString
function "&" (C : Character; V : VString) return VString

function "&" (I : Integer; V : VString) return VString
function "&" (V : VString; I : Integer) return VString

function "&" (R : Real; V : VString) return VString
function "&" (V : VString; R : Real) return VString

7.4 =

- Description

Test equality between a VString and another VString or String.

- Usage

```
function "="  (Left, Right : VString) return Boolean
function "="  (Left : VString;  Right : String) return Boolean
```

- Example

<<<TODO>>>

7.5 <

- Description

<<<TODO>>>

- Usage

```
function "<"  (Left, Right : VString) return Boolean
function "<"  (Left : VString;  Right : String) return Boolean
```

- Example

<<<TODO>>>

7.6 <=

- Description

<<<TODO>>>

- Usage

```
function "<=" (Left, Right : VString) return Boolean
function "<=" (Left : VString;  Right : String) return Boolean
```

- Example

<<<TODO>>>

7.7 >

- Description

<<<TODO>>>

- Usage

```
function ">"  (Left, Right : VString) return Boolean
function ">"  (Left : VString;  Right : String) return Boolean
```

- Example

<<<TODO>>>

7.8 >=

- Description
- Usage

function ">=" (Left, Right : VString) return Boolean
function ">=" (Left : VString; Right : String) return Boolean

- Example

<<<TODO>>>

8 Type conversion functions

8.1 Chr

- Description

Convert an Integer to a Character.

- Usage

function Chr (I : Integer) return Character

- Example

<<<TODO>>>

8.2 Float_Value

- Description

Convert a VString to a Real.

- Usage

function Float_Value (V : VString) return Real

- Example

<<<TODO>>>

8.3 Image

- Description

Image returns a VString representation of Integer, Real, Ada.Calendar.Time & Duration.

- Usage

function Image (I : Integer) return VString
function Image (F : Real) return VString
function Image (T : Ada.Calendar.Time) return VString
function Image (D : Duration) return VString

- Example

<<<TODO>>>

8.4 Image_Attribute

- Description

Image_Attribute returns an Real VString image "as is" of F, instead of Image (F : Real) which returns a "nice" VString image of F

- Usage

Image_Attribute (F : Real) returns VString

- Example

```
Put_Line (Image (Real (4.56789e10)));

45678900000.0

Put_Line (Image_Attribute (4.56789e10));

4.56789000000000E+10
```

8.5 Integer_Value

- Description

Convert a VString to an integer.

- Usage

function Integer_Value (V : VString) return Integer

- Example

<<<TODO>>>

8.6 Ord

- Description

Convert a Character to an Integer

gdemont@hotmail.com
https://github.com/zertovitch/hac

- Usage

function Ord (C : Character) return Integer

- Example

<<<TODO>>>

8.7 To_VString

- Description

Convert a Char or a String type into VString type.

- Usage

function To_VString (C : Char) return VString
function To_VString (S : String) return VString

- Example

```
Input : String := "ABC";
Result : VString;
Result := To_VString (Input);
```

9 Console i/o functions

HAC comes with a real console/terminal input where several inputs can be made on the same line, followed by a "Return". It behaves like for a file. Actually it *could* be a file, if run like this:

```
user@system: prog <input.txt
```

9.1 End_Of_File

- Description

Return True if the end of file is reached.

- Usage

function End_Of_File return Boolean

- Example

```
Open (File_Tmp_Handle, +"./toto");
while not End_Of_File (File_Tmp_Handle) loop
  Get_Line (File_Tmp_Handle, Line_Buffer);
```

```
end loop;

Close (File_Tmp_Handle);
```

9.2 End_Of_Line

- Description

Return True if the end of line is reached.

- Usage

function End_Of_Line return Boolean

- Example

<<<TODO>>>

9.3 Get_Needs_Skip_Line

- Description

This function tells how the standard inputs occurs. If the standard input is the console, the return value is always True. It is what happens when the HAC program is compiled with a “full Ada” system.

With the HAC Virtual Machine, it can be different. It can be instantiated with the regular console I/O, but also with different kinds of I/O, typically, for inputs, input boxes, as in the LEA editor, but also in a possible Web interface.

In the latter case, the “Return” key press to conclude an imaginary input line after one or more Get’s is superfluous since each data entered is anyway validated by the user, through an “OK” button or a “Return” key press.

- Usage

function Get_Needs_Skip_Line return Boolean is (True)

- Example

Thanks to Get_Needs_Skip_Line, the exm/console_io.adb example behaves differently depending on the implementation media: command line for “hac”, or a user interface like LEA.

9.4 Get

- Description

- Usage

procedure Get (C : out Character) renames Ada.Text_IO.Get
procedure Get (I : out Integer)

procedure Get (F : out Real)
procedure Get (S : out String)

- Example

exm/console_io.adb

9.5 Get_Immediate

- Description

- Usage

procedure Get_Immediate (C : out Character)

- Example

```
procedure Pause is

Dummy : Character;

begin

  Put_Line (+"Press any key to continue...");
  Get_Immediate(Dummy);

end Pause;
```

9.6 Get_Line

- Description

Get and then move file pointer to next line (Skip_Line)

- Usage

procedure Get_Line (C : out Character)
procedure Get_Line (I : out Integer)
procedure Get_Line (F : out Real)
procedure Get_Line (V : out VString)

- Example

<<<TODO>>>

9.7 New_Line

- Description

Create a new blank line, or more than one when Spacing is passed.

- Usage

procedure New_Line (Spacing : Positive)

gdemont@hotmail.com
https://github.com/zertovitch/hac

- Example

<<<TODO>>>

9.8 Put

- Description

<<<TODO>>>

- Usage

```
procedure  Put (C : Character)
procedure  Put (I : Integer;
                Width : Ada.Text_IO.Field := IIO.Default_Width;
                Base : Ada.Text_IO.Number_Base := IIO.Default_Base);
procedure  Put (F : Real;
                Fore  : Integer := RIO.Default_Fore;
                Aft : Integer := RIO.Default_Aft;
                Expo : Integer := RIO.Default_Exp);
procedure  Put (B : Boolean;
                Width : Ada.Text_IO.Field := BIO.Default_Width);
procedure  Put (S :  String);
procedure  Put (V :  VString);
```

- Example

<<<TODO>>>

9.9 Put_Line

- Description

Put and then New_Line.

- Usage

```
procedure Put_Line (C : Character);
procedure Put_Line (I : Integer; Width : Ada.Text_IO.Field := IIO.Default_Width;
                          Base : Ada.Text_IO.Number_Base := IIO.Default_Base);
procedure Put_Line (F : Real; Fore : Integer := RIO.Default_Fore;
                 Aft : Integer := RIO.Default_Aft; Expo : Integer := RIO.Default_Exp);
procedure Put_Line (B : Boolean; Width : Ada.Text_IO.Field := BIO.Default_Width);
procedure Put_Line (S : String);
procedure Put_Line (V : VString);
```

- Example

<<<TODO>>>

9.10 Skip_Line

- Description

Clear the current line and gets ready to read the line after it, or skip more when Spacing is passed.

- Usage

procedure Skip_Line (File : File_Type; Spacing : Positive)

- Example

<<<TODO>>>

10 File i/o functions

subtype File_Type is Ada.Text_IO.File_Type

10.1 Append

- Description

Append a file.
File mode is “Out” (write mode).

- Usage

procedure Append (File : in out File_Type; Name : String)
procedure Append (File : in out File_Type; Name : VString)

- Example

```
Append (File_Tmp_Handle, +"./toto");
while not End_Of_File (File_Tmp_Handle) loop
  Get_Line (File_Tmp_Handle, Line_Buffer);
end loop;
Close (File_Tmp_Handle);
```

10.2 Close

- Description

Close a file.

- Usage

procedure Close (File : in out File_Type)

- Example

```
Open (File_Tmp_Handle, +"./toto");
while not End_Of_File (File_Tmp_Handle) loop
  Get_Line (File_Tmp_Handle, Line_Buffer);
end loop;
Close (File_Tmp_Handle);
```

10.3 Create

- Description

Create a file.
File mode is "Out" (write mode).

- Usage

procedure Create (File : in out File_Type; Name : String)
procedure Create (File : in out File_Type; Name : VString)

- Example

```
Create (File_Tmp_Handle, +"./toto");
while not End_Of_File (File_Tmp_Handle) loop
  Get_Line (File_Tmp_Handle, Line_Buffer);
end loop;
Close (File_Tmp_Handle);
```

10.4 New_Line

- Description

Create a new blank line, or more than one when Spacing is passed.

- Usage

procedure New_Line (File : File_Type; Spacing : Positive)

- Example

<<<TODO>>>

10.5 End_Of_Line

- Description

Return true if end of line is reached.

- Usage

function End_Of_Line (File : File_Type) return Boolean
function End_Of_Line (File : File_Type) return Boolean

- Example

<<<TODO>>>

10.6 End_Of_File

- Description

Return true if end of file is reached.

- Usage

function End_Of_File (File : File_Type) return Boolean
function End_Of_File (File : File_Type) return Boolean

- Example

<<<TODO>>>

10.7 Error handling

Input/output file errors are managed by HAC by means of an orderly exit :

```
HAC VM: raised Status_Error
File already open
Trace-back locations:
file_append.adb: File_Append.Nested_1.Nested_2 at line 12
file_append.adb: File_Append.Nested_1 at line 23
file_append.adb: File_Append at line 26
```

10.8 Get

- Description

Get the current line.

- Usage

procedure Get (File : File_Type; C : out Character)
procedure Get (File : File_Type; S : out String)
procedure Get (File : File_Type; I : out Integer)
procedure Get (File : File_Type; F : out Real)

- Example

```
Create (File_Tmp_Handle, +"./toto");

while not End_Of_File (File_Tmp_Handle) loop

  Get (File_Tmp_Handle, Line_Buffer);
  Skip_Line;

end loop;

Close (File_Tmp_Handle);
```

10.9 Get_Line

- Description

Get the current line and then move file pointer to the next line.

- Usage

procedure Get_Line (File : File_Type; C : out Character)
procedure Get_Line (File : File_Type; I : out Integer)
procedure Get_Line (File : File_Type; F : out Real)
procedure Get_Line (File : File_Type; V : out VString)

- Example

```
Create (File_Tmp_Handle, +"./toto");

while not End_Of_File (File_Tmp_Handle) loop

  Get_Line (File_Tmp_Handle, Line_Buffer);

end loop;

Close (File_Tmp_Handle);
```

10.10 Open

- Description

Open a file.
File mode is "In" (read mode).

<<<TODO>>> Comment File_Type

- Usage

procedure Open (File : in out File_Type; Name : String)
procedure Open (File : in out File_Type; Name : VString)

gdemont@hotmail.com
https://github.com/zertovitch/hac

- Example

```
Open (File_Tmp_Handle, +"./toto");
while not End_Of_File (File_Tmp_Handle) loop
  Get_Line (File_Tmp_Handle, Line_Buffer);
end loop;
Close (File_Tmp_Handle);
```

10.11 Put

- Description

<<<TODO>>>

- Usage

```
procedure Put (File : File_Type; C : Character)
procedure Put (File : File_Type
               I : Integer;
               Width : Ada.Text_IO.Field := IIO.Default_Width;
               Base : Ada.Text_IO.Number_Base := IIO.Default_Base)
procedure Put (File : File_Type;
               F : Real;
               Fore : Integer := RIO.Default_Fore;
               Aft : Integer := RIO.Default_Aft;
               Expo : Integer := RIO.Default_Exp)
procedure Put (File  : File_Type; B : Boolean; Width : Ada.Text_IO.Field := BIO.De-
fault_Width)
procedure Put (File : File_Type; S : String)
procedure Put (File : File_Type; V : VString)
```

- Example

<<<TODO>>>

10.12 Put_Line

- Description

Put and then New_Line (for S: it is the same as Ada.Text_IO.Put_Line)

- Usage

```
procedure  Put_Line (File : File_Type; C : Character)
procedure  Put_Line (File  : File_Type;
                     I     : Integer;
                     Width : Ada.Text_IO.Field       := IIO.Default_Width;
                     Base  : Ada.Text_IO.Number_Base := IIO.Default_Base)
procedure  Put_Line (File : File_Type;
```

```
                    F    : Real;
                    Fore : Integer := RIO.Default_Fore;
                    Aft  : Integer := RIO.Default_Aft;
                    Expo : Integer := RIO.Default_Exp);
procedure  Put_Line (File  : File_Type;
                    B     :  Boolean;
                    Width : Ada.Text_IO.Field := BIO.Default_Width)
procedure  Put_Line (File : File_Type; S : String)
procedure  Put_Line (File : File_Type; V : VString)
```

- Example

<<<TODO>>>

10.13 Skip_Line

- Description

Clear the current line and gets ready to read the line after it, or skip more when Spacing is passed.

- Usage

procedure Skip_Line (File : File_Type; Spacing : Positive)

- Example

```
Create (File_Tmp_Handle, +"./toto");

while not End_Of_File (File_Tmp_Handle) loop

  Get (File_Tmp_Handle, Line_Buffer);
  Skip_Line;

end loop;

Close (File_Tmp_Handle);
```

11 Time functions

subtype Time is Ada.Calendar.Time;

11.1 Clock

- Description

Returns current Time at the time it is called.

- Usage

function Clock return Time

gdemont@hotmail.com
https://github.com/zertovitch/hac

- Example

```
function Time_Stamp return VString is

Current_Time : constant Time := Clock;
Day_Secs, Day_Mins : Integer;

begin

  Day_Secs := Integer (Seconds (Current_Time));
  Day_Mins := Day_Secs / 60;

  return Image (Year (Current_Time)) &
         Time_Format (Month (Current_Time)) &
         Time_Format (Day (Current_Time)) &
         & "-" &
         Time_Format (Day_Mins / 60) &
         Time_Format (Day_Mins mod 60) &
         Time_Format (Day_Secs mod 60);

end Time_Stamp;
```

11.2 Day

- Description

Return Day from Date.

- Usage

function Day (Date : Time) return Integer

- Example

See Clock function above.

11.3 Month

- Description

Return Month from Date.

- Usage

function Month (Date : Time) return Integer

- Example

See Clock function above.

11.4 Seconds

- Description

Return Seconds from Date.

gdemont@hotmail.com
https://github.com/zertovitch/hac

- Usage

function Seconds (Date : Time) return Duration

- Example

See Clock function above.

11.5 Year

- Description

Returns year from Date.

- Usage

function Year (Date : Time) return Integer

- Example

See Clock function above.

12 Time operators

12.1 -

- Description

Return a Duration substracting Right from Left times.

- Usage

function "-" (Left : Time; Right : Time) return Duration

- Example

<<<TODO>>>

13 Semaphore functions

Specific stuff from SmallAda.

13.1 Signal

- Description

<<<TODO>>>

- Usage

procedure Signal (S : Semaphore)

- Example

<<<TODO>>>

13.2 Wait

- Description

<<<TODO>>>

- Usage

procedure Wait (S : Semaphore)

- Example

<<<TODO>>>

14 System

14.1 Argument

- Description

Returns Argument of index Number.

- Usage

function Argument (Number : Positive) return VString

- Example

See Argument_Count above.

14.2 Argument_Count

- Description

Count arguments passed to the current program.

- Usage

function Argument_Count return Natural

- Example

```
procedure Arguments is
begin
  Put_Line (Command_Name);
  Put_Line (Argument_Count);

  for A in 1 .. Argument_Count loop
    Put_Line ("  --> [" & Argument (A) & ']');
  end loop;
```


gdemont@hotmail.com
https://github.com/zertovitch/hac

```
end Arguments;
```

14.3 Command_Name

- Description

Returns full qualified current program name.

- Usage

function Command_Name return VString

- Example

```
user@system: hac gnx-instance

Put_Line (Command_Name);

/home/sr/Seafile/Sowebio/informatique/dev/ada/app/gnx/src/gnx-instance
```

14.4 Copy_File

- Description

Copy a file Source_Name to a file Target_Name

- Usage

procedure Copy_File (Source_Name : String; Target_Name : String)
procedure Copy_File (Source_Name : VString; Target_Name : String)
procedure Copy_File (Source_Name : String; Target_Name : VString)
procedure Copy_File (Source_Name : VString;Target_Name : VString)

- Example

<<<TODO>>>

14.5 Current_Directory

- Description

Returns current directory.

- Usage

function Current_Directory return VString

- Example

<<<TODO>>>

14.6 Delete_File

- Description

Delete a file Name.

- Usage

procedure Delete_File (Name : String)
procedure Delete_File (Name : VString)

- Example

<<<TODO>>>

14.7 Directory_Exists

- Description

Returns True if directory Name exists.

- Usage

function Directory_Exists (Name : String) return Boolean
function Directory_Exists (Name : VString) return Boolean

- Example

```
if Directory_Exists (HAC_Dir) then
   Put_Line ("HAC directory exists");
end if;
```

14.8 Directory_Separator

- Description

Returns system directory separator.

- Usage

function Directory_Separator return Character

- Example

With an Unix or Unix like system :

```
Put_Line (Directory_Separator);

/
```


gdemont@hotmail.com
https://github.com/zertovitch/hac

14.9 Exists

- Description

Returns True if file or directory Name exists.

- Usage

function Exists (Name : String) return Boolean
function Exists (Name : VString) return Boolean

- Example

```
if Exists (HAC_Dir & "/hac") then
   Put_Line ("HAC installation is done : )");
end if;
```

14.10 File_Exists

- Description

Returns True if file Name exists.

- Usage

function File_Exists (Name : String) return Boolean
function File_Exists (Name : VString) return Boolean

- Example

```
if File_Exists (HAC_Dir & "/hac") then
   Put_Line ("HAC interpreter is build : )");
end if;
```

14.11 Get_Env

- Description

Returns environment variable Name.

- Usage

function Get_Env (Name : String) return VString
function Get_Env (Name : VString) return VString

- Example

<<<TODO>>>

gdemont@hotmail.com
https://github.com/zertovitch/hac

14.12 Rename

- Description

Renames a file or a directory from Old_Name to New_Name.

Rename should be used with caution. Workaround could be using a shell command like “mv” or “ren”, depending of your OS. Linux example: Shell_Execute ("mv " & Old_Name & " " & New_Name);

- Usage

```
procedure Rename (Old_Name : String; New_Name : String)
procedure Rename (Old_Name : VString; New_Name : String)
procedure Rename (Old_Name : String; New_Name : VString)
procedure Rename (Old_Name : VString; New_Name : VString)
```

- Example

<<<TODO>>>

14.13 Set_Directory

- Description

Change to an existing directory Directory.

- Usage

```
procedure Set_Directory (Directory : String)
procedure Set_Directory (Directory : VString)
```

- Example

<<<TODO>>>

14.14 Set_Env

- Description

Set environment variable Name.

- Usage

```
procedure Set_Env (Name : String;  Value : String)
procedure Set_Env (Name : VString; Value : String)
procedure Set_Env (Name : String;  Value : VString)
procedure Set_Env (Name : VString; Value : VString)
```

- Example

<<<TODO>>>

14.15 Set_Exit_Status

- Description

Returns exit code to system.

- Usage

procedure Set_Exist_Status (Code : in Integer)

- Example

<<<TODO>>>

14.16 Shell_Execute

- Description

Executes Command returning the exit code Result from executed Command if needed.

- Usage

procedure Shell_Execute (Command : String)
procedure Shell_Execute (Command : VString)
procedure Shell_Execute (Command : String; Result : out Integer)
procedure Shell_Execute (Command : VString; Result : out Integer)

- Example

Without exit code handling :

```
Shell_Execute ("upx " & HAC_Dir & "/hac");
```

With exit code handling :

```
Shell_Execute ("ln -s -f " & HAC_Dir & "/hac /usr/local/bin/hac", Err);

if Err /= 0 then
  Put_Line (+"Error, symbolic link not created");
end if;
```

- Handling exit codes from Shell_Execute under Linux

Exit codes are shifted by some bits on the left, when the command is run on POSIX-compatible systems. The number of bits is 8 on Linux. So 0 becomes 256 and 1 becomes 0. This has been verified with the Ada procedure counterpart.

Under Linux, you must post process Err with this formula (see Issues at the bottom of this manual to further explanations):

```
Err := (Err / 256) mod 256;
```

Program example:

```
#!/usr/bin/env hac

with HAT; use HAT;

procedure test_Exit is

Err : integer;

begin

Shell_Execute ("exit 123", Err);

New_Line;
Put ("Raw value returned by Shell_Execute ('exit 123', Err); : ");
Put_Line (Trim_Left (Image( Err)));
New_Line;

if (Err = 123) then
Put_Line ("All good, Alles Gut, Tout est bon");
elsif (Err = 31488) then -- (123 * 256)

Put_Line ("We should do some math :)");

Err := (Err / 256) mod 256;

New_Line;
Put ("Corrected value returned: ");
Put_Line (Trim_Left (Image( Err)));

end if;

Set_Exit_Status (Err);

end test_Exit;
```

Program output:

```
# ./test-exit
Raw value returned by Shell_Execute ('exit 123', Err); : 31488

We should do some math :)

Corrected value returned: 123

sr@ro8    123 <- The correct exit code is returned…
```

HAC architecture

Doubling the number of programmers on a late project does not make anything else than double the delay.
Second Brook's Law

1 Introduction

<<<TODO>>> Short intro about compilers, interpreters, runtimes and executors, mainly some definitions to ease the reader.

2 Architecture

Let's do a quick breakdown of what happens when doing the "hac test.adb" command.

2.1 The builder

The builder is invoked: its role is to build an application in machine code corresponding to the Ada program test.adb. Side note: the machine in question doesn't exist for real, so it must be emulated at a later stage (imagine running an emulator for a computer like an old PC through DOS Box, or home computers of the 1980's, where the real machines have disappeared, but the games are still around). A machine that has never existed so far is called a Virtual Machine (VM).

The builder calls the compiler (via Compile_Main).

gdemont@hotmail.com
https://github.com/zertovitch/hac

2.2 The compiler

- The scanner

The compiler delegates the job of splitting the text code into symbols to the scanner. For instance an Ada keyword like “return” or “RETURN” is a symbol, “,” is another symbol, “1234” is another symbol.

- The parser

The job of analyzing the Ada code within the main procedure is also delegated to the parser.

The HAC parser is straightforward and, because of that, very fast. There is no code optimization, but the purpose of HAC is to minimize the time until your program is starting to run. There are several Ada compilers doing optimization, one of them is GNAT with amazing optimizations. HAC is not one of them!

- Modularity and packages

Before the main procedure, you have a few lines with the keyword “with”. That’s the modularity (connecting various pieces of code together). You can imagine those various pieces as books in a library (a common metaphor in computing).

Typically, HAC will make good use of the HAT package (HAT stands for HAC Ada Toolbox). So, when seeing the “with HAT” sequence, the compiler will call the librarian and ask for a book called “HAT”.

- The librarian

The librarian will make sure to handle the book and make its table of contents visible to the compiler. HAT is a built-in package, but you can have also a custom-programmed unit X (a package, a procedure, or a function).

Then, if the librarian doesn’t find the book labeled “X”, it needs to go shopping for it. Concretely, the librarian will call… the compiler (Compile_Unit), for compiling the source code of X.

Once compiled, the key information of X is in the library and the librarian will find the book each time X is asked for.

Once everything in the main procedure (and depending units) has been compiled, there is a heap of virtual machine code ready to be run.

2.3 The Virtual Machine interpreter

Enters the VM interpreter. It will execute the virtual machine code, instruction by instruction, like a real processor, on your machine. Actually, a future version of HAC could produce machine code for a real machine as well. Then, in that case, the interpreter would not be needed.

FAQ

The last bug isn't fixed until the last user is dead.
Sidney Markowitz

1 Return codes

HAC returns:
- 0 if the execution was completed without exception;
- 1 if an exception occurs during execution.

2 How to compile HAC programs with a “full Ada” compiler

2.1 HAC example

For many reasons, you may wish to compile a HAC program with a native-code Ada compiler like GNAT. For example to deeply speed execution time, even though HAC is a quite fast interpreter.

Create hac_to_gnat.adb:

```
with HAT; use HAT;

procedure HAC_To_GNAT is

   Start : Time := Clock;

begin

  New_Line;
  Put_Line ["Process 500M iterations - each * is 1M iterations"];
  New_Line;
```

```
  for I in 1 .. 500_000_000 loop
    if (I mod 1_000_000 = 0) then
      Put ("*");
    end if;
  end loop;

  New_Line;
  New_Line;
  Put_Line ("Process time:" & Image (Clock - Start) & "s");
  New_Line;

end HAC_To_GNAT;
```

Then execute it the HAC way:

```
user@system: hac hac_to_gnat.adb

Process 500M iterations - each * is 1M iterations

**********************************************************************************
**********************************************************************************
**********************************************************************************
**********************************************************************************
**********************************************************************************
**********************************************************************************
**************************

Process time: 672s
```

2.2 GNAT compilation

Then compile hac_to_gnat .adb:

```
user@system: gnatmake hac_to_gnat -I/etc/genesix/hac/src

gcc -c -I/etc/genesix/hac/src hac_to_gnat.adb
gcc -c -I./ -I/etc/genesix/hac/src -I- /etc/genesix/hac/src/hat.adb
gnatbind -I/etc/genesix/hac/src -x hac_to_gnat.ali
gnatlink hac_to_gnat.ali
```

And execute it:

```
user@system: ./hac_to_gnat

Process 500M iterations - each * is 1M iterations

**********************************************************************************
**********************************************************************************
**********************************************************************************
**********************************************************************************
**********************************************************************************
**********************************************************************************
**************************

Process time: 2s
```

These timings were obtained on a Intel(R) Xeon(R) CPU D-1521 @ 2.40GHz running HAC 0.091 and GNAT Community 2020 (20200429-93) under GNU/Linux Debian 10.

Seems HAC is slow, but HAC compiles to a Virtual Machine which is interpreted. HAC is not a native-code compiler. How HAC compares with common shell scripting languages like Bash ? The answer's below !

3 How fast is HAC ?

3.1 Timings compared to Bash

The same program written in Bash can't be executed because it failed to allocate 500M elements:

```
user@system: ./hac_to_bash

Process 500M iterations - each * is 1M iterations

./hac_to_bash: ligne 6: expansion des accolades : échec lors de l'allocation mémoire pour 500000000 éléments
./hac_to_bash: ligne 8: {1..500000000} %1000000 : erreur de syntaxe : opérande attendu (le symbole erroné est « {1..500000000} %1000000 »)
```

So at 50M iterations, Bash was *still unable to execute the script*, ending with a cryptic "Process stopped" message. Finally, at 5M elements, after 19 seconds (no kidding) used to allocate elements, execution begins:

```
user@system: time ./hac_to_bash

Process 5M iterations - each * is 10K iterations

**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************

real        0m42,542s
user        0m31,988s
sys         0m7,000s
```

Using this source:

```
echo ""
echo "Process 5M iterations - each * is 10K iterations"
echo ""

for i in {1..5000000}; do

  if [ $[[$i %10000]] == 0 ]; then
```

```
      echo -n "*"
   fi

done

echo ""
```

Reducing hac_to_gnat.adb to 5M elements to easily compare execution times:

```
user@system: hac hac_to_gnat_5M.adb

Process 5M iterations - each * is 10K iterations

**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************************************************************************
**************************

Process time: 6s
```

3.2 Conclusion

➪ This demonstrates HAC is not only an interpreter coming with *extensive capabilities and native strong typing*, but that *HAC it's seven times faster than Bash* too.

4 How to add a runtime procedure

<<<TODO>>> Explain and justify each steps.

https://github.com/zertovitch/hac/commit/56880520761 04a6ec639b680fedbced2dfa4ce20

5 How to add a runtime function

<<<TODO>>> Explain and justify each steps.

https://github.com/zertovitch/hac/commit/707ebcd1a2010d83c3217512705a4bd71bbf31cd

6 How to add bound checking to a runtime function

<<<TODO>>> Explain and justify each steps.

https://github.com/zertovitch/hac/commit/4b5154f117e146e413531aafd1b0b604f2ca878c

Programs examples

My Operating System is Emacs and Windows is its driver.
Anonymous

1 HAC firsts programs

Here are some very simple programs in HAC, in order to familiarize yourself with the syntax of Ada.

1.1 A salute to the world

Create the file hello.adb:

```
with HAT; use HAT;

procedure Hello is - Program´s name

begin

  Put_Line ["Salute to the World :)"]; -- Display the string and go to the next
line
  New_Line; -- Line feed

end Hello;
```

Run it:

```
user@system: hac hello

Salute to the World: )
```


gdemont@hotmail.com
https://github.com/zertovitch/hac

1.2 A simpler way to run HAC programs

To simplify things, at least in Unix based system, rename hello.adb without extension and make it executable:

```
user@system: mv hello.adb hello

user@system: chmod +x hello
```

Add a shebang at the very first line of the program:

```
#!/bin/env hac

with HAT; use HAT;

procedure Hello is - Program´s name

begin

  Put_Line ["Salute to the World :)"]; -- Display the string and go to the next
line
  New_Line; -- Line feed

end Hello;
```

Verify hello is now a standalone program:

```
user@system: hello

Salute to the World :)
```

1.3 A salute without use clause

You can omit the use clause but need then to prefix the procedures with the package name HAT. This is just for information and probably not the way to go for simple scripts :

```
#!/bin/env hac

with HAT;

procedure Hello is - Program´s name

begin

  HAT.Put_Line ["Salute to the World :)"]; -- Display the string and go to the
next line
  HAT.New_Line; -- Line feed

end Hello;
```

2 HAC examples

The program "gallery.adb" will run a bunch of demos that are located in the "exm" directory.
You can test HAC on any other example of course (the "*.adb" files in the "exm" and "test" directories).

As a bonus, you can build some examples with GNAT to compare the output. You can do it easily with the hac_exm.gpr project file. Since hac_exm.gpr is a text file, you can see there the progress (or the lack thereof) in the pieces of Ada code that are really working with HAC. See the "Limitations" section below as well.

❑ Files

HAC examples files are located in *~/hac/exm* directory :

```
user@system: find *.adb -type f | xargs ls -lS | awk -v OFS='\t' '{print $9, $5}' | sort -n

ackermann.adb                                635
anti_primes.adb              1297
arguments.adb                                295
bwt.adb                      4392
console_io.adb                               2174
covid_19_s.adb                               7874
days_1901.adb                                2441
doors.adb                    1329
echo.adb                     403
einmaleins.adb                               414
env.adb                      522
existence.adb                                558
file_append.adb              367
file_copy.adb                                857
file_read.adb                                310
fill_drive.adb                               546
gallery.adb                  1915
hello.adb                    568
mandelbrot.adb                               2014
maze_gen.adb                                 5345
merge_sort.adb                               2512
names_in_boxes.adb           1365
random.adb                   1160
shell.adb                    1081
shell_sort.adb                               2125
strings_demo.adb             6570
three_lakes_s.adb            7471
timing.adb                   873
unit_a.adb                   757
unit_b.adb                   460
unit_c.adb                   322
```

❑ Build

A project file *hac_exm.gpr* is available to compile all the programs :

```
user@system: cd ~/hac/exm/aoc
```

```
- Gprbuild without parameter get the first gpr file available.

user@system: gprbuild

using project file hac_exm.gpr
Compile
   [Ada]          ackermann.adb
.../...
   [Ada]          unit_c.adb
Bind
   [gprbind]      ackermann.bexch
.../...
   [Ada]          overloading.ali
Link
   [link]         ackermann.adb
.../...
   [link]         overloading.adb
```

3 HAC A**dvent** of Code

3.1 2020 year

❑ Links

Advent of code [https://adventofcode.com/2020]

https://adventofcode.com/2020/about

https://github.com/zertovitch/hac/tree/master/exm/aoc/2020

❑ Files

Hac Advent of Code files are located in *~/hac/exm/aoc/2020* directory :

```
user@system: find *.adb -type f | xargs ls -lS | awk -v OFS='\t' '{print $9,
$5}' | sort -n

aoc_2020_02.adb                2602
aoc_2020_03.adb                1849
aoc_2020_04.adb                4353
aoc_2020_04_b_full_ada.adb     2653
aoc_2020_05.adb                1241
aoc_2020_06.adb                2435
aoc_2020_06_full_ada.adb       1483
aoc_2020_06_full_ada_using_hat.adb             1496
aoc_2020_07.adb                9768
aoc_2020_07_full_ada.adb       5074
aoc_2020_07_full_ada_vectors_2x.adb            5027
aoc_2020_08.adb                2816
aoc_2020_09.adb                2625
aoc_2020_10.adb                3900
aoc_2020_11.adb                4583
aoc_2020_11_full_ada.adb       3900
aoc_2020_12.adb                2515
aoc_2020_13.adb                3107
aoc_2020_14_full_ada.adb       6033
aoc_2020_15.adb                2691
aoc_2020_15_full_ada.adb       2199
```

```
aoc_2020_15_full_ada_hashed_maps.adb           2709
aoc_2020_16.adb               6227
aoc_2020_17.adb               5330
aoc_2020_18_full_ada.adb      1373
aoc_2020_18_weird_formulas.adb                 35461
aoc_2020_19_full_ada.adb      5352
aoc_2020_20.adb               4080
aoc_2020_21_full_ada.adb      6977
aoc_2020_21_full_ada_preproc.adb               1539
aoc_2020_22.adb               6089
aoc_2020_22_full_ada.adb      6883
aoc_2020_23.adb               6255
aoc_2020_23_simple_array.adb  4569
aoc_2020_24.adb               6350
aoc_2020_25.adb               2195
```

- Build

A project file *aoc_2020.gpr* is available to compile all the programs :

```
user@system: cd ~/hac/exm/aoc/2020

- Gprbuild without parameter get the first gpr file available.

user@system: gprbuild

using project file aoc_2020.gpr
Setup
   [mkdir]        object directory for project AoC_2020
Compile
   [Ada]          aoc_2020_25.adb
.../...
   [Ada]          aoc_2020_18_weird_formulas.adb
Bind
   [gprbind]      aoc_2020_25.bexch
   .../...
   [Ada]          aoc_2020_02.ali
Link
   [link]         aoc_2020_25.adb
.../...
   [link]         aoc_2020_02.adb
```

Programming basics

Weinberg's Second Law : If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.
Gerald Weinberg

HAC is a scripting, prototyping and educational procedural Ada subset language. It is perfect to write small, medium and - why not ? - huge non object programs in a modular and structured way.

If your are not an experienced programmer mastering a method programming as a tool, you may find this chapter useful. Your creative spirit's the limit.

This chapter deals with top-down analysis and modular programming method. Understanding and assimilating the following will already make you a very good developer. This matter is not an end but a foundation to go further. Like understanding the differences between object programming by classification or by composition. And why, for many projects, object programming should be avoided and for others, it should really be adopted.

So, no object methods will be discussed. It's beyond the scope of this manual. Most developers using object-oriented languages have not learned any methods, using wrong tools with no thinking. We know the result. To be good at object-oriented development, you must already understand the basics of analysis and modular and structured programming. One step after the other :)

1 Tools

To create a program, you must:
- Master an analytical method;
- Know Boolean algebra;

- Use a programming language;
- Have a good general culture and know-how.

Of these four elements, the first one is the most difficult to acquire, but I hope that the following lines will help you in this field.

I could have added: paper, a pencil and an eraser, because these three objects are always the basis of a good program and you should not rush to code.
always the basis of a good program and that one should not rush to code.

You will notice that the knowledge of a language comes after the theory. This is normal. As analysis precedes writing, mastering design precedes mastering a language. Finally...

⇨ The joy of programming must remain the driving force of your motivation.

2 Analysis

2.1 Methods overview

The main classes of methods are:
- Modular and structured programming ;
- Object method by composition
- Object method by classification.

The modular and structured programming method is still used in many fields as the main programming method.

It is also used in object methods, at least in the following contexts:
- In the main startup and finalization module;
- In the functions (methods) of the objects.

Object method can be divided into object methods by composition or by classification.

The object method by classification (hierarchical) is the most known object method and yet the least relevant, except for developing a graphical interface or any other project clearly requiring the inheritance tool.

Object method programming is beyond the scope of this manual.

2.2 Top-Down example

The top-down analysis approach is one among many. It is intuitive and efficient. One can fly rockets with it but it is good that you know that other ways exist.

Everyone programs, the car mechanic, the postal worker and the cook. Didn't you know that? So let's start by cooking an egg!

Mastering an analysis method allows to *analyze a problem*, even a very complex one, and break it down by *successive refinements*, into a sum of problems, one by one so obvious to solve, that one stops the analysis by declaring it is finished!

So we're going to cook an egg, a hard-boiled egg to be precise. But could you detail such a seemingly simple process without hesitation? Let's see it together.

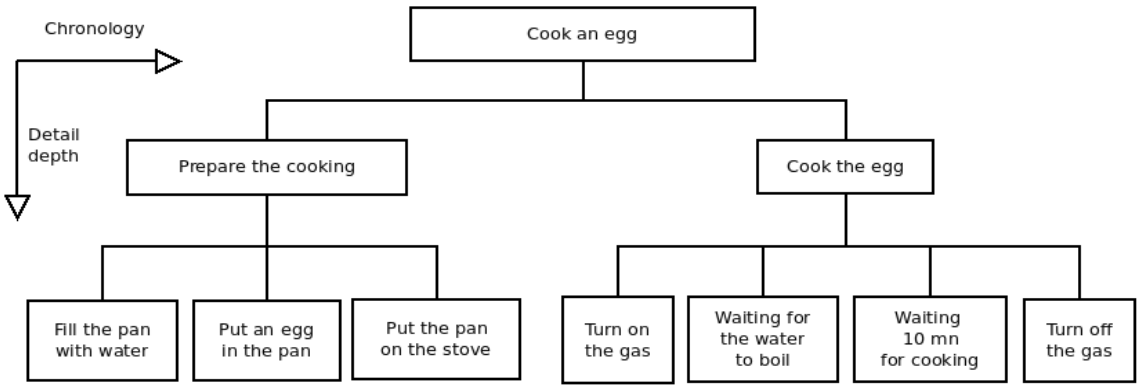
❑ Problem's decomposition

We could, for example, start by breaking down, by *refining*, the action of cooking an egg into two main steps two main steps: *preparation* and *cooking*.

Then we could take these two main steps and refine them again:
- The preparation is to fill the pan with water, put an egg in it and put the pan on the stove;
- Cooking is turning on the gas, waiting for the water to boil, wait 10 minutes for cooking[11] , then turn off the gas.

This approach is known as *decomposition by successive refinements*.

Once this decomposition is completed, it is essential to represent it visually, thanks to the *SPD*, the *Structured Program Diagram*, sometimes called the *JSP diagram*, after its inventor[12]:



This SPD works in two dimensions:
- In the vertical plane, we go down from the most complex to the simplest;
- In the horizontal plane, the direction of the reading represents naturally, chronologically,
the tasks to be performed.

The SPD has a dual purpose:
- In the first instance, it allows you to gain an *overview of the problem at hand* and ensure that your analysis is *consistent and complete* ;
- Secondly, since each box represents an action that is so simple to solve that it does not require further analysis, the DSP allows you to go directly to the second phase: *the pseudo-code*!

11 It's a lot, but not a problem, unless you like them soft. The shell will come off more easily.

12 It is difficult to determine the origin of these concepts. Many researchers worked on them at the same time. One of Jackson's merits was to promote the notion of initial read-current read in loop processing. https://en.wikipedia.org/wiki/Jackson_structured_programming

In creating this SPD, we have *modularized* our problem. We have *decomposed* our problem into a series of *elementary modules*. When writing the *pseudo-code*, we will describe the functioning of each *module* using *structures*. These *structures* form the basic building blocks of *structured* programming, without goto or spaghetti code.

- Pseudo-code

The pseudo-code is the computer translation, as structures, of the already written SPD.

The SPD and the pseudo-code are linked. They must be consistent with each other.

It is often while checking this consistency, at the time of writing the pseudo-code, that one realizes that the level of detail of the SPD is incorrect. If the level of detail is too high, the pseudo-code contains useless modules that do not contain any processing that deserves to be modularized. On the other hand, if the level of detail is not high enough, the pseudo-code contains modules that are far too big.

Before going into the details of the general writing of a pseudo-code, let's see a small example, with our hard-boiled egg, just to get a taste of it.

```
begin *** cook an egg ***

do *** prepare the cooking ***
do *** cook the egg ***

end *** cook an egg ***
```

In this first pseudo-code, representing the main module of the "cook an egg" program, the analogy between SPD and pseudo-code is clear.

The term *do* before *prepare the cooking* represents the call to the module *prepare the cooking*. Each module starts with *start *** module name **** and ends with *end *** module name ****.

Let's move on to writing prepare the cooking module:

```
begin *** prepare the cooking ****

 do while “pan is not filled”
  fill with water
 end do while

 do *** put the pan on the stove ***

 do *** put an egg in the pan ***

end *** prepare the cooking ***
```


gdemont@hotmail.com
https://github.com/zertovitch/hac

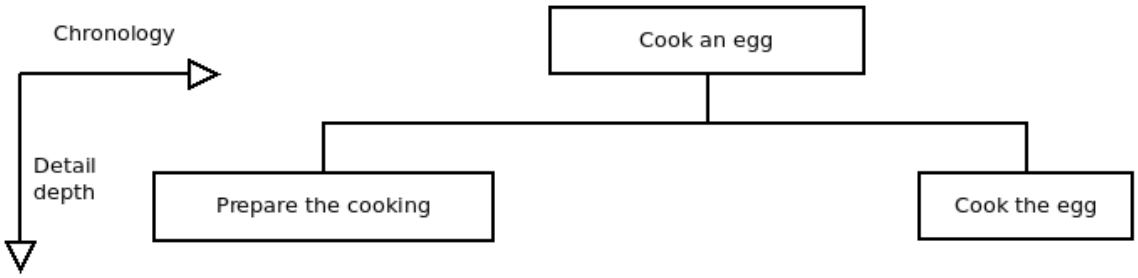
This is when a problem arises. The module putting the pan on the stove is a really very simple action. A so simple one that it does not, in fact, deserve to be isolated in a module. Leaving the analysis as it is, without changing anything, would result in making the program more complex than it deserves to be.

So we will simplify the pseudo-code:

```
begin *** prepare the cooking ****

 do while “pan is not filled”
  fill with water
 end do while

 put the pan on the stove

 put an egg in the pan

end *** prepare the cooking ***
```

So it appeared that the level of detail in the SPD was too high. The actions of the last rank: *pan on the fire*, *fill with water*, etc. did not deserve, by themselves, a separate module.

They should be grouped together in the modules of higher rank: *prepare the cooking* and *cook the egg*.



The analysis of *the cooking of the egg* ends with the pseudo-code of the last module:

```
begin *** cook the egg ***

 turn on the gas

 do while “water does not boil”
  wait
 end do while

 do while “not 10 minutes elapsed”
  wait
 end do while

 turn off the gas

end *** cook the egg ***
```


gdemont@hotmail.com
https://github.com/zertovitch/hac

After this example, we now take a closer look at this analysis method.

3 Modular and structured programming method

3.1 Introduction

This modular and structured programming approach is generic to dozens of methods invented in the 1980s to make software execution more reliable and improve maintenance.

These methods differed essentially in the symbols, vocabulary and aesthetics of the diagrams. They are still relevant today as the indispensable basis of the methods used by a good developer.

The method illustrated here is GMSP: General, Modular and Structured Programming[13]. It comes from the teaching provided by the french Control Data Institute, located in Paris, which has now disappeared, with the help of PLATO[14], a Computer Aided Learning sytem. Graphical extensions to these methods exist, for example SADT or its real-time extension SART.

The author does not really appreciate graphical representations (which make nice drawings for IT managers) in analysis methods. Flowcharts, flow diagrams, SADT or UML graphs generally bring more confusion than information.

However, some graphical representations, such as the SPD or the HOOD method diagrams, are good tools. They are the first steps of the written specifications, which can be found, strictly speaking, in the specifications of an Ada package.

3.2 Program's Structured Diagram

Writing a SPD, *Structured Program Diagram*, means identifying, decomposing and prioritizing functions in a coherent whole, in order to allow the writing of the program pseudo-code.

- Process detailed

The process of creating the SPD is an iterative one, which loops around itself, to identify all the tasks to be carried out, until the possibilities of refinement are exhausted, i.e. until the problem to be solved can no longer be detailed.

This approach is called a *top-down approach*, in order to show that we start from the global problem, at *the top of the diagram*, and work our way down to the smallest detail, *towards the bottom of the diagram*. Each time we add a level of detail, we create a new line.

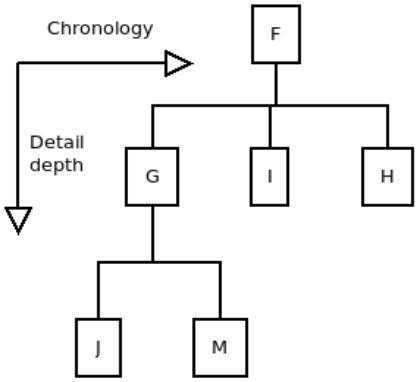
For each detail level, the identified tasks are written in the reading direction, in order of execution. They are placed in *boxes*. For clarity of the SPD, all boxes of a lower rank are connected by lines to the box of the higher rank.

[13] PGMS in french, as "Programmation Générale, Modulaire et Structurée"

[14] Programmed Logic for Automatic Teaching Operations - https://en.wikipedia.org/wiki/PLATO_[computer_system]

SPDs are always written and read:
- Top to bottom, for level of detail;
- From left to right, for chronological steps.

Example:



In no case does a SPD show the tests and other low-level actions that are the responsibility of programming.

A SPD is both the overview and the backbone of the analysis.

Writing the SPD is the most difficult part of the analysis.

3.3 Pseudo-code

The pseudo-code writing is done from the SPD. Each *box* of the SPD will correspond to a module in the *pseudo-code*.

➪ We repeat : one SPD box to one module in the pseudo-code.

The writing of a pseudo-code is done from elementary bricks, which we will examine now.

❑ Main module

A program *starts* and *ends* at the master module.

Here is the pseudo-code, also called PC, of the previous SPD, describing the master module of program F :

```
begin *** F ***

 do *** G ***

 do while P [while P is true]
  do *** I ***
 end do while

 do *** H ***
```


gdemont@hotmail.com
https://github.com/zertovitch/hac

```
end *** F ***
```

The beginning of a module is represented by begin *** module name *** and the end of a module is represented by end *** module name ***.

The name of the main module is the name of the program.

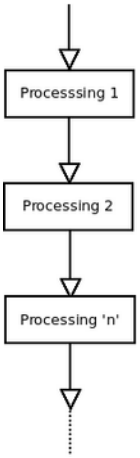
❏ Other modules

Other modules are written the same way. Here is the pseudo-code of module G of the previous SPD, describing the program G:

```
begin *** G ***

 do *** J ***

 if Q [if Q is true]
  do *** M ***
 end if

end *** G ***
```

❏ Sequence

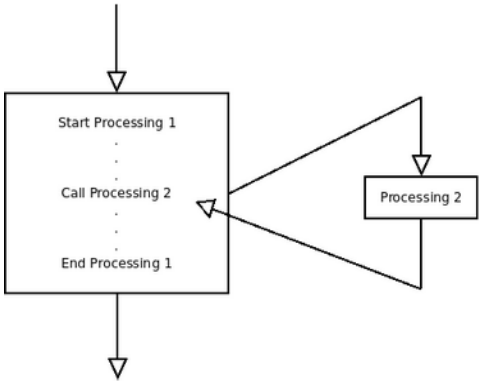
Sequence is the simplest form of pseudo-code. It just represents the sequence of several processes, which are executed one after the other:



❏ Module call

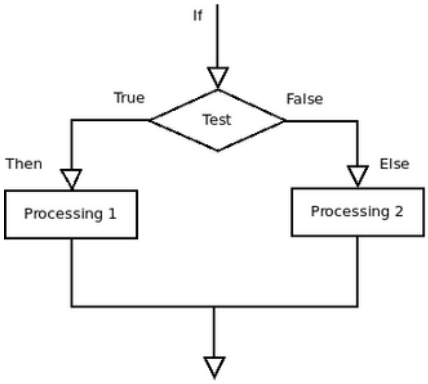
Module call is represented by *do *** module name ****. The processing of the calling module stops at the line of the call and the called module executes.

At the end of the called module, the latter returns to the calling module and the execution of the latter resumes at the line following the call which has just been executed:



❑ If… else… end if

The alternative is the simplest test of a pseudo-code. Depending on the truth of the test condition, the program flow is directed to one processing or another:



The alternative is represented in pseudo-code as follows:

```
if test condition [is true]
 Processing 1
else
 Processing 2
end if
```

❑ If… elsif… else… endif

This structure is an extension of the alternative:

```
if test condition 1
 Processing 1

elsif test condition 2
 Processing 2

elsif test condition 3
 Processing 3

else
 Default proessing
end if
```


gdemont@hotmail.com
https://github.com/zertovitch/hac

The default processing is executed when no test condition has been checked.

This structure is equivalent to a nesting of alternatives. But these nestings are much less readable, as shown in the example below:

```
if test condition 1
 Processing 1
else
 if test condition 2
  Processing 2
 else
  if test condition 3
   Processing 3
  else
   Default processing
  end if
 end if
end if
```

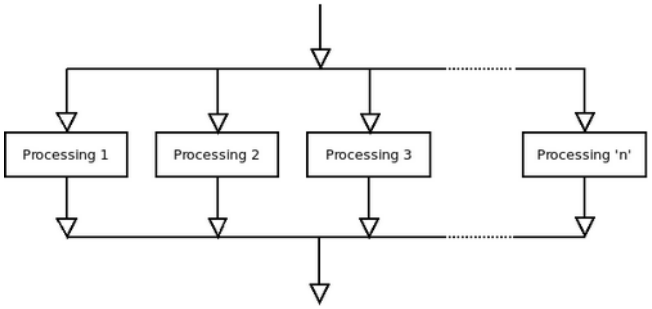
- Case... when... else... end case

The selection is a different form of the alternative because the test is no longer Boolean (true or false) but depends on the content of the tested value. A pseudo-code is more meaningful:

```
selection value to test

 when value 1
  Processing 1

 when value 2
  Processing 2

 when value 3
  Processing 3

 when others
  Default processing

end selection
```

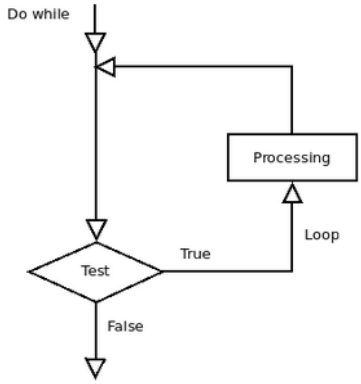
This structure can be represented as follows:

gdemont@hotmail.com
https://github.com/zertovitch/hac



❑ Do while… end do

This loop structure is useful when you want the program flow *to avoid processing in the loop if the condition is false* at the *first pass in the loop*:

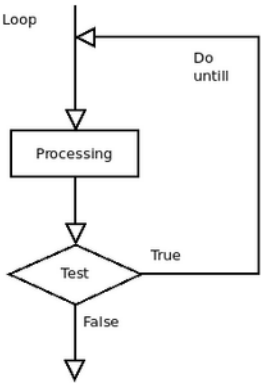


The pseudo code of such a structure is as follows:

```
do while test (is true)
 process
end do while
```

❑ Loop… until

This loop structure differs from the previous one because the processing in the loop is done once before the loop condition is tested. Thus, one will always pass at least once in this type of loop:



Here is the notation of the loop… until in pseudo-code:

```
loop
 process
until condition test [is true]
```

It is clear that the test is performed after a first pass in the loop.

3.4 Functions

➪ One point of entry, one point of exit. No anticipated exit. Never. We repeat: never :)

All parameters will be named and, if the language - such as Ada - allows it, the parameter names will be used in the function calls.

4 Boole algebra

Here is a practical summary about Boolean algebra, which should be known by all developers.

4.1 Identities, properties and De Morgan's laws

Two conventions are used:
– ≡ for equivalence. A ≡ B means that A and B are two equivalent conditions and that they are interchangeable;
– NOT A for the negation of A. If A is true, NOT A is false.

❑ Identities

```
A OR 0 ≡ A                    NOT [NOT A] ≡ 1
A OR 1 ≡ A                    A OR A ≡ A
A OR [NON A] ≡ 1              A AND A ≡ A
A AND [NON A] ≡ 0
```

❑ Properties

```
A AND B ≡ B AND A
A OR B ≡ B OR A
A AND [B AND C] ≡ [A AND B] AND C
A OR [B OR C] ≡ [A OR B] OR C
A AND [B OR C] ≡ [A OR B] ET [A OR C]
A AND [B OR C] ≡ [A AND B] ET [A AND C]
A OR [B AND C] ≡ [A OR B] ET [A OR C]
```

❑ De Morgan's law

```
NOT [A OR B] ≡ [NOT A] AND [NOT B]
NOT [A AND B] ≡ [NOT A] OR [NOT B]
```

4.2 Practical advises

In your current language manual, you will certainly find the description of priorities in the evaluation of logical expressions.

The following is an example of evaluation priorities:

1. Expressions located in the innermost brackets ;
2. Negation;
3. AND and OR (In the Ada language, these two operators are on an equal footing, which is not the general rule in other languages where AND usually has a higher priority than OR);
4. With equal priority, evaluate expressions from left to right.

⇨ One might be tempted to take these priorities into account to write the shortest possible test condition, but *this should be avoided at all costs* for reasons of clarity.

Here are three basic rules *to follow in all circumstances*:

1. Never hesitate to *use parentheses* to increase readability and reliability.
2. To work on or reverse a complex condition, you must *first restore the implicit parentheses*.
3. A simplification of a complex condition is *done by applying the De Morgan's laws*.

Notes

With the Wildebeest and the Penguin, there's no Bull.
Number Six

1 To-do list

1.1 HAC

The to-do list is located in the spreadsheet "To do", within the workbook "hac_work.xls".

1.2 Doc

Hunt **<<<TODO>>>** tags :]

2 Issues

2.1 Github

Issues are listed on Github: https://github.com/zertovitch/hac/issues

Ada, “it’s stronger than you”.

Tribute to Daniel Feneuille, legendary french Ada teacher

In Strong Typing We Trust !

gdemont@hotmail.com
https://github.com/zertovitch/hac

https://this-page-intentionally-left-blank.org

gdemont@hotmail.com
https://github.com/zertovitch/hac