The Mentor-V5 documentation

B. Melese, V. Migot, D. Verove

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 43

THE MENTOR - V5 DOCUMENTATION

Bertrand MÉLÈSE
Valérie MIGOT
Denis VEROVE

Janvier 1985

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tel.(3) 954 90 20

La documentation du système Mentor-V5

Ce rapport contient la plupart des fichiers de documentation du système Mentor-V5. La documentation complète de Mentor-V5 est incluse dans la bande de distribution de ce système. Mentor-V5 existe sur Vax/Unix, Multics et SM90/SMX. D'autres transports sont prévus (Vax/VMS, Ridge) ou en cours (Data General MV10000). Dans ce rapport, nous ne décrivons ni la philosophie du système ni ses futurs developpements. Pour les connaitre, le lecteur se reportera aux publications sur Mentor. Ce rapport est une version préliminaire. Toutes les remarques visant à l'améliorer seront les bien venues.

The Mentor-V5 Documentation

This report contains large parts of the information files of the Mentor-V5 system. The complete documentation is included in the distribution tape. Mentor-V5 exists on Vax/Unix, Multics and SM90/SMX. Other transports are planned (Vax/VMX, Ridge) or already started (Data general MV10000). In this report, no attempt is made to describe the philosophy of the system, its history or future plans. These can be found in publications about the Mentor system. This report is intended to be a preliminary version. All remarks from readers and users are welcome, whether suggestions for improvement or corrections of errors in the text.

Contents

This report is made of several information files from the directory "info" of the Mentor-V5 distribution tape. The name of the corresponding file is indicated, between parentheses, after its title.

1- The Mentor-V5 documentation (README)..... 4

2- New features in the version 5 of Mentor and main differences between version 5 and previous versions (Mentor.Ver5.i)..... 8

3- The Mentor program manipulation system. General information and Mentol user's manual (mentor.gi.i)..... 13

4- A brief description of the annotation mechanism in Mentor-V5 (annotation.i)..... 46

5- The Multi-language facility in Mentor-V5 (multi-lang.i)..... 55

6- How to use the menu-driven input mode (beginners.i)..... 61

7- How to Write Mentor procedures (mentor.wr_pr.i)..... 64

8- Language independent commands of Mentor (mentor.procs.i)..... 74

9- The full screen environment of Mentor (mentor.video.i)..... 91

10- The Pascal Environment (pascal.procs.i)..... 95

11- The Rapport Environment (rap.procs.i)..... 112

12- The Ada Environment (ada.procs.i)..... 124

13- How to introduce a user defined language inside Mentor-V5 (mentorkit.i)..... 128

14- The Metal Environment (metal.procs.i)..... 132

15- Known traps in using Metal and new features of Metal in Mentor-V5 (metal.traps.i)..... 139

16- How to write an unparser for a new language defined in Metal (mentor.dec.i)..... 149

17- How to use the Mentor interface (interface.i and TREEFRONT.p)..... 157

18- Dials and Toggles (togg-dials.i)..... 161

19- Mentor interrupt facility (mentor.break.i)..... 183

Bibliography

1- Publications in English

V. Donzeau-Gouge, G. Kahn, G. Huet, B. Lang, J.J. Levy,
"A structure oriented program editor: a first step toward computer assisted programming", International Computing Symposium, North Holland Publishing Co. (1975)

V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse, E. Morcos,
"Outline of a tool for document manipulation", IFIP, septembre 1983, Paris

V. Donzeau-Gouge, B. Lang, B. Mélèse,
"Practical Applications of a Syntax Directed Program Manipulation Environment", Proceedings of the 7th Int. Conf. on Soft. Eng., Orlando, Florida, March 1984

V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse,
"Documents Structure and Modularity in Mentor", ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments, Pittsburgh, April 1984. SIGPLAN Notices, Vol 19, No 5, May 1984.

G. Kahn, B. Lang, B. Mélèse, E. Morcos,
"Metal: a formalim to specify formalisms", Science of Computer Programming, North Holland, Vol. 3, No. 2, 151-188, August 1983 (Referenced in this report under the name "metal.i")

B. Mélèse,
"Structured Editing - Unstructured Editing, Cooperation and complementarity", Proceedings of the 2nd Software engineering conference, Nice (1984). Edited by E. Girard, North Oxford Academic, Oxford, England

2- Publications in French

B. Mélèse,
"Mentor: l'environnement Pascal", Rapport Technique No. 5, INRIA, Octobre 1981

B. Mélèse,
"Métal, un langage de spécification pour le système Mentor"
Technique et Science Informatique (AFCET), Vol. 1 No 4, Juillet-Aout 1982

B. Mélèse,
"Mentor Rapport: Manipulation de textes structurés sous Mentor"
Rapport Technique No. 23, INRIA, Avril 1983 (Referenced in this report under the name "Rapport_gi.i")

B. Mélése,
"Edition structurée - Edition non structurée, Coopération et complémentarité", 2ième Colloque de Génie logiciel, Nice, Juin 1984, (AFCET)

B. Mélése,
"Manipulation de programmes Pascal au niveau des concepts du langage"
Thèse de 3ième cycle, Université Paris 11, Orsay, 1980

V. Migot,
"Un Pascal Modulaire sous Mentor",
Thèse de 3ième cycle, Université Paris 11, Orsay, 1983

T. Despeyroux,
"Introduction de spécifications sémantiques dans Mentor",
Thèse de 3ième cycle, Université Paris 11, Orsay, 1983

D. Verove,
"Mentor-LTR: Un système de manipulation de programmes LTR-V3",
Rapport technique interne DRET-SEMA, Mars 1983

This is the Version 5 of MENTOR called MENTOR-V5.
See the README file in the mentor directory.

****
********** WARNING **********
****

Mentor-V5 is slightly incompatible with previous versions of Mentor:

This concerns writing of unparsers and of Pascal programs using the interface TREEFRONT.p. If you are such a user, you MUST read the files Mentor.Ver5.i and metal.traps.i before any attempt to user Mentor-V5.

****
* ALL UNPARSERS AND PROGRAMS WRITTEN WITH THE INTERFACE *
* TREEFRONT.P HAVE TO BE SLIGHTLY MODIFIED *
****

README

The Mentor-V5 documentation

Bertrand Mélèse
(July 84 updated November 84)

*** WARNING *** (For users that have already defined a language under Mentor or wrote pascal programs linked to Mentor using the interface TREEFRONT.p)

Mentor-V5 is slightly incompatible with previous versions of Mentor: This concerns writing of unparsers and of Pascal programs using the interface TREEFRONT.p. If you are such a user, you MUST read Mentor.Ver5.i and metal.traps.i before any attempt to user Mentor-V5.

0- The pascal compiler used by Mentor-V5 is the Berkeley Pascal compiler slightly modified. Modifications made are recorded in the file PC.changes.i.

1- General Information:

Mentor-V5 makes extensive use of the character @. Then, this character cannot be used as kill line when working with Mentor-V5.

To get started with Mentor-V5 you should at least read the files

- mentor.gi.i (obtained on-line by the command .ginfo)
- mentor.video.i
- mentor.procs.i (obtained on-line by the commands .list and .sysdetails)
- mentor.break.i

See also the files

- mentor.demo.i.
- mentor.input.i

They contain samples of mentor sessions that can help you in getting the feeling of the system.

In Mentor-V5 THE new things are the annotation mechanism and the multi-language facility. To known about new features in Mentor-V5 see Mentor.Ver5.i, annotation.i and multi-lang.i.

YOU SHOULD AT LEAST HAVE A LOOK AT THESE FILES:
THEY CONTAINS ALSO USEFUL GENERAL PURPOSE INFORMATION.

2- Information concerning Mentor-Pascal:

The file pascal.procs.i (obtained on-line by the commands .list and .details) contains documentation about commands available in the mentor-pascal environment.

When starting to work in Pascal under Mentor-V5, one may want to parse some existing Pascal files (using the .parse command). A problem there is that lexical conventions often differ from one Pascal compiler to another.
For example, "not equal" can be # or <>
"and" can be and or &
comments may be enclosed in { } or (* *) or % %
access through pointers may be @ or ^ etc ....

Most things of this kind are known by the Pascal parser and unparser of Mentor-V5. Their behavior can be adapted by setting or resetting TOGGLES and DIALS (using the Mentol commands .set and .reset).

For example, setting the TOGGLE named "infsup" will make the parser recognize <> as the "not-equal" operator. Setting the TOGGLE named "amperbar" will make the parser recognize & and | as the "and" and "or" operators respectively.

For complete description of TOGGLES and DIALS see the file toggs-dials.i.

The command .charid (see mentor.procs.i) can be used to change the set of characters allowed in Pascal identifiers. One can for example used it to make characters like '$' or '_' accepted in identifiers by the Pascal parser.

In the directory .../mentor/test/pascal you will find small pascal programs you can load under Mentor-V5 to start trying the system with Pascal.

3- Information concerning Mentor-Rapport:

The file rap.procs.i (obtained on-line by the commands .list and .details) contains documentation about commands available
in the mentor-rapport environment.
The file Rapport_gi.i is the mentor-rapport user manual (in french).

In the directory .../mentor/test/rapport you will find a document you can load under Mentor-V5 to start trying the system with Rapport.

Documentation about the Rapport language and the Mentor Environment for Rapport can be found in following publications:

B. Mélèse,
Mentor Rapport: Manipulation de textes structurés sous Mentor
Rapport Technique No. 23, INRIA, Avril 1983

B. Mélèse,
Edition structurée - Edition non structurée, Coopération et complémentarité, 2ième Colloque de Génie logiciel, Nice, Juin 1984 (AFCET)

The first one is in french. It is given the file Rapport_gi.i in the directory mentor/info. The second one is also in french but has been translated in english. The english reference is:

B. Mélèse,
Structured Editing - Unstructured Editing, Cooperation and complementarity,

2nd Software engineering conference, Nice, June 1984

4- Information concerning Mentor-Metal:

The file metal.procs.i (obtained on-line by the commands .list and .details) contains documentation about commands available in the mentor-metal environment.

The file metal.i is the Metal user manual.
The file mentor.traps.i contains useful practical information for those who will define a language in Metal.

In the directory .../mentor/test/metal you will find a small language definition you can load under Mentor-V5 to start trying the system with Metal.

5- Information concerning Mentor-Ada:

The file ada.procs.i (obtained on-line by the commands .list and .details) contains documentation about commands available in the mentor-ada environment. The ADA environment provided is a minimun environment.

In the directory .../mentor/test/ada you will find small Ada programs you can load under Mentor-V5 to start trying the system with Ada.

6- For users who want to write new commands in mentol:

Read again the file mentor.gi.i
The file mentor.wr_pr.i indicates how to write mentol programs in mentol files.
The file mentor.mess.i indicates how to use the message system of mentor.

The good way to write new mentol command is to look at the mentol library in .../mentor/mentol that contains all already written mentol programs.

7- To create a new language under Mentor-V5.

Read again the documentation about Metal, and see the files mentorkit.i and mentor.dec.i.

If you want to modify the dialect of Pascal implemented in Mentor-V5, see pascalkit.i.

If you want to modify the definition of ADA as it exists in Mentor-V5 (for example if you find bugs in it ...) see adakit.i.

8- For users who want to write software around Mentor-V5:

It is possible to write programs (in C or pascal) to be linked with Mentor-V5 and that access Mentor-V5 data structure and make profit of the Mentor-V5 abstract syntax tree manipulation primitives.

To help in doing this, and interface with the Mentor-V5 internal structure is given in the file INTERFACE.p. (see the info file interface.i)

Good Luck !!

For all problems concerning Mentor-V5, please contact

Bertrand Mélèse at INRIA.

The full address is

INRIA
Domaine de Voluceau - Rocquencourt
B.P. 105 78153
LE CHESNAY CEDEX
FRANCE.

network address is ...!decvax!mcvax!inria!melese

Mentor.Ver5.i

New features in the version 5 of Mentor

and main differences between Version 5 and previous versions

Bertrand Mélèse

November 1984

1--

The version 5 of Mentor (called Mentor-V5) is a multi-language Version. It supports the annotation mechanism that has been described in recent publications about Mentor:

-- V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse, E. Morcos,
Outline of a tool for document manipulation,
IFIP, september 1983, Paris

-- V. Donzeau-Gouge, B. Lang, B. Mélèse,
Practical Applications of a Syntax Directed Program Manipulation Environment,
Proceedings of the 7th Int. Conf. on Soft. Eng., Orlando, Florida, March 1984

-- V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse,
Documents Structure and Modularity in Mentor,
ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments, Pittsburgh, April 1984
SIGPLAN Notices, Vol 19, No 5, May 1984.

The use of the multi-language aspect of Mentor-V5 is described in the file multi-lang.i.

The use of the annotation mechanism is described in the file annotation.i.

2-- *** WARNING *** (For users that have already defined a language under Mentor or wrote pascal programs linked to Mentor using the interface TREEFRONT.p)

Users that have already defined a language under Mentor using a previous version (4.n) will have to MODIFY slightly their Metal programs and their unparsers to become compatible with Mentor-V5.
We are sorry for this but it is not possible to keep all old code for ever.

i) Modifications that should be made in the Metal program

In the Mentor-V5, a minimal format for unparsing of annotations (see annotation.i) of the defined language may be specified. When it is not, a default format is used (see below). This format is kept in an annotation called ANNOT hooked to the definition of the "comment" node of the abstract syntax.

Comments in program are a special case of annotations. This format is then also used to unparse comments.

The easiest manner to create this annotation is by calling the new command .comment_format of the Metaluser environment. (see the description of that command in metal.procs.i)

This format contains:

The annotation delimiters
The maximum level of visible annotations.

Annotations of level 1 are annotations hooked directly to a node of the main tree. Annotations of level n+1 are annotations hooked to a tree which is an annotation of level n.

Examples:

In Pascal the annotation delimiters are (* and *) by default and the maximum level of visible annotations is 1 because in most Pascal compilers, nested annotations (comments) are prohibited.

In Ada the annotation delimiters are -- and EOL (end_of_line) and the maximum level of visible annotations is unlimited.

The ANNOT annotation described here is actually very minimum and is just a quick adaptation of Metal to cope with new features about annotations needed in the multi-language version of Mentor (i.e. Mentor-V5). It will be developped from now on to become a complete annotation definition language.

When unparsing format of annotations is not specified in a Metal program, a default format is automatically provided by Mentor-V5. This default format is the format of Ada annotations: the delimiters are -- and EOL and the number of visible levels in unlimitted.

ii) Modifications that can be made in the Metal program

In Mentor-V5 it is possible to specify a version number of the languages defined by a Metal program. This version number is recorded in polish files allowing Mentor-V5 to check it when loading a polish file.

A polish file can then be loaded if the version number it contains is not gr ater than the current version number of the language (the current version number of a language is given by Mentor-V5 when this language is loaded under Mentor-V5). This means that, increasing the version number of a language always keeps previous polish files loadable (provided that the modifications made in the Metal program do not affect existing operators: see metal.traps.i) but decreasing this version number disallows the loading of polish files that were created by the previous version of the language.

The version number is specified in a special annotation called VERSION (see annotation.i for explaination of what an annotation is) of the

Metal program. It can be created or changed by using the command .version_number of the metaluser environment.

iii) Modifications that MUST be made in UNPARSERS and in Pascal programs written using the interface TREEFRONT.p.

**** WARNING: MODIFICATIONS (2) (3) and (4) below MUST BE MADE. YOUR UNPARSERS AND PROGRAMS HAVE NO CHANCE TO RUN WITHOUT THEM !!!

A DECSKELETON adapted to Mentor-V5 is given in info/DECSKELETON.p (and info/DECSKELETON.po).

A new interface TREEFRONT.p (.po) is given in the info directory. (see also the info file interface.i)

The differences from previous versions are:

(1) Unparsing of annotations should no longer be done by the user's unparser.

It should be done by the procedure DECCOMMS (instead of DECCOM). The advantage of the new one is that it knowns how to unparse not only textual annotations but also annotations possibly in a different language. However, if you do not modify your unparser in this respect, it will still work in mono-language mode as in previous versions of Mentor.

(2) Because of the multi-language, some Pascal procedures now have one more parameter which is the current language. For example the procedures CHILD and FATHER now take the language as parameter.

(3) The type TDECOMPARAMS and the imported variable DECOMPARAMS no longer exist. Now, the exported procedure in unparsers takes directly three parameters : the tree to unparse, the holophrast and the language.

(4) The body of the exported procedure in unparsers has changed slightly. You have to take these changes in account in your already written unparsers.

The right manner to adapt your programs is to make a listing of TREEFRONT.p and DECSKELETON.p and to compare them carefully with your programs.

The great advantage of the new interface and the new unparsers is that they are compatible with the multi-language aspect of Mentor-V5 and with the annotation mechanism. This means that, in Mentor-V5, any user defined language can be put in annotations of any language and can receive annotations written in any other languages. Try it, it's fun !!!

3--

In version 5, all languages existing under Mentor are grouped into one system, just called mentor, instead of being separated in several systems (mentorada, mentorpascal, mentorrapport) as it was in version 4.3.

The standard languages available in mentor are:
Pascal, Ada, Metal, Rapport, Flip.

By using the mentorkit facility (see mentorkit.i) a user can add new languages in Mentor-V5.

The only separate system that remains in version 5 is GENERATOR that contains the Metal compiler. This is because the Metal compiler is large and seldom called. Thus, generator has to be used instead of mentor to compile a Metal program. The only language known by generator is Metal.

4--

In Mentor-V5 the Rapport environment supports multi-language documents. The program paragraphs (progpar) can be in any of the languages known by Mentor-V5, that is the languages provided in the distribution tape plus all user defined languages that have been added to Mentor-V5 using the mentorkit facility. (see the description of the commands .creer and .edit in the file rap.procs.i).

5--

In multi-language sessions under Mentor-V5 it is NOT SAFE to load several mentol environments. For example, when working both with Pascal and Metal, ones have to choose when entering Mentor-V5 if he prefers to have the Pascal programming environment OR the Metal programming environment. Loading both environments at the same time is unsafe because of possible name clashes between global Mentol variables and commands.

To make it SAFE we have to carefully clean all our Mentol code. This will be done in future versions.

6--

A new menu driven input mode exists in Version 5. This mode is well suited for beginners (and even for experts during multi-language sessions). This mode is language independant and can then be used even with user defined languages (see beginners.i). The menu mode is called by issuing the .menu command.

7-- The Metal compiler has been improved: it is now more robust and the matching of lists has been implemented (see metal.traps.i).

8-- The predefined command ".comment" no more exists in Mentor-V5. It is replaced by ".defframe" (see annotation.i). The new command ".defframe" is compatible with the old ".comment". It is enough to change the name ".comment" by ".defframe" in the Mentol programs. The new command ".defframe" is more powerful and robust than .comment was: see its description in the file annotation.i.

9-- The commands ".hide" and ".show" no more exist in Mentor-V5. They are no more needed because of improvements made in the Mentor-V5 redisplay.

10-- The documentation has been improved.

11-- A lot of known bugs have been fixed
(and a lot of still unknown bugs have been inserted !!)

12-- The Mentor-V5 is available on Vax under Berkeley Unix 4.1 and 4.2,
on SM90 under SMX and will be soon available on Multics.

mentor.gi.i

THE MENTOR PROGRAM MANIPULATION SYSTEM
(1979)

Updated November 1984 by

Bertrand Mélèse

The introduction contains information to help people to get started with Mentor. Following sections are a rough user's manual for the MENTOR manipulation language: MENTOL. Some brief examples are given and a sample session is included in the appendices (both using Pascal as sample language). No attempt is made to describe the Philosophy of the System, its history or future plans: this can be found in recent publications about Mentor (see annotation.i and multi-lang.i).

MENTOR is a program manipulation system developed at INRIA by V.Donzeau-Gouge, G.Huet, G.Kahn and B. Lang and B. Mélèse. MENTOR manipulates programs represented as abstract syntax trees, using a specialized programming language called MENTOL. MENTOR is not specialized to any specific programming language. In the version 5 (Mentor-V5) available languages are: Pascal, Ada, Metal, Flip, Rapport.

This manual is an introductory manual to MENTOL. See also the info file mentor.wr_pr.i to know what is the format of Mentol files and for more information about how to write Mentol procedures. A lot of Mentol files can be found in the directory .../mentor/mentol for those who prefer to learn Mentol programming from examples.

MENTOR is entirely written in PASCAL, and was bootstrapped from the start of the project. The initial version was running on the CII IRIS 80 computer. A DEC 10 version was obtained from it completely mechanically, using MENTOR itself. The Multics version was easier to obtain since the Multics PASCAL compiler was designed to run the IRIS 80 PASCAL programs with minimal changes. The first VAX-Unix Version has been obtained from the Multics version mechanically using Mentor. The SM90 version has been obtained from the Vax version.

Introduction

To run Mentoron Vax/Unix, one must define a variable (in his/her .login or .profile file) called MENTOR containing the directories where Mentor searches for tables and for Mentol libraries. If we suppose that the Mentor directory has the path-name /usr/local/mentor this variable must be defined as follows:

setenv MENTOR /usr/local/mentor/langtbls:/usr/local/mentor/mentol (in csh)

or

MENTOR=/usr/local/mentor/langtbls:/usr/local/mentor/mentol (in sh)

To get the executable code of Mentor the PATH must be completed by the following directory:

/usr/local/mentor/exe

The Pascal compiler used by Mentor:

A special case has to be made for the pascal compiler. Changes have been made in the BERKELEY pascal compiler to handle separate compilation in a way closer to the standards. Then we use a local version of the pascal compiler named mpc and located in the directory mentor/mpc.

Differences between pc and mpc can be found in the file
/usr/local/mentor/info/PC.changes.i.

MENTOR

The distributed Version of Mentor is now the version 5.0.

This version is multi-languages. Available languages in the standard distribution tape are: Pascal, Ada, Rapport, Metal, Flip.

Users can also define their own languages and introduce them under Mentor (see mentorkit.i, metal.i, metal.traps.i, mentor.dec.i).

Differences between Version 5 of Mentor and previous versions are listed in the file Mentor.Ver5.i. Specific information about the multi-language facility and the annotation mechanism can be found in annotation.i and multi-lang.i.

When you call the Mentor system, it first asks for a user NAME. It then uses this NAME to find the appropriate environment to load. In that manner each user can define his own environment and load it when entering the system. On Unix, the file loaded when a name is given is the file called NAME.pre (see multi-lang.i).

There are predefined environments that you can use before having your own environment:

To work in PASCAL under mentor using the full Mentor-Pascal programming environment, use the predefined user name "pascaluser".

To work in METAL under mentor or generator use the user name "metaluser". Generator in a special version of Mentor that contains only the language Metal and the Metal compiler. Since the Metal compiler is big and is not used very often it has been removed of mentor and kept only in generator. Thus, generator has to be used instead of mentor to compile Metal programs.

To work in RAPPORT, use the user name "rapuser".

To work in ADA, use the user name "adauser".

During a Mentor session you can call the internal help system of Mentor.
It is made of five commands:
.help, .ginfo, .list, .details, and .sysdetails

USING MENTOR ON YOUR TERMINAL

Mentor uses the termcap definition of terminals.

To run Mentor properly your terminal must have enough capabilities to support a full screen editor (Emacs-like) AND must have at least two different fonts. If you have a terminal with many fonts good choices for Mentor are to use the standard font as primary font and bold or underlined as secondary font.

The secondary font Mentor uses is the font defined by the "so" and "se" attributes in the termcap definition of the terminal. (see documentation about termcap in the Unix documentation).

It is comfortable to use function keys of the terminal to move in the structure, and to control screen display. This will be possible if your terminal as programmable function keys. In fact Mentor needs 8 programmable function keys. They must be programmed to send, respectively, the following strings:

.p<CR> , .q<CR> , .r<CR> , .s<CR> , .t<CR> , .u<CR> , .v<CR> , .w<CR>

The first character, '.', may be replace by ESC if more convenient.

If we call f1 to f8 these function keys, their effect under mentor is the following :

(The video pointer is the pointer to the sub-tree displayed on the screen. The current pointer is the pointer to the sub-tree on which modifications can be applied.)

f1 to f4 only affect the video pointer. (see mentor.video.i)

f1	->	Clean screen
f2	->	Big Zoom : The video pointer becomes equal to the current pointer.
f3	->	Un-zoom : The video pointer goes one level up.
f4	->	Little Zoom : The video pointer goes one level in the direction of the current pointer.

f5 to f8 only affect the current pointer. However, the video pointer can be moved automatically to keep the current pointer on the screen.

f5	->	Go to the father
f6	->	Go to left brother
f7	->	Go to right brother
f8	->	Go to the next node in a preorder traversal

(see mentor.video.i for more details about the Mentor display commands)

PARSERS:

An other point that I want to mention here is that Mentor makes extensive use of parsing. Parsers are built using yacc and lex. (Only the PASCAL parser is wired into Mentor). The source files for YACC are generated by the Metal compiler. One has to complete the source file for LEX starting from the initial lex file generated by the METAL compiler and containing the definitions of the TOKENS.

To make your own MENTOR system working with your personnal language, see the documentation about METAL and the files mentorkit.i and metal.traps.i.

Another important point to note is that Mentor makes extensive use of the character @. If in your environment @ means "kill line" or has some other funny meaning it will be better to restore it.

For all problems concerning Mentor, please contact

Bertrand Mélèse at INRIA.

The full address is

INRIA
Domaine de Voluceau - Rocquencourt
B.P. 105 78153
LE CHESNAY CEDEX
FRANCE.

network address is ...!decvax!mcvax!inria!melese

1. Filing System

MENTOR uses the suffix of the files it handles to distinguish their types, according to the table below:

Suffix	Type
p	Text file in PASCAL
metal	Text file in METAL
rapport	Text file in RAPPORT
ada	Text file in ADA
po	Tree form of a program (polish files)
tol	Command file for MENTOR (in Mentol)
pre	Prelude file (see multi-lang.i)
i	Info files
mess	message files (see messages.i, useful for implementors only)

You never have to worry about these suffixes. They are automatically

supplied by the system. They are indicated here so that you understand what is happening in your directory.

Each filing command, when typed with no parameter, prompts you with:

```
File name:
```

to which you would reply WITHOUT mentioning any extension.

Here are the main commands for dealing with files (see mentor.procs.i):

```
.parse
.unparse
.store
.load
```

Assume you want to start using MENTOR with an already existing PASCAL program, called EX1.p. You should type:

```
.parse
File name: ex1
```

This creates a file that contains the tree representation of your program. Later on, you will keep all your programs in tree form. To create a file containing the tree form of your program type

```
.store
File name: ex1
```

To load the tree form, just type:

```
.load
File name: ex1
```

At some point, you will want to make a text file (for input to the PASCAL compiler for example). The command to use is:

```
.unparse
File name: ex1
```

REMARK: All filing procedures may be called with arguments. In that case, the first argument must denote a tree reduced to a single identifier that will be taken to be the file name. You will find this useful whenever you wish to COMPUTE the name of a file. (see mentor.procs.i).

In each existing programming environments (loaded through the predefined user names) there exists commands to save both the tree form and the text form in files whose name in computed from the program name. See the specific information files:

```
mentor.procs.i          for Pascal
```

metal.procs.i	for Metal
ada.procs.i	for Ada
rap.procs.i	for Rapport

2. Addressing Mechanism

MENTOR manipulates programs in tree form. To manipulate programs, you need to know how to move about in trees, to give a name to a subtree. In fact, you need to understand MENTOR's addressing mechanism, in the way you have to understand the addressing mechanism of any usual text editor.

2.1. Markers

The variables of Mentol are markers into program trees. There is a conventional marker, called the CURRENT marker, that is used implicitly in most usual manipulations. If you want to name it explicitely, it is called @K. All markers are of the form:

@<identifier>

(where the identifier is a sequence of alphanumeric characters, starting with an alphabetic character).

2.2. Structural Addresses

A structural address denotes the location of a subtree with respect to some marker used as a base. The exact syntax is:

```
<structural address> ::= <marker>   |
                         <structural address><modifier>.        (1)

<modifier> ::= <immediate modifier>   |
               <search modifier>

<immediate modifier>::= U <number>   |     (up)
                        L <number>   |     (left)
                        R <number>   |     (right)
                        S <number>   |     (son)
                        B0           |     (prefix attribute)
                        B1           |     (postfix attribute)
                        ^ <ident>    |     (annotation)
                        U0           |     (antiattribute)
```

For complete explainations of B0, B1, ^ and U0, see annotation.i.

```
<search modifier> ::=   F <schema>   |     (within the subtree)
                        FF <schema>  |     (within subtree then further)

<schema> ::= <structural address>    |     (schema in store)
             D <structural address>  |     (extracted schema)
```

```
<structural address> Z <number>  |        (sublist schema)
&                                |     (input schema from terminal)
E <schema>                       |   (schema obtained by instantiation)
```

The modifiers in equation (1) are performed sequentially from left to right.

2.2.1. Immediate modifiers

For U,L and R the <number> on the right indicates repetition of the modifier. For S , Sn denotes the n-th son and Sn is equivalent to: S1 R(n-1)

Negative numbers are accepted on the right of S and S-1 designates the last son, S-2 the one before last, etc.....

A star may be used instead of a <number>. The star denotes the largest possible integer that makes sense for a given modifier. For example:

```
S*        denotes the last son
R*        denotes the right-most brother
```

Example:

```
@here U
@here R2
@here S*
@here S2 S3
```

ABBREVIATIONS: Whenever the base marker is @K, it may be omitted. When there is no chance of ambiguity, you can also omit S. So, in the proper context:

```
@K S1      is abbreviated by     1
```

The modifiers B0, B1 and U0 allow you to move from one tree to its predefined prefix and postfix annotations and back. These predefined annotations may either be textual comments or trees in the same language as the tree they annotate. These annotations may be annotated recursively in the same manner. If you are at the top of an annotation, the command U0 will denote the subtree that this is an attribute of, otherwise it will fail. In Mentor-V5 a powerful multi-language annotation mechanism is available. See the information files annotation.i and multi-lang.i.

2.2.2. Search Modifiers

(A) All search modifiers perform a preorder search for the first occurrence of the schema given as a parameter. A schema is a tree (i.e. a fragment of a program) that may contain some special leaves called METAVARIABLES. When they are represented, these metavariables appear as:

$<identifier>

A given tree A is an instance of a schema S iff A may be obtained from S in substituting the metavariables of S by some subtrees.

Example (in PASCAL):

Assume the marker @SCH denotes the schema

```
if $V1 then $V2 else X:=$V3
```

and that the marker @TXT denotes a tree represented by:

```
begin
X:=Y
if T=0 then Y:=0
else if B>0 then Z:=0 else X:= A+1
end
```

The address:

```
@TXT F @SCH
```

denotes the most internal if.

The evaluation of search modifiers has a side effect: it assigns to the markers whose names are those of the metavariables of the argument schema the relevant subtree locations. As a consequence, you should only use LINEAR schemas (i.e. without repetition of metavariables) as arguments of searches.

In the example above, after evaluating the address @TXT F @SCH, the markers @V1, @V2 and @V3 denote respectively the occurrences of B>0, Z:=0 and A+1 in the tree denoted by @TXT.

(B) When using the D (delete) modifier, the structural address denotes the schema obtained after destructive extraction from its surrounding tree.

(C) The Z operator allows to build a sublist of a list. The structural address on the left must be that of a list element, and the number on the right denotes how many elements ih the list should be taken.

EXAMPLE: S-3 Z3 denotes a copy of the last 3 elements of a list.

(D) The symbol & (ampersand) is used whenever a schema must be input from the console. Whenever the symbol & is evaluated, MENTOR tries to see from the context what should be parsed. If it is possible, the user is prompted by the kind of subexpression expected, such as:

[STAT]:

If it is not, the user is prompted with:

[

and should specify first what to parse, typing for example:

stat]<CR>

then it is prompted with a colon indicating that MENTOR is ready to PARSE.

EXAMPLE: To enter a program from the terminal all you have to type is:
:&
Then Mentor will prompt you with:
[
You will then indicate that you want to enter a program by typing:
program]<CR>
then you will be prompted by a colon indicating that Mentor is ready to parse a program. Mentor will wait for input until the program is complete, that is, until ou have entered 'end.' .

(E) Predefined Patterns

To every construct in the abstract syntax corresponds a predefined schema. If OP is such a construct, @OP denotes the associated PREDEFINED schema (see also multi-lang.i). The name of the metavariables in the schema corresponds to the name of the sorts of the operands. In the case of a list construct, the associated predefined schema is a one element list. The predefined operators are very handy to move about in structural trees. For example the schema @IF denotes:

if $EXP1 then $STAT1 else $STAT2

and F @IF denotes the first conditional statement encountered in the current expression. A complete list of predefined schemas of Pascal can be found in Appendix B.

(F) Instantiation

The expression E <schema> denotes a new schema obtained after substitution of the metavariables in the argument schema by the subtrees denoted by the markers with the same name. The instantiation facility fits in smoothly with the search capability.

EXAMPLE: Assume the current expression denotes

A+B

and you wish to transform + into -. This may seem difficult at first since the operator + sits at a node. But since the

schemas @PLUS and @MINUS denote respectively
$EXP1+$EXP2 and $EXP1-$EXP2 , all you have to type is

F @PLUS C E @MINUS

The search modifier collects an environment that is then used for instantiation.

If you wish to apply rewrite rules to your programs, you will want to use E.

3. Primitive Commands

You can manipulate trees via commands, or procedures. The programming language of MENTOR is called MENTOL. MENTOL does what you want it to do, if not always elegantly or safely.

3.1. Handling of Markers

3.1.1. Assignment

<Assignment> ::= <marker>:<structural address> |
<marker>:& .

The second form is to be used when you want to input PASCAL text without modifying another tree.

ABBREVIATION: When the marker on the left hand side is also the base of the structural address on the right, you may write instead:

<structural address>

with the same meaning.

EXAMPLE:

@T:@T S2

may be written simply

@T S2

NOTE: This abbreviation combines with the elision of @K so that @K : @K S2 may be written S2 or even 2.

3.1.2. Stack Manipulation

Each marker denotes in fact, the top of a stack of tree references. This stack is handled with the commands:

<marker> PD	(push-down)
<marker> PU	(pop-up)
<marker> PT	(push-transpose)

3.2. Handling Trees

3.2.1. Printing (in teletype mode only, see mentor.video.i)

<printing> ::= <structural address> P <number> .

The number on the right must be a positive integer or *. The tree denoted by the lefthand side is displayed on the teletype up to the level of detail requested by this number. It is shown entirely if * if used. The text of the program is abbreviated with and ... . You can also omit the level of detail, in which case the default level is 5. So that:

P

is taken to mean,

@K P 5

In full screen mode, the control of the level of details in unparsed program in done with the command '#'. See mentor.video.i.

3.2.2. Copying

<copy> ::= <marker> = <schema> .

After the execution of this command, the <marker> denotes a brand new tree in all cases, so that <marker>U is undefined.

3.2.3. Deleting

<destruction> ::= D <structural address> .

The designated subtree is extracted from its context. The top of the special @DUMP marker now denotes this subtree. This allows you to go backwards if you delete a tree by mistake. The marker @DUMP may contain the n+1 last deleted trees, provided that you have executed @DUMP PD n times beforehand.

CAUTION: It is not a good idea to saw the branch of the tree on which you are sitting, when it is a list element. This will make you jump to the father list node. If you are worried about this, avoid using the command "D" by itself.

3.2.4. Changing Trees

<change> ::= <structural address> C <schema> .

The schema is put at the structural address. The marker @DUMP is updated with the tree that has been dislodged. MENTOR checks that the tree you try to insert belongs exactly where you have tried to put it, and rejects your command otherwise.

3.2.5. Exchanging Trees

<exchange> ::= <structural address> X <structural address> .

The subtrees are exchange, types are checked. Note that this instruction makes sense only when the addresses denote disjoint subtrees.

3.2.6. Insertion Within Lists

```
<insertion> ::= <structural address> I <schema>    |
                <structural address> J <schema>    |
                <structural address> II <schema>   |
                <structural address> JJ <schema>   .
```

In all cases, the address on the left must denote a list element t. If you use I, the tree on the right is inserted after t. If you use J, the tree is inserted before t. Of course, syntactic consistency checks are performed. With II and JJ, the tree on the right hand side must be a list of the same kind as the one you insert into. This whole list is merged in either after t (with II) or before t (with JJ) .

Thus, the primitive commands of MENTOR are:

```
<primitive command> ::= <assignment>      |
                        <stack-handling>  |
                        <printing>        |
                        <copy>            |
                        <destruction>     |
                        <change>          |
                        <exchange>        |
                        <insertion>       .
```

It is useful to be able to build a sequence of these commands and to control their flow of execution, just like in any programming language.

4. Control Constructs

4.1. Sequencing

```
<command list> ::= <command> ; <command list>   |
                   <command>    .
```

Commands in such a list are executed from left to right as soon as the command list has been received (i.e. after <CR>).

Now a command is defined this way:

```
<command> ::= <empty command>     |
              <primitive command> |
              <loop>              |
              <test>              |
              <case>              |
              <exit>              |
              <procedure call>    .
```

The way control proceeds is akin to SNOBOL or TECO. Every command may succeed or fail. If, in a command sequence, a command fails, the sequence is interrupted and fails, UNLESS the next command is a test.

4.2. Loop

```
<loop> ::= ( <commandlist> ) <number>    .
```

The command is iterated as specified. If * is specified, the command list is iterated forever (i.e. probably until it fails or jumps out). If the number is OMITTED, it is defaulted to 1. In this way, the parentheses act as usual for you.

4.3. Test

```
<test> ::= ? <command1> , <command2>     .
```

If the previous command succeeded, the first command is executed otherwise the second one.

4.4. Case Command

```
<case> ::= <command1> / <command2> .
```

Command1 is executed. If it suceeds, then command2 is executed, and then the ENCLOSING list of commands is terminated.

If command1 fails, command2 is not executed, but the case command does NOT fail.

4.5. Exit

<exit> ::= $ <integer> |
$ - <integer> .

If the integer's value is n, the command jumps out of n levels of loop. Then, if the minus sign is specified, the command fails. If the integer is omitted, it defaults to 1.

4.6. Procedure Call

All MENTOR procedures (there are, alas, no functions) have a name in the form:

. <identifer>

A procedure call looks like an Algol procedure call, except that pointed brackets are used instead of parentheses.

Procedures may take either structural addresses or commands as parameters.

4.7. Comments

Comments are introduced with a % symbol and terminated either with an end of line or with another %.

Appendix A -Sample session

The following is an example of a Mentor session with the language Pascal. A future Mentor user should read it very carefully to understand each manipulation . To help the reader, a complete listing of the program that is manipulated is found at the beginning of the session. Note that the file BERTRAND.pre is just a copy of the file PASCALUSER.pre.

This session does not reflect the video display on the terminal!!. It has been done on a paper terminal for obvious reasons. All the printing commands (the commands of the form pn where n is an integer) do not have to be done by the user using a screen terminal.

Last remark: This session has been done a long time ago using a old version of Mentor on Multics but this is not disturbing to get the filling of the system.

```
mentor
- MENTOR  VERSION 4.2 / May 83  ON VAX-UNIX (Berkeley)
USER NAME:Bertrand
BERTRAND.pre FILE LOADING
? % this question mark is the Mentor prompt symbol
? % Each line beginning with a percent is a comment line
? % Load a Pascal program which is in the file BASIS.p and create the
? % tree form.
?.parse
FILE NAME:basis
? % Now I have a current expression which is a full program pointed by the
? % current marker called @k
? % Print this program under different levels
?@k p2

(*********************)
(*.LAST UPDATE :    .*)
(*10/10/80  15:33    *)
(*.BY :             .*)
(*BERTRAND*)
(*********************)
(**)
program BASIS(INPUT,OUTPUT);
     ...;
....
?@k p3

(*********************)
(*.LAST UPDATE :    .*)
(*10/10/80  15:33    *)
(*.BY :             .*)
(*BERTRAND*)
(*********************)
(**)
program BASIS(INPUT,OUTPUT);
     ...;...;#;#;#;#;
     begin
     #;#;#;#;#;#
     end.
? % the symbol # stands for a sub-tree whose root is a fixed arity node
? % the symbols  ...  stand for a sub-tree whose root is a list node
?@k p4

(*********************)
(*.LAST UPDATE :    .*)
(*10/10/80  15:33    *)
(*.BY :             .*)
(*BERTRAND*)
(*********************)
(**)
program BASIS(INPUT,OUTPUT);
     type #;
     var  #;#;#;#;#;
```

```
     function PPCM ...:INTEGER;
          ...;
     ...;

     procedure INITIAL;
          ...;
     ...;

     procedure FILTER;
          ...;
     ...;

     procedure XLOOP ...;
          ...;
     ...;

     begin
     INITIAL;
     XLOOP(1,0,INIMAXSUM,INIMAX);
     L:=DIM (*JUMP TRIVIAL SOL*);
     while # do ...;
     for I:=# do #;
     WRITELN
     end.
? % We can see the whole program by the command p*. We print all
? % here for reference throughout the session.
?@k p*

(********************)
(*.LAST UPDATE :   .*)
(*10/10/80  15:33   *)
(*.BY :            .*)
(*BERTRAND*)
(********************)
(**)
program BASIS(INPUT,OUTPUT);
     type TABLE   =array[1..10]of INTEGER;
     var  INIMAX:TABLE;
          COEF,SOL:array[1..20]of INTEGER;
          D,E:array[1..10,1..10]of INTEGER;
          SOLS:array[0..10000]of INTEGER;
          TIM1,TIM2,LASTSOL,MAXA,MAXB,INIMAXSUM,I,J,CDIM,L,M,N,DIM:INTEGER;

     function PPCM(X,Y:INTEGER):INTEGER;
          var  P:INTEGER;
          begin
          P:=X*Y;
          while X#Y do
               if X>Y then X:=X-Y else Y:=Y-X;
          PPCM:=P div X
          end;

     procedure INITIAL;
          var  I1,J1,P1:INTEGER;
```

```
     begin
     MAXA:=0;
     MAXB:=0;
     INIMAXSUM:=0;
     READ(M,N);
     DIM:=M+N;
     WRITELN('M=',M:2,' N=',N:2);
     WRITELN('A[I]:');
     for I1:=1 to M do
          begin
          READ(P1);
          WRITE(P1:4);
          COEF[I1]:=P1;
          if MAXA<P1 then MAXA:=P1
          end;
     WRITELN;
     WRITELN('B[J]:');
     for J1:=1 to N do
          begin
          READ(P1);
          WRITE(P1:4);
          COEF[M+J1]:=P1;
          INIMAX[J1]:=MAXA;
          INIMAXSUM:=INIMAXSUM+P1;
          if MAXB<P1 then MAXB:=P1
          end;
     INIMAXSUM:=INIMAXSUM*MAXA;
     WRITELN;
     WRITELN;
     LASTSOL:=-DIM;
     CDIM:=COEF[DIM];
     for I1:=1 to M do
          for J1:=1 to N do
               begin
               P1:=PPCM(COEF[I1],COEF[M+J1]);
               D[I1,J1]:=P1 div COEF[I1];
               E[I1,J1]:=P1 div COEF[M+J1]-1
               end
     end;

procedure FILTER;
     label
          1;
     var  Q,L1:INTEGER;
     begin
     L1:=DIM (*JUMP TRIVIAL SOL*);
     while L1<=LASTSOL do (*TEST WHETHER SOLS[L]<SOL*)
          begin
          Q:=2 (*BECAUSE ALWAYS TRUE OF 1*);
          while SOLS[L1+Q]<=SOL[Q]do
               if Q=DIM then goto 1 else Q:=Q+1;
          L1:=L1+DIM
          end;
     LASTSOL:=LASTSOL+DIM (*SOL HAS NOT BEEN FILTERED OUT*);
     for Q:=1 to DIM do SOLS[LASTSOL+Q]:=SOL[Q];
```

```
      1:
      end;

 procedure XLOOP(I2,SUM,MAXSUM:INTEGER;MAXY:TABLE);
      label
           3;
      var  K:INTEGER;
           L:INTEGER;

      procedure YLOOP(I3,REST:INTEGER);
           label
                2;
           var  K1:INTEGER;
           begin
           if I3#DIM then
                for K1:=0 to MAXY[I3-M]do
                     begin
                     SOL[I3]:=K1;
                     YLOOP(I3+1,REST);
                     REST:=REST-COEF[I3];
                     if REST<0 then goto 2
                     end
           else if REST mod CDIM=0 then
                     begin
                     K1:=REST div CDIM;
                     if K1<=MAXY[N] then
                          begin
                          SOL[DIM]:=K1;
                          FILTER
                          end
                     end;
           2:
           end;

      begin
      for K:=0 to MAXB do
           begin
           for L:=1 to N do
                if K=D[I2,L] then
                     if MAXY[L]>E[I2,L] then
                          begin
                          MAXSUM:=MAXSUM-(MAXY[L]-E[I2,L])*COEF[M+L];
                          MAXY[L]:=E[I2,L]
                          end;
           if SUM>MAXSUM then goto 3;
           SOL[I2]:=K;
           if I2<M then XLOOP(I2+1,SUM,MAXSUM,MAXY)else YLOOP(I2+1,SUM);
           SUM:=SUM+COEF[I2]
           end (*FOR K*);
      3:
      end;

begin
INITIAL;
XLOOP(1,0,INIMAXSUM,INIMAX);
```

```
     L:=DIM (*JUMP TRIVIAL SOL*);
     while L<=LASTSOL do
          begin
          for I:=1 to DIM do WRITE(SOLS[L+I]:4);
          WRITELN;
          L:=L+DIM
          end;
     for I:=1 to M do
          for J:=1 to N do
               begin
               for L:=1 to I-1 do WRITE('   0');
               WRITE(D[I,J]:4);
               for L:=I+1 to M+J-1 do WRITE('   0');
               WRITE(E[I,J]+1:4);
               for L:=J+1 to N do WRITE('   0');
               WRITELN
               end;
     WRITELN
     end.
?
? % Examples of high level manipulations
? % go to the top of some procedure or function
?.fproc
procedure or function name
[IDENT]:xloop
?@k p

procedure XLOOP(I2,SUM,MAXSUM:INTEGER;MAXY:TABLE);
     label
          3;
     var  K:INTEGER;
          L:INTEGER;

     procedure YLOOP(#);
          ...;...;
          begin
          #;#
          end;

     begin
     for K:=0 to MAXB do
          begin
          #;#;#;#;#
          end (*FOR K*);
     3:
     end
? % P alone means P5
? % go to the body of this procedure
?.body
? % print it . Remember that @k may be ommited
?p
     begin
     for K:=0 to MAXB do
          begin
          for L:=# do #;
```

```
          if # then #;
          #:=K;
          if # then # else #;
          SUM:=#
          end (*FOR K*);
     3:
     end
?f @if %go on the first if from this point
?p
if K=D[I2,L] then
     if MAXY[L]>E[I2,L] then
          begin
          MAXSUM:=#;
          #:=#
          end
?@k:@k s1   % go to the first son of this tree, that is to the condition
?p
K=D[I2,L]
? % the current pointer was moved by the preceding assignement
? % now, change this expression by something that will be entered from the termi
nal
?@k c &
[EXP]:a or b;
?p
A' or B
?.up<@if>  % go back to the if statement
?p
if A or B then
     if MAXY[L]>E[I2,L] then
          begin
          MAXSUM:=#;
          #:=#
          end
? % go to the 'then' part which is the second son of the if (and which is
? %                                          another if statement)
?@k:@k s2
?p  % is equivalent to @k p  by the abbreviation rules
if MAXY[L]>E[I2,L] then
     begin
     MAXSUM:=MAXSUM-(MAXY[L]-E[I2,L])*COEF[M+L];
     MAXY[L]:=E[I2,L]
     end

? % We now want to change "<" in ">" in the condition.
? % Here we shall use (E)-Instantiation.
? % To do this, we first move
? % to this node using the  'f' command with  the predefined
? % schema  @gtr, and then change it by the  predefined schema @lss
? % At the same time we perform the instantiation of the meta-variables
? % of this predefined schema.
?
? f @gtr;p
MAXY[L]>E[I2,L]
? % The command f moves the pointer @k if the schema is found
? % Change, Instantiation and print the result
```

```
?@k c e @lss
?p
MAXY[L]<E[I2,L]
? % go up one level and print
?u;p
if MAXY[L]<E[I2,L] then
     begin
     MAXSUM:=MAXSUM-(MAXY[L]-E[I2,L])*COEF[M+L];
     MAXY[L]:=E[I2,L]
     end
? % go to the second son of that if statement
?s2  % is an abbreviation for  @k s2  which itself is an abbreviation for
?    %  @k:@k s2
?p
     begin
     MAXSUM:=MAXSUM-(MAXY[L]-E[I2,L])*COEF[M+L];
     MAXY[L]:=E[I2,L]
     end
? % go to the first statement of this list
? s1  %means @k:@k s1
?p
MAXSUM:=MAXSUM-(MAXY[L]-E[I2,L])*COEF[M+L]
? % go to the second part (right member of this assignement)
?s2;p
MAXSUM-(MAXY[L]-E[I2,L])*COEF[M+L]
? % ask for the type of this sub-tree
?.stype<@k>
MINUS
?s2;p
(MAXY[L]-E[I2,L])*COEF[M+L]
?.stype   % means .stype<@k>
MULT
?s1 p % means print the first son without going on it
MAXY[L]-E[I2,L]
?s2 p % same thing for the second son
COEF[M+L]
? s1 x s2   % exchange the first and second son
?p
COEF[M+L]*(MAXY[L]-E[I2,L])
? % declare a new variable in that procedure
?.decvar
list of variables
[LVARBL]:ars;
[TYP]:integer;
? % go to the variable declaration part of that block to see if every thing
? % is right
? % Save the current position to be able to come back easily.
?@here:@k
?.var;p
var  K:INTEGER;
     L:INTEGER;
     ARS:INTEGER
? % initialize  ars to zero at the beginning of the body and increment it
? % after the assignment  whose the second son is pointed by
? % @here
```

```
?.body
?sl j &  % insert before the first son (first son is the first statement)
[STAT]:ars:=0;
?p
     begin
     ARS:=0;
     for K:=0 to MAXB do
          begin
          for L:=# do #;
          if # then #;
          #:=K;
          if # then # else #;
          SUM:=#
          end (*FOR K*);
     3:
     end
?@k:@here
?@k p
COEF[M+L]*(MAXY[L]-E[I2,L])
?@k u;@k p
MAXSUM-COEF[M+L]*(MAXY[L]-E[I2,L])
?@k u;@k p
MAXSUM:=MAXSUM-COEF[M+L]*(MAXY[L]-E[I2,L])
?i &   % insert after the current position
[STAT]:ars:=ars+1;
?.up<@if>;p
if MAXY[L]<E[I2,L] then
     begin
     MAXSUM:=MAXSUM-COEF[M+L]*(MAXY[L]-E[I2,L]);
     ARS:=ARS+1;
     MAXY[L]:=E[I2,L]
     end
? % go to the top of the program and save it
?u*
?.save  % to save the tree form (polish)
FILE NAME:Newbasis
NEWBASIS.po  FILE CREATED
?.unparse    % to create the text form which will be used to compile
FILE NAME:newbasis
NEWBASIS.p  FILE CREATED
?
? % show all procedures and funtions heading
?.forall<@block, sl p>
(********************)
(*.LAST UPDATE :    .*)
(*10/10/80  15:33   *)
(*.BY :             .*)
(*BERTRAND*)
(********************)
(**)
program BASIS(INPUT,OUTPUT)
function PPCM(X,Y:INTEGER):INTEGER
procedure INITIAL
procedure FILTER
procedure XLOOP(I2,SUM,MAXSUM:INTEGER;MAXY:TABLE)
```

```
procedure YLOOP(I3,REST:INTEGER)
?.fproc
procedure or function name
[IDENT]:ppcm
?1 p
function PPCM(X,Y:INTEGER):INTEGER
? % put a prefix comment to that function
?b0 c &   % means @k b0 c &
[LLINE]:prefixe comment of the funtion PPCM%
? % put a postfix comment to the same function
?b1 c &
[LLINE]:postfix comment
:of
:the
:function
:PPCM%
?@k p

(*prefixe comment of the funtion PPCM*)
function PPCM(X,Y:INTEGER):INTEGER;
     var  P:INTEGER;
     begin
     P:=X*Y;
     while X#Y do
        if # then # else #;
     PPCM:=P div X
     end (*postfix comment*)
         (*of*)
         (*the*)
         (*function*)
         (*PPCM*)
? % go on the postfix comment
?@k:@k b1  % may be reduced to b1
?p
(*postfix comment*)
(*of*)
(*the*)
(*function*)
(*PPCM*)
?s4;p
function
? % go back to the top of the comment, then go out of the comment (by U0)
?u*;p
(*postfix comment*)
(*of*)
(*the*)
(*function*)
(*PPCM*)
?u0;p

(*prefix comment of the funtion PPCM*)
function PPCM(X,Y:INTEGER):INTEGER;
     var  P:INTEGER;
     begin
     P:=X*Y;
```

```
        while X#Y do
             if # then # else #;
        PPCM:=P div X
        end (*postfix comment*)
            (*of*)
            (*the*)
            (*function*)
            (*PPCM*)
  ? % when on top of a comment, u0 goes to the commented node
  ?
  ?
  ? % go to the next procedure declaration
  ?r;p    % r is an abbreviation for @k r  which is an abbreviation for @k:@k r

  procedure INITIAL;
       var  I1,J1,P1:INTEGER;
       begin
       MAXA:=0;
       MAXB:=0;
       INIMAXSUM:=0;
       READ(M,N);
       DIM:=M+N;
       WRITELN('M=',M:2,' N=',N:2);
       WRITELN('A[I]:');
       for I1:=1 to M do
            begin
            #;#;#;#
            end;
       WRITELN;
       WRITELN('B[J]:');
       for J1:=1 to N do
            begin
            #;#;#;#;#;#
            end;
       INIMAXSUM:=INIMAXSUM*MAXA;
       WRITELN;
       WRITELN;
       LASTSOL:=-DIM;
       CDIM:=COEF[DIM];
       for I1:=1 to M do
            for J1:=# do ...
       end
?f @for;.body;p
       begin
       READ(P1);
       WRITE(P1:4);
       COEF[I1]:=P1;
       if MAXA<P1 then MAXA:=P1
       end
?s1;p
READ(P1)
?r;p
WRITE(P1:4)
?r;p
COEF[I1];=P1
```

```
?r;p
if MAXA<P1 then MAXA:=P1
?r;p
E 25 R-FAIL
?1;p
COEF[I1]:=P1
?1;p
WRITE(P1:4)
?1;p
READ(P1)
?1;p
E 24 L-FAIL
?r3;p
if MAXA<P1 then MAXA:=P1
?13;p
READ(P1)
?.up<@for>;p
for I1:=1 to M do
     begin
     READ(P1);
     WRITE(P1:4);
     COEF[I1]:=P1;
     if MAXA<P1 then MAXA:=P1
     end
? % go to the top of the enclosing procedure
?.proc
?s1 p
procedure INITIAL
?p4

procedure INITIAL;
     var  #;
     begin
     MAXA:=0;
     MAXB:=0;
     INIMAXSUM:=0;
     READ(M,N);
     DIM:=#;
     WRITELN('M=',M:2,' N=',N:2);
     WRITELN('A[I]:');
     for I1:=# do ...;
     WRITELN;
     WRITELN('B[J]:');
     for J1:=# do ...;
     INIMAXSUM:=#;
     WRITELN;
     WRITELN;
     LASTSOL:=#;
     CDIM:=#;
     for I1:=# do #
     end
? % find something that I will enter from the terminal
?f &
[stat]
:writeln($e1);
```

```
 ?p
 WRITELN('M=',M:2,' N=',N:2)
 ?@el p
 'M='
 ? % label this statement and declare the label
 ?.label
 GIVE THE LABEL
 [INTCST]:1;
 ?p
 1: WRITELN('M=',M:2,' N=',N:2)
 ? % the statement is now labeled. go to the top of this procedure
 ? % and print it
 ?.proc;p3

 procedure INITIAL;
      ...;...;
      begin
      #;#;#;#;#;#;#;#;#;#;#;#;#;#;#;#;#;#;#;#;#
      end
?p4

 procedure INITIAL;
      label
           1;
      var  #;
      begin
      MAXA:=0;
      MAXB:=0;
      INIMAXSUM:=0;
      READ(M,N);
      DIM:=#;
      1: #;
      WRITELN('A[I]:');
      for I1:=# do ...;
      WRITELN;
      WRITELN('B[J]:');
      for J1:=# do ...;
      INIMAXSUM:=#;
      WRITELN;
      WRITELN;
      LASTSOL:=#;
      CDIM:=#;
      for I1:=# do #
      end
? % the label declaration part was created by .label
?.body;s3;p
INIMAXSUM:=0
?.label
GIVE THE LABEL
[INTCST]:1;
GIVE ANOTHER ONE, NOT IN THIS LIST
label
     1
[INTCST]:2;
?p
```

```
2: INIMAXSUM:=0
?.lab;p
label
     1,2
?.body;s5;p
DIM:=M+N
?.label
GIVE THE LABEL
[INTCST]:2;
GIVE ANOTHER ONE, NOT IN THIS LIST
label
     1,2
[INTCST]:3;
?p
3: DIM:=M+N
? % some verifications are done
?
?.dectype
type name
[IDENT]:table
ALREADY EXISTS, give another name
[IDENT]:ttx
[TYP]:array[1..35] of integer;
?.typ;p
type TTX     =array[#]of INTEGER
?.proc;p4

procedure INITIAL;
     label
          1,2,3;
     type #;
     var  #;
     begin
     MAXA:=0;
     MAXB:=0;
     2: #;
     READ(M,N);
     3: #;
     1: #;
     WRITELN('A[I]:');
     for I1:=# do ...;
     WRITELN;
     WRITELN('B[J]:');
     for J1:=# do ...;
     INIMAXSUM:=#;
     WRITELN;
     WRITELN;
     LASTSOL:=#;
     CDIM:=#;
     for I1:=# do #
     end
?s2;  % go on the declaration parts
?p
label
     1,2,3;
```

```
type TTX     =array ... of INTEGER;
var  I1,J1,P1:INTEGER
?
?
?
?
?
?
?
?
?
?
?
?
? %  Now load another program without loosing this one.
? % Then, push-down pointer @k.
?@k pd
?.load  %the program to be loaded is already in polish form
FILE NAME:defigure
?p*

program DEFIGURE;
     var  I,J,K:INTEGER;

     procedure INUTILE;
          begin
               begin

               end
          end;

     begin
     J:=5;
     K:=6;;
          begin
          ;;
          end;
     INUTILE;
     for I:=0 to 10 do
          begin
          WRITELN(I+J+K);
          end;
     FIN;;;;;;
     end.
? % clean up this dirty program
?.clean
?p*

program DEFIGURE;
     var  I,J,K:INTEGER;

     procedure INUTILE;
          begin

          end;
```

```
     begin
     J:=5;
     K:=6;
     INUTILE;
     for I:=0 to 10 do WRITELN(I+J+K);
     FIN
     end.
? % I like it better this way.
? % Now put some comments
?.comprocs
?p*

program DEFIGURE;
     var  I,J,K:INTEGER;

     procedure INUTILE;
          begin

          end (*INUTILE*);

     begin
     J:=5;
     K:=6;
     INUTILE;
     for I:=0 to 10 do WRITELN(I+J+K);
     FIN
     end.(*DEFIGURE*)
?.combods
?p*

program DEFIGURE;
     var  I,J,K:INTEGER;

     procedure INUTILE;
          begin

          end (*INUTILE*);

(*body of DEFIGURE*)
     begin
     J:=5;
     K:=6;
     INUTILE;
     for I:=0 to 10 do WRITELN(I+J+K);
     FIN
     end.(*DEFIGURE*)
?.save
FILE NAME:newdefigure
NEWDEFIGURE.po  FILE OVERWRITTEN
? % go back on the preceding program
?@k pu  % pop-up the current pointer
?p
label
     1,2,3;
```

```
type TTX      =array ... of INTEGER;
var  I1,J1,P1:INTEGER
? % We are where we were on that program
?.end
EXIT LEV.1-PASCAL
```

Appendix B -The Abstract Syntax of Pascal

PHYLA TABLE

every	::	IDENT	META	INTCST	ALFACST	CHARCST	NIL	HEXCST
		REALCST	METASYM	PASCAL	FORWARD	LINE	SETOF	NOT
		UPLUS	UMINUS	UNREF	HEXA	DEF	FUNCPAR	VARPAR
		REF	PACKED	FILE	SET	PROCPAR	GOTO	EQLC
		EQLT	EQLV	INTERV	EQL	LSS	GTR	NEQ
		LEQ	GEQ	IN	INTDIV	MOD	DIV	MULT
		PLUS	MINUS	OR	AND	INDEX	DOT	FORMAT
		RANGE	CASERC	ARRAY	DECTAG	COLRC	DECL	UPSTEP
		DWNSTEP	REPEAT	ASS	CALL	CASE	COLON	WHILE
		WITH	LABSTAT	TIMES	PROG	PROC	LELEM	LABEL
		LEXP	LIDENT	LCST	LDEFID	VAR	LTYP	LFIELD
		LCOLRC	LSTAT	LCOLON	LVARBL	LPARAM	LVAL	CONST
		TYPE	VALUE	LZONE	LLINE	LCHAR	FOR	IF
		FUNC	BLOCK					
cst	::	IDENT	META	INTCST	ALFACST	CHARCST	NIL	HEXCST
		REALCST						
varbl	::	IDENT	META	UNREF	INDEX	DOT		
casetg	::	IDENT	META	DECTAG				
exp	::	IDENT	META	INTCST	ALFACST	CHARCST	NIL	HEXCST
		REALCST	SETOF	NOT	UPLUS	UMINUS	UNREF	HEXA
		EQL	LSS	GTR	NEQ	LEQ	GEQ	IN
		INTDIV	MOD	DIV	MULT	PLUS	MINUS	OR
		AND	INDEX	DOT	FORMAT	CALL		

```
elem      ::  IDENT    META     INTCST   ALFACST  CHARCST  NIL      HEXCST
              REALCST  SETOF    NOT      UPLUS    UMINUS   UNREF    INTERV
              EQL      LSS      GTR      NEQ      LEQ      GEQ      IN
              INTDIV   MOD      DIV      MULT     PLUS     MINUS    OR
              AND      INDEX    DOT      CALL

stat      ::  META     GOTO     REPEAT   ASS      CALL     CASE     WHILE
              WITH     LABSTAT  LSTAT    FOR      IF

step      ::  META     UPSTEP   DWNSTEP

param     ::  META     PROCPAR  DECL

local     ::  META     FUNCPAR  VARPAR   LDEFID

spltyp    ::  IDENT    META     RANGE    LIDENT

typ       ::  IDENT    META     REF      PACKED   FILE     SET      RANGE
              ARRAY    LIDENT   VAR      LFIELD

defid     ::  IDENT    META     DEF

body      ::  META     METASYM  PASCAL   FORWARD  LSTAT

field     ::  META     CASERC   DECL

val       ::  IDENT    META     INTCST   ALFACST  CHARCST  NIL      HEXCST
              REALCST  TIMES

valu      ::  IDENT    META     INTCST   ALFACST  CHARCST  NIL      HEXCST
              REALCST  LVAL

zone      ::  META     LABEL    VAR      CONST    TYPE     VALUE    BLOCK

title     ::  META     PROG     PROC     FUNC
```

STRUCTURE OF OPERATORS

TERMINAL OPERATORS

IDENT META INTCST ALFACST CHARCST NIL HEXCST REALCST
METASYM PASCAL FORWARD LINE

OTHER OPERATORS

```
SETOF    -->  lelem
NOT      -->  exp
UPLUS    -->  exp
UMINUS   -->  exp
UNREF    -->  varbl
HEXA     -->  exp
```

```
DEF       -->  ident
FUNCPAR   -->  ldefid
VARPAR    -->  ldefid
REF       -->  ident
PACKED    -->  typ
FILE      -->  typ
SET       -->  spltyp
PROCPAR   -->  lident
GOTO      -->  intcst
EQLC      -->  ident    cst
EQLT      -->  ident    typ
EQLV      -->  ident    valu
INTERV    -->  exp      exp
EQL       -->  exp      exp
LSS       -->  exp      exp
GTR       -->  exp      exp
NEQ       -->  exp      exp
LEQ       -->  exp      exp
GEQ       -->  exp      exp
IN        -->  exp      exp
INTDIV    -->  exp      exp
MOD       -->  exp      exp
DIV       -->  exp      exp
MULT      -->  exp      exp
PLUS      -->  exp      exp
MINUS     -->  exp      exp
OR        -->  exp      exp
AND       -->  exp      exp
INDEX     -->  varbl    lexp
DOT       -->  varbl    ident
FORMAT    -->  exp      exp
RANGE     -->  cst      cst
CASERC    -->  casetg   lcolrc
ARRAY     -->  ltyp     typ
DECTAG    -->  ident    ident
COLRC     -->  lcst     lfield
DECL      -->  local    typ
UPSTEP    -->  exp      exp
DWNSTEP   -->  exp      exp
REPEAT    -->  lstat    exp
ASS       -->  varbl    exp
CALL      -->  ident    lexp
CASE      -->  exp      lcolon
COLON     -->  lcst     stat
WHILE     -->  exp      stat
WITH      -->  lvarbl   stat
LABSTAT   -->  intcst   stat
TIMES     -->  intcst   cst
PROG      -->  defid    lident
PROC      -->  defid    lparam
LELEM     -->  elem     ...
LABEL     -->  intcst   ...
LEXP      -->  exp      ...
LIDENT    -->  ident    ...
LCST      -->  cst      ...
```

```
LDEFID    -->  defid     ...
VAR       -->  decl      ...
LTYP      -->  spltyp    ...
LFIELD    -->  field     ...
LCOLRC    -->  colrc     ...
LSTAT     -->  stat      ...
LCOLON    -->  colon     ...
LVARBL    -->  varbl     ...
LPARAM    -->  param     ...
LVAL      -->  val       ...
CONST     -->  eqlc      ...
TYPE      -->  eqlt      ...
VALUE     -->  eqlv      ...
LZONE     -->  zone      ...
LLINE     -->  line      ...
LCHAR     -->  charcst   ...
FOR       -->  ident     step     stat
IF        -->  exp       stat     stat
FUNC      -->  defid     lparam   ident
BLOCK     -->  title     lzone    body
```

A brief description of the annotation mecanism in Mentor-V5

Bertrand Mélèse

November 1984

Mentor-V5 supports the annotation mechanism that has been described in recent publications about Mentor:

-- V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse, E. Morcos
Outline of a tool for document manipulation,
IFIP, september 1983, Paris

-- V. Donzeau-Gouge, B. Lang, B. Mélèse,
Practical Applications of a Syntax Directed Program Manipulation Environment,
Proceedings of the 7th Int. Conf. on Soft. Eng., Orlando, Florida, March 1984

-- V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse,
Documents Structure and Modularity in Mentor,
ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments, Pittsburgh, April 1984
SIGPLAN Notices, Vol 19, No 5, May 1984.

In this info file you will find only how to use the annotation mechanism, that is, the commands Mentor-V5 provides to deal with annotations.

1-- The commands to defined and modify annotation frames.

```
.defframe

.newframe
.setprop
.resetprop
.setprefix
.setpostfix
```

The command .defframe is the internal Mentor-V5 command to deal with annotation frames. Other commands form a little user interface around .defframe and are implemented in Mentol using .defframe.

An annotation frame is the definition of a given class of annotations. An annotation is a structure (an abstract syntax tree) hooked somewhere in another structure. The behaviour of a given annotation is defined by properties of the annotation frame it belongs to.

Many annotations belonging to the same annotation frame may be attached to

annotation.i

different node of a tree.

Characteristics of an annotation frame are:

-- The name of the frame.

-- The annotated language, that is the language that is allowed to receive annotations belonging to that frame.

-- The annotation language, that is the language in which the annotation is written.

-- The properties of the frame. Existing properties of annotation frames that a user can modify in Mentor-V5 are: PRE, POST, VISIBLE. Other properties exist (SAVE NAMED COPY) but are not yet effective in Mentor-V5 (not implemented).

   Annotations belonging to a frame that has the property VISIBLE are visible, that is are shown by the unparser if permitted by other properties and by the values of the toggle COMMENT and the dial COMLEVEL (see section 5 below).

   Annotation belonging to a frame that has the property PRE are unparsed before the node they are hooked to.

   Annotation belonging to a frame that has the property POST are unparsed after the node they are hooked to.

   A frame CAN have simultaneously the properties PRE and POST. In this case, annotations belonging to it are unparsed twice. If a frame has none of these properties, it is not unparsed even if it has the property VISIBLE.

1.1-- .defframe

When called without parameters, .defframe asks questions to the user.

   The first question is about the annotated language.

   The second question is the command to execute.

   The third is the Frame reference name.

   The fourth question depend of the command given as answer to the second question.

The command .defframe can also be called with up to four parameters. Each parameter corresponds to the answer to one of the previous questions. If less than four parameters are given, missing information is requested interactively from the user.

The first parameter (as well as the answer to the first question when the first parameter is omitted) is the annotated language. If the desired annotated language is the current language, the first parameter can be

"curlang instead of a language name. (see multi-lang.i to know what the current language is).

The second parameter (as well as the answer to the second question when the second parameter is omitted) is the command to be done on the annotation frame. Possible commands are: new, set, reset, addname.

The command "new" is to create a new annotation frame.
The command "set" is to set a property of that frame
The command "reset" is to reset a property.
The command "addname" is to associate a new name to the same annotation frame.

The third parameter is the frame name to be created if the command was "new" or to be modified if the command was "set" "reset" or "addname". When the command is "set" or "reset", this parameter can be "curframe. In this case, the command applies to the frame used in the previous call of .defframe.

The fourth parameter depend on the command (i.e. the second parameter):

If the command was "new", the fourth parameter is the language in which the annotation belonging to this new frame will be written.

If the command was "set" or "reset" the fourth parameter is the property to set or reset.

Properties that can be set or reset are currently: PRE, POST, VISIBLE.

If the command was "addname", the fourth parameter is the name to be added to this annotation frame. This added name can then be used instead of the frame reference name in all commands involving the frame. Names added to frames ARE NOT SAVED in polish files. The only frame name saved in polish files is the frame reference name. Added names are temporary names existing only during the Mentor-V5 session in which they are explicitely declared using the ADDNAME command of .defframe.

*** REMARK 1: Frame reference names versus names added to frames.

Because of the dynamic definition of frames, it may appear that a polish file corresponding to a program on which several people are working, contains a lot of frames and annotations belonging to them. Among these frames, some may be known by a programmer and unknown by other programmers. Some frames and annotations can also have been created by commands of a programming environment to record some computed information about this program, information that is to be used by other commands of this environment. To limit the risk of conflicts between frame names, it is recommanded to choose long reference names for frames. Then, during a Mentor-V5 session, it is possible to temporarily add shorter names to some frames to make it quicker to reference them.

*** END REMARK 1.

Examples:

.defframe all parameters will be requested interactively

.defframe<"curlang,"new>
The annotated language is the current one the command is "new", other parameters (frame name and frame language) are requested interactively.

.defframe<"pascal,"set>
The annotated language is Pascal, the command is "set", other parameters (frame name and Property to set) are requested interactively.

.defframe<"rapport,"new,"foo,"pascal>
The annotated language is Rapport, the command is "new", the new frame name is foo and this frame is in language Pascal. No parameters are requested interactively.

.defframe<"rapport,"set,"curframe,"pre>
The annotated language is Rapport, the command is "set", the frame name is the frame name considered by the previous call of .defframe and the property to set is PRE.

In all these examples, parameters of defframe are given as immediate strings (starting with a " symbol). They can also be mentol variables denoting atomic trees containing the relevant information. This is a standard Mentol facility.

1.2-- .newframe, .setprop, .resetprop, .setprefix, .setpostfix

These commands are implemented using .defframe. They all act on frames defined for the CURRENT language (see multi-lang.i).

1.2.1-- .newframe

To create a new frame in the CURRENT language. Asks for the frame name and the frame language name. Its implementation in terms of .defframe is:

```
.defframe<"curlang,"new>
```

1.2.2-- .setprop

To set a property of a frame defined for the current language. Asks for the frame name and the property to set. Its implementation in terms of .defframe is:

```
.defframe<"curlang,"set>
```

1.2.3-- .resetprop

To reset a property of a frame defined for the current language. Asks for the frame name and the property to reset. Its implementation in terms of .defframe is:

```
.defframe<"curlang,"reset>
```

1.2.4-- .setprefix

To make a frame to have the property PRE and to NOT have the property POST. Asks for the frame name that must be a frame defined for the current language. Its implementation in terms of .defframe is made of two calls to .defframe:

```
.defframe<"curlang,"set,,"pre>
.defframe<"curlang,"reset,"curframe,"post>
```

Note that, in the first call the third parameter is omitted. Since the fourth parameter is present the omission of the third one is made by leaving an empty parameter position. This is a standard facility provided by the Mentol language.

In the second call of defframe the name "curframe is used to apply the reset property to the same frame as in the previous call of defframe.

1.2.5-- .setpostfix

Is the obvious reverse of .setprefix. Its implementation is:

```
.defframe<"curlang,"reset,,"pre>;
.defframe<"curlang,"set,"curframe,"post>
```

Note: all these procedures fail, in the usual way of Mentol procedures, when something goes wrong.

2-- Help command on frames.

The new keyword "frames" of the help system of Mentor-V5 lists all existing annotation frames associated with the current language.

To get that list, first call the help system by issuing the .help command. Then, when the prompt symbol becomes - type "frames".

Two predefined frames always exist in all languages: the frames named 0 and 1. They are kept for compatibility with previous versions of Mentor and thus correspond to old prefix and postfix comments provided by previous versions of Mentor and accessed by the Mentol commands b0 and b1. (see section 3 below). The language of frames 0 and 1 is initialized to be the language on which they are defined. In Mentor-V5 these frames should be used only to receive text comments but this is not enforced.

The frame 0 has the properties PRE and VISIBLE while the frame 1 has properties POST and VISIBLE.

This frames can be modified using the .defframe command exactly for as user

defined frames but this is not encouraged.

When listing the existing frames you can see the other (not yet implemented) properties associated with frames: COPY SAVE NAMED. It is NOT SAFE to change these unimplemented properties (sounds strange !!).

3-- The mentol commands to interactively deal with annotations.

Annotations ARE normal Mentor-V5 trees: all Mentol commands apply to them exactly as they apply to trees that are not annotations.

However, two special moving commands are needed: the first is to go from a node onto one of its annotations, and the second is to go back from an annotation to the annotated node.

3.1-- To go from a node to one of its annotations.

Let's first take an example:

Suppose @foo be a Mentol variable denoting a Pascal if statement:

The address	@foo s1	denotes its first son (the expression)
The address	@foo s2	denotes its second son (the "then" clause)
The address	@foo s3	denotes its third son (the "else" clause)
The address	@foo u	denotes its father if exists

etc ... (see mentor.gi.i, mentor.demo.i and mentor.input.i)

The new addressing command for annotations is ^name where name is the name of an existing annotation frame of the language the tree belongs to.

The address @foo ^myframe denotes the annotation belonging to the frame myframe hooked to the tree denoted by @foo.

3.2-- To go back from an annotation to the annotated tree.

The command is: @foo u0

It works only when @foo denote the ROOT of the annotation tree.

3.3-- Compatibility with annotation frames 0 and 1.

Remember that, in previous versions the commands b0 and b1 were used to go on the prefix and postfix comments of a tree. These prefix and postfix comments are now the predefined annotations 0 and 1. Thus, they are accessed by the commands ^0 and ^1 respectively. The commands b0 and b1 are kept for compatibility.

In Mentor-V5

^0 is equivalent to b0

and ^1 is equivalent to b1.

4-- Multi-language polish files

Annotations are of course recorded in Mentor-V5 polish files. When loading a polish file that contains annotations, these annotations are rebuilt and their languages are automatically loaded when needed.

Loading a multi-language polish files makes the session multi-language automatically. When Mentor-V5 has to decide itself to load a new language to be able to create some annotations found in the file, it signals this fact by writing a message of the form:

"Polish file is in ADA while current language is PASCAL"

If you do not want to get this message, you have to reset the toggle named monolang using the predefined command .reset. the command to do is

```
.reset<"monolang>
```

or

```
.reset
```

and answer monolang to the question. (see toggs-dials.i).

5-- Useful toggles when using the annotation mechanism.

5.1-- monolang

To cancel messages about languages found in multi-language polish files

5.2-- comment

When this toggle is reset all annotations become invisible without touching the VISIBLE property of frames. When setting again this toggle annotations belonging to frames that have the property VISIBLE become visible again.

Resetting this toggle also affects the copy of tree (i.e. the Mentol command "=". See mentor.gi.i). Annotations are not copied when the toggle comment is reset.

5.3-- comlevel

It is a dial. Its value is the maximum number of visible levels of commenting. Its default value is large. Giving to this dial the value 0 is equivalent to resetting the toggle comment. In all cases, the number of visible levels of commenting indicated, for a given language, in the Metal program is tested before the dial comlevel. This means that the number of levels unparsed is taken to be the minimum of these two values.

6-- The command .sysannot

This command is the internal command of Mentor-V5 to deal with annotations

(not annotation frames !!).

This command is usable interactively but is mainly intended for performing actions on annotations from within Mentol programs. This command is never needed in interactive use of Mentor-V5. It can always be replaced the Mentol primitive addressing commands on annotations described in the next section. This commands becomes useful only when computed annotation commands are needed to implement multi-language environments as it is the case in the Rapport environment.

Examples of use of the command .sysannot can be found in the Mentol files RAPCREER.tol and RAPEDITOR.tol that are parts of the Rapport environment loaded by Mentor-V5 when the supplied user name is "rapuser".
(see mentor.gi.i).

This command takes up to five parameters. In interactive use, the user is requested for parameters that are not specified in the call.

The first parameter is an action. Existing actions are:

goto create change

The second parameter is the Mentol variable on which the command will apply. If the second parameter is missing the current pointer K is used.

The third parameter is the name of the annotation frame. It can be an immediate string ("foo) or a Mentol variable. If not specified, a randomly chosen annotation existing on the sub-tree denoted by the second parameter is used. In the current implementation the chosen annotation actually is the the MOST recently created annotation of that tree. User written Mentol commands should not rely on this: it is likely to change in future versions.

The fourth parameter is the annotation language name. It can be an immediate string or a Mentol variable. If not specified, no check is done on the annotation language.

The fifth parameter is needed only when the action is "create or "change. In this case, it is a Mentol variable denoting the tree to be hooked as annotation of the tree denoted by the second parameter.

examples:

.sysannot<"goto>
moves the current pointer K to the first annotation of the tree denoted by K.

.sysannot<"goto,@foo,"env,"pascal>
moves the current pointer K to the annotation in frame env annotating the tree denoted by K provided this annotation is in Pascal.

.sysannot<"goto,@foo,"env,>
moves the current pointer K to the annotation in frame env of the tree denoted by @foo whatever the language of this annotation may be.

.sysannot<"change,,"env,"pascal,@foo>

changes the annotation in frame env of the tree denoted by the current pointer K (because the second parameter is not specified) by the tree denoted by @foo, provided the annotation language of env is Pascal and the language of the tree denoted by @foo is a Pascal tree.

The procedure .sysannot fails, in the usual way of Mentol commands, when something goes wrong.

multi–lang.i

The Multi-langage facility in

Mentor-V5

Bertrand Mélèse

November 1984

Mentor-V5 is "Multi-language". This means that several languages can be manipulated at the same time during the same Mentor-V5 session. Using the annotation mechanism, different languages can also be mixed in the same Mentor-V5 object (see annotation.i).

Various aspects of the multi-language facility of Mentor-V5 have been described in recent publications about Mentor:

-- V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse, E. Morcos,
Outline of a tool for document manipulation,
IFIP, september 1983, Paris

-- V. Donzeau-Gouge, B. Lang, B. Mélèse,
Practical Applications of a Syntax Directed Program Manipulation Environment,
Proceedings of the 7th Int. Conf. on Soft. Eng., Orlando, Florida, March 1984

-- V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse,
Documents Structure and Modularity in Mentor,
ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments, Pittsburgh, April 1984
SIGPLAN Notices, Vol 19, No 5, May 1984.

When entering the Mentor-V5 system, you are first in a mono-language session: The current language (which, up to now, was the only one) is the language you have chose for when Mentor-V5 requested a language name.

If you did enter Mentor-V5 using a prelude file the current language is the language specified in that file.

1-- Prelude files.

The role of prelude files is to provide the user with a standard beginning of sessions that indicates the language used and loads an environment of Mentol commands.

A Mentor-V5 prelude file has the extention .pre. As all Mentor-V5 files, the name preceeding the extension must be in upper cases.

Example: MYNAME.pre

Predefined prelude files are loaded when answering "pascaluser", "metaluser", "rapuser" or "adauser" to the user name question of Mentor-V5. They make the current language being Pascal, Metal, Rapport or Ada respectively and they load the corresponding standard environment.

You can of course copy any of these prelude files in your own space (or make links on them) to give them other names.

For example, the predefined prelude file PASCALUSER.pre contains:

```
%PASCAL                    The desired language is Pascal
.XIN<@PASCALSTD>           Loads the standard Pascal environment
.xin<@pasvideo>            Loads the interface between Pascal and
                           text editors
.xin<@passubstr>           Library for string substitution
.xin<@bwsearch>            library for backward searching
.xin<@annottool>           library for annotation handling
.p                         clears the screen (see mentor.video.i)
#4                         sets the holophrasting level to 4
.emacs                     sets emacs as default text editor
.xin                       returns control to user's terminal
```

A line of the form .xin<@foo>, loads the Mentol file FOO.tol that must be in a directory whose name is given in the shell variable called MENTOR (see mentor.gi.i). One can add any command in his/her prelude file.

The predefined prelude file RAPUSER.pre contains:

```
%RAPPORT                   The desired language is RAPPORT
.xin<@rapportstd>          loads the standard Rapport library
.xin<@annottool>           loads the annotation library
.reset<"monolang>          resets the toggle name monolang
                                 (see annotation.i)
.emacs                     sets emacs as current text editor
.p                         clears the screen (see mentor.video.i)
.xin                       returns control to user's terminal
```

A possible prelude file to start a Mentor-V5 session with a newly user defined language call "userlang" is:

```
%USERLANG
.xin<@commonenv>           loads the language independant mentol
                           environment COMMONENV.tol.
.xin<@annottool>           loads the annotation library
.p
.xin
```

The file COMMONENV.tol in the directory .../mentor/mentol contains the language independant Mentol commands (see mentorkit.i).

Actually, the command .language explained in the next section can be called in a prelude file, to initialize a multi-language session.

2-- The command .language.

Once a (mono-language) Mentor-V5 session is started you can make it multi-language by issuing the .language command.

When called without parameter, this command will ask for a language name. The name of the desired language can also be passed as parameter to .language. In this case the call as the following syntax: .language<"langname> where langname is a language name or .language<@foo> where @foo is a Mentol variable denoting an atom whose value will be interpreted as a language name.

Examples: (1) immediate argument .language<"metal>
(2) Mentol variable as argument .language<@foo>
and @foo must denote an atom.

The command .language fails if the desired language does not exists.

The language is then loaded in the SAME session and becomes the CURRENT language of the session (see section 1.3 below). The session is now a multi-language session. In this session different Mentol variables can denote trees belonging to different languages.

3-- Current language and blip window

The blip window is the small window in the upper right corner of the screen. It contains three informations.

i) On the first line and in upper cases: The name of the current language.

ii) On the second line: The type of the current subtree (i.e. the name of the top operator of the tree displayed in enhanced font on the screen), and a ~ (tilda) sign followed by the name of the language of that tree.

Example: A possible blip window is

```
ADA
while~pascal
```

It means that the current language is ADA, but the current subtree is a Pascal while operator.

The CURRENT language is the language in which new default trees will be created, in particular to initialize newly used Mentol variables. Since the current language is not necessarily the same language than the language of the current tree (i.e. the tree that you see unparsed on your screen) one must take care of some strange possible situations:

Suppose for example that, in the situation stated by this blip window, one

issues the Mentol command

f @case

thinking that this command will go to the next Pascal case statement of the current Pascal program fragment: IT IS ACTUALLY WHAT THIS COMMAND DOES IN MONO-LANGUAGE SESSIONS !!!. Here, in our ADA-PASCAL session, with ADA as current language, this command does the following things:

(1) build a Mentol variable called @case denoting an (undefined) ADA structure. The Mentol variable @case is now known to Mentor-V5 as denoting something in Ada.

(2) Fails in trying to search an ADA structure in a Pascal program.

This command would of course have made what the user did expect if the current language had been Pascal.

Thus, the command to do there was:

f @case~pascal

to say that the wanted Mentol variable is the PREDEFINED @case schema of the Pascal abstract syntax. Another solution, is to change the current language before issuing the command. This is done by the .language command (the same as before): the .language command only changes the current language when the language given to it has already been loaded.

Thus, with the blip window

ADA
while~pascal

you can first call the command .language and answer pascal to the question. The blip window becomes

PASCAL
while~pascal

and the command

f @case

works properly as expected.

****** REMARK 1:

If you have a lot of command to make with a given language, it is more comfortable to have this language as current language.

***** END REMARK

Let suppose again that we are in the bad situation above: A Mentol variable called @case has been created denoting an ADA structure (the fact that the Ada structure was an undefined structure is not relevant here):

SWITCHING BACK TO PASCAL AS CURRENT LANGUAGE
WILL NOT MAKE @CASE DENOTE AGAIN THE PREDEFINED
CASE STATEMENT OF PASCAL !!!!!

This predefined case statement schema of Pascal can, from now on, be accessed only through its COMPLETE name which is @case~pascal. This is actually a disagrement in a lot of cases. To come back to the nice situation where @case denote the predefined schema of a Pascal case statement you have to redefined it by the Mentol command:

@case : @case~pascal

which means "assign to the Mentol variable @case the predefined @case schema of the Pascal abstract syntax". (':' in the standard assignment operator of Mentol: see mentor.gi.i).

***** REMARK 2:

Of course, the situation described above is independant of the fact that the Mentol command was f and the pattern to search was @case !! The important thing is that THE FIRST TIME A MENTOL VARIABLE IS INVOQUED, IT IS INITIALIZED ACCORDING TO THE CURRENT LANGUAGE AND THEN KEEPS ITS VALUE IN THAT LANGUAGE (unless it is reassigned later) EVEN WHEN THIS LANGUAGE IS NO MORE THE CURRENT ONE.

The complete name of the predefined operators of a language under Mentor-V5 is of the form

@opname~langname

In good situations it can be abbreviated to @opname.

Goods situations are: the current language is langname AND the Mentol variable @opname has not been already created when another language was the current language.

A consequence of that situation is: WHEN, IN YOUR MULTI-LANGUAGE SESSION SEVERAL LANGUAGES HAVE OPERATORS WITH THE SAME NAME, THEIR COMPLETE NAMES HAVE TO BE USED IN ALMOST ALL CASES.

We are aware that this behaviour is rather inconvenient. In future versions, we intend to deduce a default language from the context of each Mentol command.

***** END REMARK

4-- Current language and Parsing

Parsing (using the & command of Mentol) can be made in any language known in the session.

4.1-- When parsing in a context, Mentor-V5 always deduces the parser to call, by examinating the place where the tree built by this parsing will be put.

If the parsed language is the current one Mentor-V5 prompts only with the wanted phylum name as in previous versions.

If the parsed language is NOT the current one Mentor-V5 prompts with the language and the phylum. This is a little help offered for free. The current language is then sets by Mentor-V5 to be the language in which the last parsing did occur.

4.2-- When parsing occurs out of context, the parsed language will be the current one and, as in previous versions, the prompt is [and the user has to supply himself the name of a phylum. In Mentor-V5, this situation (having to indicate a phylum name) can always be avoided by using the .menu command (see beginners.i).

***** REMARK 3:

When working with more than one language it is difficult even for experts to remember the abstract syntaxes of all the languages. The .menu command is there to help, use it !!!!!!

***** END REMARK

Final remark:

In the directory .../mentor/test/multi, you will find polish files in various languages. The file MULTI.po is a multi-language document in Rapport be manipulated under Mentor-V5. The best way to try that is to call mentor, aswer "rapuser" to the "user name" question and then load this file using the command .load and answering "multi" as file name.
A good idea is to look at the file rap.procs.i before trying that to take advantage of the user interface the rapport environment provides.

beginners.i

How to use menu mode: .menu

Valerie Migot
November 1984

This is a new input mode available in the Version 5 of Mentor. It is a menu driven input mode. This mode is well suited for beginning users since using this mode is a good way to learn the abstract syntax of a language.

It will also be useful to experienced users in multi-language sessions: they can call .menu on a subtree, exit (see the '!' command), create a frame in another language, call .menu on this annotation, and come back (with the '$' Mentol command) to the previous call of .menu.

An interesting point is that the menu driven input mode can be temporarily escaped at any point (see the '&' command) by experienced users who know exactly what they want to enter at the current point.

The menu mode is language independant and can then be used even with user defined languages. It takes the language of the tree on which it is called.

The menu is calculated both from the abstract syntax of the language and from the current position in the program. It contains the names of the operators that can be put at the current place according to the abstract syntax of the current language. The user answers the name of the operator he wants and the predefined schema of this operator replaces the meta-variable (when creating a new piece of code) or the old piece of code (in case of modication).

The menu mode is called by issuing the .menu command. After the .menu command has been called, the screen contains three windows:

- the dialog window at the top of the screen contains the menu and user's answers
- the text window contains the text of the program
- the help window at the bottom of the screen contains help messages which depend on the current position in the program and on the proposed menu.

In menu mode, one can edit or modify a program.

When the user calls .menu on a yet undefined program (**undef), the first menu displays all the operators of the abstract syntax of the language. The user must answer with the complete name of the operator he wants to start with. To choose in subsequent menus, the user can enter only the first distinctive letters of the operator's name he wants. If the given name is wrong (i.e unknown in the current menu, or not distinctive enougth and multiple possibilities remains), the user is requested again.

When the current position is an atomic operator, the user has to enter the value of the atom (identifier, ...).
To terminate the input, there are two possible cases depending on the current context:

-- in "good" cases, the answer will be directly interpreted after a carriage return has been typed in.

-- in other cases, after the carriage return, the menu mode prompts again with a ':' on the next line. In this case, the user has to type a '.' to terminate the input.

These two possibilities should not get the user into trouble because he does not have to guess in what case his current input falls: he just has to enter a '.' when he is prompted on a new line.

*** Warning

The use of the menu mode in Pascal is a little different, because the Pascal parser is wired in Mentor instead of being generated using yacc and lex.

The first difference is about the way to indicate the end of input when the menu mode promts again on the next line: In pascal, instead of a '.' as in other languages, the Pascal user has to enter a semicolon (;).

The second difference concerns the first menu when starting from an empty program. In Pascal, the first menu (when one calls .menu on an undefined tree) is empty and the user has to answer an operator's name without any help !!!!. To build a Pascal program, one has to answer 'block' (not 'program') to this first question of the menu mode. More help will be given at this point on subsequent versions.

Here are hints on operator's name in Pascal:

-block	to build a program, procedure of function
-stat	to build any pascal statement
-zone	to build any pascal declaration parts

At any moment, the user can answer '?' to a menu to get more information. He can also answer '?' followed by a name contained in the menu to get specific information about this operator's name.
He can answer '&' if he wants to use direct parsing at the current point (for expert beginners only).
He can exit temporarily the menu mode with the '!' key. Then, to come back into the menu mode at the previous current position, the Mentol command '$' has to be used. (Note for experts : '!' is the .user Mentol command).

The edition stops when the current subtree, on which .menu was called, has been completed. The '.' key allows to leave definitively the menu mode, except during list processing (in this case, the '.' stops the insertion in the list, while the 'm' or 'M' key insert a new element in the list).

Available keys in menu mode:

?	help
.	stop
&	direct parsing
!	exit from menu mode ($ to come back)
a name	in the menu

During insertion in a list:

M or m	to insert in the list
.	stop the insertion

Known bugs:

After the '!' command, it is fatal to destroy the tree containing the current sub-tree: this will crash the system when coming back to menu mode!
(this is a difficult problem: what should happen when you saw the branch on which you were seated.)

mentor.wr_pr.i

WRITING MENTOR PROCEDURES

Bertrand Mélèse
(June 1980 - July 1984)

The predefined procedure '.def' is used to define Mentol procedures.

The procedure '.def' takes 2 arguments:

1- The header of the defined procedure, composed of the procedure name, followed by the list of formal parameters between "<" and ">". Parameters can be either pointers (i.e. Mentol variables denoting trees) or procedures. It is possible to declare local pointers or procedures.

2- The body of the defined procedure which is a single command.

```
Example:  .def<.foo<.f,@p1,@p2>,(.f<@p1>;?.foo<,@p1 s>,$)>
                 (....header....) (.........body...........)
```

Note that when the body is a list of commands it has to be made into a single one with "(...)" .

Syntax of definition:

```
.def<.procedure_name<arg1,arg2,...,argn>,(instruction_list)>
```

Remember that lower cases and upper cases are equivalent under Mentor, thus they are equivalent in the procedure definitions.

Examples:

```
.def<.print<@x>,(@x p*)>
```
The procedure .print will just print his parameter.

```
.def<.print2<@x,@y>,(.eqval<@x,@y>;?@x p,(@y=@x;@y p))>
```

The procedure .print2 tests its two parameters for equality. If they are equal, the first one is printed. If they are different, the second one receives a copy of the first and then it is printed. See the description of .eqval in the file mentor.procs.i.

One can see on this example the syntax of the conditional statement of Mentol. In fact the syntax used to print Mentor-procedures is EXACTLY the same as the syntax of commands as described in mentor.gi.i.

During a Mentor session you can define procedures in the same way as

indicated in the examples, but, the definition must not be cut by the character <CR> (carriage return). However the procedures defined this way will be lost when you will stop the current Mentor session. We shall see below how to create and use files containing Mentol procedure definitions.

I. How to create a file of Mentol procedures:

Let's suppose that you want to define some Mentol procedures, put them in a file, and then use them under Mentor.

1. The name of the file must be suffixed by '.tol', because the procedures will be written in the language mentol . Thus, your file of procedures can be called YOURFILE.tol (the first component of the name must be in upper cases).

2. Format of the file :

-- The file must begin by a blank line.

-- The file must end by a call to the predefined Mentor procedure '.XIN'. This procedure indicates to Mentor that the end of the file has been reached and that control must be switched back to the previous input device (Your terminal in general).

-- After the first line, which is a blank line, you have to write the two following lines:

```
.rec
editeur
```

These two lines indicate to Mentor that it will now find Mentol procedures in your file. The command .rec calls Mentor recursively. The name of the language manipulated in this recursive call will be "editeur" which is equivalent to "mentol".

We will see later that such a file may contain other things than Mentol procedures.

Somewhere in the file must appear the line

```
.end
```

to indicate that there are no more procedure definitions in this file. This call to .end in facts terminates the recursive call initiated by the previous call to .rec.

Thus, your file looks now like this:

line number	contents
1	
2	.rec

```
3                editeur
4                .end
5                .xin
```

Your Mentol procedures will be defined between lines 3 and 4.

3. Format of a procedure definition:

The definition of a Mentol procedure begins with the line:

```
:&
```

Then, there is a definition of the form:

```
.def<.procedure_name<arg1,...,argN>,
    (procedure_body)>
```

Note that '.procedure_name<arg1,...,argN>' and (procedure_body) are the parameters of the predefined procedure '.def'. Thus they are enclosed with the characters < and > and are separated by a comma.

After this definition you have to put another blank line, and then a line containning the magic word 'ss2;.ldef' .

The file YOURFILE.tol is now something like:

```
line number          contents

    1
    2                .rec
    3                editeur
    4                :&
    5                .def<.procedure_name<arg1,...,argN>,
    6                    (procedure_body)>
    7
    8                ss2;.ldef
    9                .end
   10                .xin
```

The actual definition may, of course, be as long as you want it to be. To define a second procedure in this file you just have to repeat a sequence of lines like the lines 4 to 8 beside the line number 8.

Inside the definition, that is, between the sign '<' following '.def', and the sign '>' that ends the definition, you can format the mentol code as you want with only one restriction:

A LINE MUST NEVER BE TERMINATED BY A DIGIT

When Mentor finds errors in your procedures, first check if this condition is respected in your file (This is in fact a known bug of Mentor).

Example:

This is an example of a file containing Mentol procedures. Line numbers are not in the file. We put them here to reference easily the lines in the explanation that follows. Note that lower and upper cases are used ramdomly. The strings enclosed with two '%' are comments. Comments may appear everywhere, but a line of comments is not the same as a blank line . That is, you cannot replace the lines that must be blank (first line and line preceding 'ss2;.ldef'), by comments. A comment ends at the end of the line if there is no '%' before the end of the line.

```
 1
 2    .rec
 3    editeur
 4    :&
 5    .DEF<.UP<@UPPAR>,(.IS<@UPPAR>;?$,(U;?,$-2))*>
 6
 7    ss2;.ldef
 8    :&
 9    .DEF<.APROC,.UP<@BLOCK>>
10
11    ss2;.ldef
12    :&
13    .DEF<.LEFTMS,((S/;$)*;.IS<@IDENT>) >
14
15    ss2;.ldef
16    :&
17    .DEF<.FPROC<.level,@LOC,@START>,
18    % @loc and @start are LOCAL variables, NOT parameters%
19    % .level is the starting level of the search
20    (@START:K;
21     .level;
22     @LOC=@LIDENT;
23     @mesapp s2 p;
24     @LOC S1 C &;@LOC S1;
25            .FORALL<@BLOCK,
26                   (.LEFTMS;.EQVAL<K,@LOC>;
27                    ?(.UP<@BLOCK>;.IS<@FORWARD,K S3>;?P,$-1))>;
28             ?(K:@START;@mesapp s1 p) )>
29
30    SS2;.LDEF
31    :&
32    .def<.bouge<@loc>, % @loc is a local variable
33     % verifies that K and K PU are on the same place
34     % but does not modify K nor the stack of K.
35    (@loc:k;pu;pd;.equal<k,@loc>;?$-,k:@loc)>
36
37    ss2;.ldef
38    :&
39     .DEF<.BODY,
40     (.IS<@WHILE>/S2;
41      .IS<@FOR>/S3;
```

```
42      .IS<@REPEAT>/S;
43      .IS<@BLOCK>/S3;
44      .IS<@LSTAT>/;
45      .IS<@CASE>/S2;
46      .IS<@CASERC>/S2;
47      .IS<@WITH>/S2;
48      .PROC;S3)>
49
50    SS2;.LDEF
51    :&
52     .DEF<.PROC,
53       (.IS<@BLOCK>/(.IS<@PROG,S>;?,(U;.APROC));
54       .IS<@FUNC>/(U2;.APROC);
55       .IS<@PROC>/(U2;.APROC);
56       .IS<@DEF>/(U;.PROC);
57       .IS<@IDENT>/(U;.PROC);
58       .APROC)
59       >
60
61    SS2;.LDEF
62
63
64    .end
65
66    .xin
```

4. Procedures parameters:

The parameters of Mentol procedures can be either pointers or procedures, and they can be passed either by value or by reference.

4.1. Formal parameters of procedures:

-- Pointers passed by value.

The name of the pointer must be placed in the procedure header in the proper argument position. ex: .def<.foo<... ,@toto, ...> , ... >;

-- Pointer passed by reference.

A special procedure '.byref' must be used as follows:

```
.def<.foo< ... ,.byref<@toto>, ... >, ...body... >;
```

-- Procedures.

The formal parameter must be a procedure header in the proper argument position as follows:

```
.def<.foo< ... ,.mac<@lulu>, ... >, ..body...>;
```

In this example, .mac is a formal procedure parameter passed by value.

4.2. Actual parameters

When a Mentol procedure is called, actual parameters are binded to formal parameters on the basis of their positions. When the formal parameter is a pointer passed by value, the corresponding actual parameter must be a Mentol variable or a structural address based on a Mentol variable (see mentor.gi.i). When the formal parameter is a pointer passed by reference, the actual parameter must also be a pointer (i.e. a Mentol variables without address modifiers). When the formal parameter is a procedure, the actual parameter can be any Mentol command.

Example:

```
.def<.foo<@f1,.byref<@f2>,.action>, ( ....) >
```

a legal call of foo is

```
.foo<@p s2,@q,.forall<@patt, d s1>>
```

4.3. Local variables in Mentol procedures.

All formal parameters declared in the header of a Mentol procedures that are not binded in the call of that procedures are considered local during that call.

Example:

```
.def<.foo2<@x,@y,@z>, ( ... ) >
```

A call of .foo2 with only one actual parameter will make @y and @z local variables during that call. We do not pretend this mechanism to be very clean in terms of programming languages. It is useful to implement procedures that can be called with different numbers of parameters.

5. How to input YOURFILE.tol in a Mentor session:

When your are in a mentor session, you can load the file containing the procedures you want to use by calling the system procedure '.XIN' or '.DEVIN'. These two procedures are equivalent when called without parameters. You can find their complete descriptions in the file mentor.procs.i.

Thus, you type .xin or .devin. The system will then ask for a file name and you answer the name of the file containing your procedures WITHOUT mentioning the extension '.tol'. Thus you just answer YOURFILE (or 'yourfile' in lower cases because of the equivalence under Mentor between lower and upper cases) to load the file called YOURFILE.tol.

However notice that the Unix-name of the file must be EXACTLY YOURFILE.tol with the first component in upper cases.

During the loading, your procedures are parsed by the Mentol

parser and then, errors may be found in your code. The error messages will appear on your terminal . You can know precisely in which procedures the errors are by setting the flag ECHO before loading the file. When ECHO is on Mentor will echo on your terminal ':&' each time it begins parsing a procedure and 'SS2.LDEF' when this procedure is parsed. If there are some errors the corresponding messages appear between these delimiters.

Setting and resetting the flag echo (see also tog-dials.i):

To set the flag ECHO, just call the Mentor procedure '.set' by typing .set on your terminal. Mentor will ask you for the flag name and you answer echo . See the documentation about .set in mentor.procs.i.

To reset the flag echo you have to do the same things but calling the Mentor system procedure .reset instead of .set. See the documentation about .reset in mentor.procs.i.

Completing your prelude file (see also multi-lang.i)

If you want to load the file YOURFILE.tol each time you will enter the system, you have to put the loading command in your prelude file. Your prelude file is the file called NAME.pre where NAME is the name you type in when Mentor asks for

"user name"

at the beginning of the session.

Standard .pre files exist for each available languages: PASCALUSER.pre for pascal, METALUSER.pre for metal and RAPUSER.pre for rapport (see in file mentor.gi.i).

To make your own NAME.pre file, copy the standard .pre file according to the language you use under Mentor. Then add in this copy the command

.xin<@YOURFILE>

This command is equivalent to the previous loading command we have seen except in that the file name is given as a parameter of the procedure .xin. Do not forget the @ preceding YOURFILE.

Example of prelude file:

```
1
2          %PASCAL
3          .XIN<@PASCALSTD>
4          .xin<@YOURFILE>
5          .XIN
```

Line 2 means that you want to manipulate Pascal programs.

Line three loads the file PASCALSTD.tol which is the standard pascal environment.
Line four loads your personnal file of Mentor procedures.
Line five switch the control to your terminal
(in the same way as does the last line of a mentol file).

6. Debbuging the Mentol procedures. Tracing:

*Tracing:

A tracing facility is available in Mentor. It uses the system procedures .oninter and .whereint, and the flags prestep, poststep, nostep, stepok and stepetat. If prestep (resp. poststep) is set, then the interrupt routine (see .oninter) is called before (resp. after) executing each mentor statement.

The flag nostep inhibits the trace when it is set . Thus you have to check that this flag is not set when you want to do some trace, and reset it if necessary.

During the execution of the interrupt routine, the flag nostep is set by Mentor, inhibiting the trace for the interrupt routine itself. The flag nostep can be reset by the user in the interrupt routine in order to perform other traces within the interruption. However this may cause loops if done without care.

*Stepok:

Before execution of the interrupt routine (see .oninter) the flag stepok is set if and only if the instruction executed just before then interrupt was successful. The user can change stepok during the interrupt routine so as to transmit either success or failure to the instruction executed after the interrupt. if a sucess is thus changed into failure, the error message given is 'e0 failure'.

The value of stepok can be superseeded on exit by the use of the exit statement $n when the absolute value of n is greater than 1.

However exiting the interrupt routine with $-1 is equivalent to exiting with $1, success or failure being determined only by stepok. (This is to avoid accidental perturbation of the interrupted program).

*.oninter:

The control flow of a mentor program may be interrupted either by means of the attention key (see attention) or by a systematic trace of the statements executed (see tracing). An interrupt routine is then invoked. The default interrupt routine is just an execution of the system procedure .user (See mentor.user.info). It is possible

to change the interrupt routine by calling .oninter< statement > . The new interrupt routine is then the statement given as parameter. this statement is executed almost (see stepok) as if enclosed in parentheses and inserted in the program where the interrupt occured.

Note that .oninter<.user> differ from the default routine because it encloses .user in parentheses.

The statement interrupted can be printed with .prmac<n> or .pp<n> where n is a holophrasting level.(see also .whereint).

*Stepetat:

When an interrupt (see tracing and attention) occurs in the execution of a mentor program, some information is printed on the terminal if the flag stepetat is set. this information consists of:

1) The nature of the interrupt:
   prestep, poststep or use of the attention key.
2) Whether the last statement executed succeded or failed (and the nature of the failure).
3) When in poststep, the number of compound statements still to be exited if an exit statement $n has been executed.

*Interrupt:

When an interrupt occurs due to the use of the attention key (see attention), then the flag interrup is set, in addition to the flag nostep (see tracing). However, unlike nostep, interrup is not reset when the interrupt routine (see .oninter) is exited.

If a new attention key interrupt occurs when interrup is set, it is the default interrupt routine .user which is executed. Thus if the interrupt routine loops, it is still possible to break the loop with a second attention key.

The user can reset interrup during execution of the interrupt routine in order to desable this protection mechanism. The next attention key interrupt will then call again the user defined interrupt routine, if any. However if this is done automatically in the user interrupt routine, without ever giving control to the console, there will then be no way to break a loop in a mentor program execution.

*Attention:

It is possible to interrupt a listing, or the execution of a mentor program by means of the attention key (BREAK on Multics). When a mentor statement is thus interrupted, an interrup routine is called. The interruption always occurs before the interrupted statement is executed. See .oninter, stepetat and interrup for more details.

*.whereint :

When a mentor program fails or is interrupted, it is possible to examine the statement that failed or was interrupted. One must call mentor recursively by executing .rec , asking for the language editeur, and then executing the system procedure .whereint .

This will assign to the variable k (of that level) the location of the guilty statement which can then be examined together with the surrounding ones, like any mentor object program.

However altering the program being executed is not garanteed to result in proper system functionning.

The manipulation of the language editeur, which is in fact the language Mentol, sould be reserved for the Mentor wizards. If you want to try it, you sould begin in studying the abstact syntax of Mentol and them try to understand very well all the Mentor commands that manipulate the language Mentol . The abstract syntax of Mentol can be obtained by asking for syntax when in help mode under an instance of Mentor in which the language edited is Mentol (editeur).

II. Comments about the Mentol procedures of the Example above:

We describe here the procedures that appear in the example of file given in the preceding section. The procedures defined in this file are actual procedures from the standard Pascal environment. Thus, their explaination can be found in the info segment mentor.procs.i.

You can find many others examples of Mentor procedures in all the files suffixed with '.tol' in the .../mentor/mentol directory.

(Sorry, this info file has to be completed)

mentor.procs.i

:Info: mentor.procs:

LANGUAGE INDEPENDANT COMMANDS OF MENTOR

Bertrand Mélèse

January 1983 (updated November 84)

10/11/83 List of system procedures available in MENTOR-

(For more informations about a system procedure during a mentor session, call the procedure ".sysdetails")

To Deal with Files

.delete	.deltxt	.devin	.devout	.load
.parse	.restore	.save	.store	.unparse
.xin	.xout			

Mentol Procedures Handling

.ass	.delmac	.def	.ldef
.lredef	.prmac	.reass	.redef

Dials and Toggles Handling

.reset	.set	.test

Tests on Trees

.eqtype	.equal	.eqval	.in	.is
.lt	.stype			

Constructors

.charid	.coerce	.concat	.gensym	.mkint
.pred	.substr	.succ	.swcase	

System Commands

.again	.day	.end	.err	.help
.language	.oncrkey	.oninter	.os	.rec
.status	.user	.whereint	.defframe	.sysannot
.newframe	.setprop	.resetprop	.setprefix	.setpostfix

Editing Mode

.invideo	.tty	.video	.menu

Others

.dump	.errmess	.eval	.phylum	.pnnl
.pp	.prmess	.searchto		

(only on Multics System)
For each one of these procedures, say .foo, there is an info segment called mentor.procs.foo.info that you can see by typing the command :

help mentor.procs.foo

:Info: mentor.procs.is:
11/13/80 .is<@construct {, syn_addr}>

To check whether the second argument matches the first one. If omitted, the syntactic address is @K by default. In the current version of Mentor, in case of success, the markers with the same name as the metavariables are side effected like in the F modifier. The procedure call may succed or fail.

Examples: (in Pascal)
.is<@if>
.is<@if,sls3>

:Info: mentor.procs.equal:
11/13/80 .equal<{syn_addr1},{syn_addr2}>

To check whether two addresses denote the same place. Any missing syntactic address defaults to @K.

Examples:
.equal<@X,@Y>
.equal<@X, S1 S4>

:Info: mentor.procs.eqval:
11/13/80 .eqval<{syn_addr1},{syn_addr2}>

To check whether the trees denoted are identical as trees. Any missing address defaults to @k. Trees are compared, but their attributes are not. This allows to compare programs up to their comments.

:Info: mentor.procs.stype:
11/13/80 .stype{< syn_addr {,syn_addr1}>}

To obtain the top level construct of a tree. If both addresses are missing, .stype prints the top construct of @K. The first address is used to know the top node of a different address. If the second address is specified, the value returned by stype is an atom that is available via this address.
The command .stype is systematically called by the Mentor redisplay procedure.

Examples:
.stype
.stype<@X S3>

.stype<@K, @RESULT>

:Info: mentor.procs.parse:
11/17/80 .parse{<{syn_addr},{syn_addr}>}

To parse a program, usually prepared under some other sytem. If the first argument is missing, a file name is requested from the user, else it must denote an atom which will be used to make_up the file name. The suffix associated by mentor to the manipulated formalism will be added to the file name. If a second argument is specified, it will denote the resulting tree.

:Info: mentor.procs.unparse:
11/17/80 .unparse{<{syn_addr},{syn_addr}>}

To create source text from a tree. If the first argument is missing, a file name is requested from the user, else it must denote an atom which will be used to make_up the file name. The suffix associated by mentor to the edited formalism will be added to the file name. If a second argument is specified, it denotes the tree to unparse, else the tree to unparse is denoted by the current pointer @k . Toggles and dials used to control pretty_printing may be used here. No check that the tree is a full-fledged program is performed.

:Info: mentor.procs.load:
11/17/80 .load{<{syn_addr},{syn_addr}>}

To read a tree from a file into your "work space". If the first argument is missing, a file name is requested from the user, else it must denote an atom will be used to make_up the file name. The suffix '.polish' ('.po' on Unix system) will be added to the file name. If a second argument is specified, it will denote the loaded tree; else the marker @K is denoting the tree after loading . Its old value is available in @dump.

:Info: mentor.procs store:
11/18/80 .store{<{syn_addr},{syn_addr}>}

To store a tree in a permanent file. If the first argument is missing, a file name is requested from the user, else it must denote an atom which will be used to make_up the file name. The suffix '.polish' ('.po' on Unix system) will be added to the file name. If a second argument is specified, the tree denoted is stored, else the tree denoted by @K is stored. This tree does not have to be a full-fledged program.

:Info: mentor.procs.os:
11/18/80 .os<syn_addr>

To passe a command line to the command processor (i.e. operating system). The argument should denote a comment line. The comment line will be executed and control will return to mentor.

:Info: mentor.procs.user:
11/18/80 .user

To give control to the user. To return control to mentor execution, use $ if you wish to succeed, $- if you wish to fail.

Example: (in Pascal and in tty mode)
This places you as a coroutine to a mentol procedure:
.forall<@block,(sl p;.user)>
If you type $, you go to the next block, if you type $-, you fail and exit the forall iteration as a consequence. Try it!

:Info: mentor.procs.end:
11/19/80 .end

To leave mentor. In fact, as a safety precaution, this will leave mentor only if you are at level 1.If you are at a higher level, mentor will kindly suggest that you should return to the ground before leaving and you will need to issue another .end command.

:Info: mentor.procs.lt:
11/19/80 .lt{<{syn_addr1 {,syn_addr2}>}

To compare two atoms. If specified, the first argument must denote an ident or a number, to obtain meaningful results. The second argument if specified must denote an atom of same type as the one denoted by the first argument. Both argument default to @k. The command succeeds iff the first atom is less than the second one. In the case of integers, the natural order is used, in the case of identifiers, the alphabetical order is used.

:Info: mentor.procs.day:
11/19/80 .day{<marker>}

To obtain the date and time. If an argument is specified, it must be a marker,

and then the result, a comment line containing the date and time, is returned in this marker. If marker is omitted, the date and time are printed on the terminal.

:Info: mentor.procs.gensym:
11/19/80 .gensym<syn_addr, marker>

To compute new identifiers. The address, specified as first argument, should denote an identifier that is going to be used as a model. The second argument must be a marker that will denote the result; it does not default to @k. The new identifier is obtain from the old one in this way:

- If the old one does not terminate with digits, 1 is tagged at right.
- If the old one terminates with a sequence of decimal digits, it is incremented by 1.
- If the trailing sequence of digits gets too big, .gensym fails.

Example:

Assume @X denotes the identifier A. The command:
(.gensym<@X,@X>)10
leaves @X denoting A10.

:Info: mentor.procs.again:
11/19/80 .again{<number>}

To execute the last command again. The argument may be used if you wish to repeat n times the last command. By the last command, we mean what mentor has understood to be the last command. So if you made a typing mistake that mentor corrected in a satisfactory manner, even though the erroneous command was not executed you may use .again to execute it. This is particularly useful for missing closed parentheses or brackets. An again command is not counted as the last command.

:Info: mentor.procs.pnnl:
12/06/80 .pnnl{<{syn_addr},{level}>}

To pretty_print wi hout a terminating CRLF. The first argument (by default @k) denotes the thing to be printed, what is usually put on the left of P. The second argument is the level of detail required, what is usually put on the right of P.

Example:

Assume K denotes a multiline comment. Then
.apl<@K,.pnnl<@k>>
will print a single line of text.

:Info: mentor.procs.eqtype:
10/11/83 .eqtype<{syn_addr1},{syn_addr2}>

To check whether the top level constructs of the trees denoted are identical. Any missing address defaults to @k .

:Info: mentor.procs.in:
10/11/83 .in<@phylum,{syn_addr}>

To check if the top level construct of the expression denoted by the second argument (by default the current expression) , belongs to the phylum specified as first argument.

Examples: (in Pascal)
.in<@stat,u2>
.in<@exp>

:Info: mentor.procs.eval:
11/30/84 .eval{<syn_addr{,@marker}>}

The first argument is @k by default. When the tree denoted by the first argument is an internal representation of a mentol command, the command is executed. When the tree denoted by the first argument is a Pascal constant expression (expression with integer constant operands and integer operators) , the expression is evaluated and the result is denoted by the second argument (@k by default).

:Info: mentor.procs.set:
10/11/83 .set{<syn_addr{,...}>}

To set toggles or to modify the dials value. Without argument, the user is prompted for a dial or toggle name. Otherwise, each syntactic address of the list passed as argument must denote an atom interpreted as dial or toggle name.

:Info: mentor.procs.help:
10/11/83 .help

To enter into the help_mode of the editor Mentor. In this mode you can get some general informations about the system or about your actual environnement.

:Info: mentor.procs.err:
10/11/83 .err

To enter in the err_mode of the editor. In this mode, the user can get the explanation of errors messages.

:Info: mentor.procs.reset:
10/11/83 .reset{<syn_addr{,...}>}

To reset dials or toggles. Without argument the user is prompted for a dial or toggle name. Otherwise, each syntactic address of the arguments list must denote an atom interpreted as a dial or toggle name.

:Info: mentor.procs.mkint:
11/30/84 .mkint{<syn_addr{,value}>}

The first argument (@k by default) must denote an atomic tree of integer or character kind. The second argument (0 by default) must be an integer constant. The procedure mkint affects that integer constant as new value for the denoted atomic tree (when the kind of the atomic tree is character, the atomic value becomes chr(constant)).

:Info: mentor.procs.succ:
10/11/83 .succ{<syn_addr>}

The tree denoted by the argument, by default the current expression, must be a numerical atom . The value supported by that atom is incremented.

:Info: mentor.procs.pred:
10/11/83 .pred{<syn_addr>}

The tree denoted by the argument, by default the current expression, must be a numerical atom. The value supported by that atom is decremented.

:Info: mentor.procs.concat:
10/11/83 .concat<syn_addr1,syn_addr2,syn_addr3>

To concat in a comment line denoted by the third argument the litteral value supported by atoms denoted by syn_addr1 and syn_addr2.

:Info: mentor.procs.coerce:
10/11/83 .coerce<@construct,@syn_addr1,@syn_addr2>

To transform, if possible, the tree denoted by the second argument in a tree denoted by the third argument with top level construct specified by the first argument.

Example: (in Pascal)
.coerce<@ident,@m b0 s1,@myname>

:Info: mentor.procs.swcase:
10/11/83 .swcase<n,@x,@y>

To change the INTERNAL REPRESENTATION of atoms by switching it from upper cases to lower cases or the reverse. The first parameter is an integer. @x must denote an atomic tree. @y will contain the resulting atom. When n is >0 , @y receives an atom whose INTERNAL REPRESENTATION is that of @x but with all letters changed into UPPER case letters. if n <= 0, all letters are changed into LOWER case letter.

Example: if @x is an identifier whose internal representation is FOO,
.swcase<-1,@x,@y> makes @y denote an identifier whose internal representation is foo.

Note that changing the internal represention DOES NOT necessarily change the way the atom is unparsed. For example in Pascal, when the toggle idlc is set (see toggs-dials.i) all identifiers are unparsed in lower cases. Thus, in this case both @x and @y in the example would have been unparsed as foo.

:Info: mentor.procs.charid:
10/11/80 .charid<@c,n1,n2>

This procedure has an effect only in Pascal. It is used to modify the set of characters allowed in Pascal identifiers. The Mentol variable @c must denote a Pascal expression which is a single character. Parameter n1 and n2 are integer values.

The value of n1 indicates whether the character must be added in or deleted from the set of legal characters. The value of n2 indicates if this addition or deletion applies only for the first character of Pascal identifiers or for all characters, but the first one, in Pascal identifiers.

When n2 is 1 the modification applies only for the first character of Pascal identifiers. When n2 is something else the modification applies for all characters but the first.

When n1 is <0 the character is made ILLEGAL in the position stated by the value of n2. When n1 is non negative the character is made LEGAL in the position stated by the value of n2.

Example:

Suppose that @c denote the expression '_' (underscore). The command .charid<@c,1,1> makes the character '_' (underscore) legal as first character of Pascal identifiers. The command .charid<@c,1,2> makes it legal in other positions. Thus to make '_' legal at any position, both commands have to be done.

:Info: mentor.procs.xin:
10/11/83 .xin

To read and execute mentol commands from a command file. The user is prompted for a file name. The suffix .mentol (.tol on Unix system) is added to the file name. The next '.xin' encountered returns command control.

:Info: mentor.procs.xout:
10/11/83 .xout

To obtain an output redirection onto a permanent file. The user is prompted for a file name. The suffix associated by mentor with the edited formalism is added to the file name. That redirection ends on the next '.xout' command.

:Info: mentor.procs.save:
10/11/83 .save{<{syn_addr},{syn_addr}>}

To store a tree in a permanent file. The command checks that you are saving a full program. If specified, the first argument must denote an atom that will be used to make up the name of the file in which the tree should be stored; otherwise, the user is requested for a file name. The suffix '.polish' ('.po' on Unix system) will be added to the file name. If specified, the second argument denotes the tree to be stored, otherwise the tree to be stored is denoted by the current pointer @k.

:Info: mentor.procs.restore:
10/11/80 .restore{<@filename{,@marker}>}

When called without parameters, it is the same as .load . When the first parameter is present, it is an immediate file name prefixed by the symbol @ .
Example:
.restore<@foo> Loads the polish file FOO.po in the current pointer.

The second parameter is the marker in which the file is loaded (@k by default).
Example:
.restore<@foo,@bar> Loads the polish file FOO.po in the mentol variable @bar .

:Info: mentor.procs.deltxt:
10/11/83 .deltxt{<syn_addr>}

To delete a text file in your "working space". If specified the argument must denotes an atom that will be used to make up the name of the file to delete; otherwise, the user is requested for a file name. The suffix associated by mentor with the edited formalism is added to the file name.

:Info: mentor.procs.delmac:
10/11/83 .delmac<.name>

To delete the definition of the mentol procedure 'name'.

:Info: mentor.procs.ass:
10/11/83 .ass<.name1,.name2>

To make the mentol procedure 'name1' equal to the procedure 'name2'.

:Info: mentor.procs.reass:
10/11/83 .reass<.name1,.name2>

To make the mentol procedure 'name1' equal to the procedure 'name2' . The previous definition of the mentol procedure 'name1' is erased.

:Info: mentor.procs.devout:
10/11/83 .devout{<syn_addr>}

To obtain an output redirection onto a permanent file. Without argument the user is prompted for a file name, otherwise the argument must denote an atom that will be used to make up the file name. The suffix associated by mentor with the edited formalism is added to the file name. That redirection ends at

the next '.devout' or '.xout' command.

:Info: mentor.procs.devin:
10/11/83 .devin{<syn_addr>}

To read and execute commands from a mentol commands file. If specified the argument must denote an atom that will be take to make up the file name, else the user is requested for a file name. The suffix .mentol (.tol on Unix system) is added to the file name. The next '.xin' or '.devin' command returns command control.

:Info: mentor.procs.pp:
10/11/80 .pp<.name>

Prints the code of the mentol procedure 'name' in a nicer format than '.prmac'

:Info: mentor.procs.prmac:
10/11/83 .prmac<.name>

Prints the code of the mentol procedure 'name'.

:Info: mentor.procs.rec:
10/11/83 .rec

To call recursively Mentor on a new session for editing in the same or another language. The procedure '.end' will terminate that new session and restore the old one with the same environnement as before the call to '.rec' command.

:Info: mentor.procs.def:
10/11/83 .def<header,body>

To define new procedures written in Mentol. The procedure takes two arguments:

- The fist one is the header of the defined procedure, composed of the name of the procedure, followed by the list of formal parameters between '<' and '>'.
- The second one is the defined procedure body which is a single command written in Mentol.

For more informations about writing procedures in mentol, see the info segment

mentor.wr_pr.info (mentor.wr_pr.i on Unix system) contained in the info sub_directory of the mentor system.

Example :

```
.def<.enter<@1,@obj>,(@1 f @obj;?,(@1 s* i @obj))>
```

:Info: mentor.procs.redef:
10/11/83 .redef<header,body>

To redefine a procedure written in Mentol. The procedure takes two arguments:

- The fist one is the header of the procedure to redefine, composed of the name of the procedure, followed by the list of formal parameters between '<' and '>' of the new definition.
- The second one is the redefined procedure body which is a single command written in Mentol.

If the procedure is already defined, then the old definition is overwritten, else '.redef' is equivalent to '.def' .

:Info: mentor.procs.ldef:
10/11/83 .ldef

To define a new procedure at previous editing level, after the procedure has been edited in a mentol mentor session.

:Info: mentor.procs.lredef:
10/11/83 .lredef

To redefine a new procedure at previous editing level, after the procedure has been edited in a mentol mentor session.

:Info: mentor.procs.oninter:
10/11/83 .oninter<mentol_command>

To change the interrupt routine. See informations about the tracing facilities by entering in help_mode during a mentor session or by consulting the info segment mentor.wr_pr.info (mentor.wr_pr.i on Unix system) in the info sub_directory of the mentor system.

:Info: mentor.procs.whereint:
10/11/83 .whereint

To examine the statement that failed or was interrupted during execution of a mentol command. See informations about that system procedure by consulting the info segment mentor.wr_pr.info (mentor.wr_pr.i on Unix system) in the info sub_directory of the mentor system, or by entering in help_mode during a mentor session.

:Info: mentor.procs.searchto:
10/11/80 .searchto<@pattern,syn_addr1,syn_addr2>

Searches the pattern in the tree denoted by the second argument without going outside the tree denoted by the third argument. This procedure is used to implement the '.foreach' tree traversal procedure.

:Info: mentor.procs.test:
10/11/83 .test{<syn_addr{,...}>}

To get the dials or toggles states. Without argument, the user is prompted for a dial or toggle name, otherwise each syntactic address in the argument list must denote an atom interpreted as a dial or toggle name. For dials, it gives the current value; for toggles, it fails iff the toggles specified are reset.

:Info: mentor.procs.dump:
10/11/83 .dump{<syn_addr>}

To push the structure denoted by the syntactic address on the pointer stack @dump . By default the syntactic address in the current pointer @k.

:Info: mentor.procs.status:
10/11/83 .status

Gives informations about the edited language, and the session level.

:Info: mentor.procs.language:
11/13/84 .language{<syn_addr>}

Only in Version 5 of Mentor.

To change the default language in the current session. After its call, the current session becomes a multi-language session.
If no argument, the user is requested for a language name. If present, the argument can be immediate or a Mentol variable denoting an atom that is the desired language name.

See multi-lang.i for more details.

:Info: mentor.procs.newframe
11/14/84 .newframe

To create a new frame in the current language. Asks for the frame name and the frame language name. Fails if something goes wrong.
(see also annotation.i)

:Info: mentor.procs.setprop
11/14/84 .setprop

To set a property of a frame defined for the current language.
Asks for the frame name and the property to set.
Fails if something goes wrong. (see also annotation.i)

:Info: mentor.procs.resetprop
11/14/84 .resetprop

To reset a property of a frame defined for the current language.
Asks for the frame name and the property to reset.
Fails if something goes wrong. (see also annotation.i)

:Info: mentor.procs.setprefix
11/14/84 .setprefix

To make a frame to have the property PRE and to NOT have the property POST.
Asks for the frame name that must be a frame defined for the current language.
Fails if something goes wrong. (see also annotation.i)

:Info: mentor.procs.setpostfix
11/14/84 .setpostfix

To make a frame to have the property PRE and to NOT have the property POST.
Asks for the frame name that must be a frame defined for the current language.
Fails if something goes wrong. (see also annotation.i)

:Info: mentor.procs.defframe:
11/13/84 .defframe{<arg1,arg2,arg3,arg4>}

Only in Version 5 of Mentor.

Definition of a new annotation frame. User is requested for Parameters that are not specified in the call.
Parameter can be immediate strings (of the form "ident) or Mentol variables.

See annotation.i for more details.

:Info: mentor.procs.sysannot:
11/13/84 .sysannot{<arg1,arg2,arg3,arg4,arg5>}

Only in Version 5 of Mentor.

Internal handling of annotations. This command is usable interactively but is mainly intended for performaing actions on annotations from within Mentol programs.
User is requested for Parameters that are not specified in the call.

The procedure .sysannot fails if something goes wrong.

See annotation.i for more details.

:Info: mentor.procs.delete:
10/11/83 .delete{<syn_addr>}

To delete a polish file in your 'working space". If specified, the argument must denote an atom that will be used to make up the name file, otherwise the user is requested for a name file. The suffix .polish (.po on Unix system) is added to the file name.

:Info: mentor.procs.oncrkey:
10/11/83 .oncrkey<command>

To specify a mentol command to be executed at each carriage return. It is used in Mentor to call the redisplay procedure at the end of every command. Calling this procedure at user level, and thus changing

the action perfom on carriage return will have bad effects since it will disable the Mentor redisplay.

Example: .oncrkey<.redisplay>

:Info: mentor.procs.prmess:
10/11/83 .prmess<file_name,number{,parameters}>

To send a message. The first two arguments are the name of the message module and the reference number of the message in the module. They may be followed by any number of other actual parameters for the message to be displayed. The first argument must be an immediate symbol (example : "EXAMPLE) or the value of a Mentol reference to a symbolic atomic tree. The second argument must be an immediate integer. For more information about messages system in mentor, see the corresponding info file (mentor.mess.i).

Example:
.prmess<"filemessl,10>

:Info: mentor.procs.errmess:
10/11/80 .errmess<errkind,file_name,number{,parameters}>

Same as .prmess but fails with the failure errkind.
Example:
.errmess<"fatal,"foo,10> fails and sends the message number 10
of the file FOO.mess prefixed by 'FATAL failure:' .

:Info: mentor.procs.invideo:
27/11/83 .invideo

Fails if you are in teletype mode. This procedure may be usefull when you want to write a mentol procedure whose comportment may be different when called in video mode or in teletype mode. (See the info file mentor.video.i).

Example : .invideo;?.switchvideo,switchtty;

:Info: mentor.procs.tty:
27/11/83 .tty

To return in teletype mode when you are in video mode. Has no effect if you are allready in teletype mode.

:Info: mentor.procs.video:
27/11/83 .video

To switch to video mode when you are in tty mode after a call to .tty. Has no effect if the current mode is already the video mode.

:Info: mentor.procs.phylum:
07/12/83 .phylum{<syn_addr>}

To get a phylum definition. Without argument the user is prompted for a phylum name. Otherwise the argument must denote an atomic tree whose value may be interpreted as a phylum name.

Example: (in pascal)
.phylum<@if s3> give the definition of the phylum STAT

:Info: mentor.procs.menu:
11/13/84 .menu

Only in the version 5 of Mentor

Calls the menu mode of editing. (i.e. beginner's mode)

:Info: mentor.procs.substr:
29/05/84 .substr<"mode,{addr1},{addr2},{addr3}>

To perform substring substitutions. The first argument must be an immediate string giving the substitution mode. The second argument must be a syntactical address (by default @k) denoting the scope of the substitution. The optional argument addr2 must denote an atom whose value will be used as substring pattern to be replaced. The optional argument addr3 must denote an atom whose value will be used as new substring. If addr2 (resp. addr3) argument does not exist, the user is prompted for the old (resp. the new) substring. If the mode argument is "first , only the first occurrence of the old string in the tree denoted by the second argument is modified. If the mode is "all , all occurrences are modified. If the mode is "query , the user is prompted for each occurrence of the old string in the tree denoted by the second argument; if the user hit a "q" the substitution is stopped, if the user hit a "n" the substitution is not done for that occurrence, otherwise the old string is replaced by the new string.

---

mentor.video.i

The full screen environment of Mentor

B.Mélèse, D.Verove
(April 84, updated November 84)

Mentor works in full screen mode on terminals with enough capabilies: The terminal must be able to support a full screen editor (Emacs-like) and must have at least two different fonts. The screen is divided in four windows. Mentor sends information into these windows according to their type.

- Switching between full screen and teletype modes :

The default mode is the full screen mode.

During a Mentor session it is possible to dynamically switch between modes.

The predefined procedure .tty switches from full screen mode to teletype mode. The predefined procedure .video switches back to full screen mode. If you are using a paper terminal put the .tty command in your prelude file. (see multi-lang.i).

- SCREEN DESCRIPTION :

The first line of the screen is the 'command window' in which the user enters commands.

The second and third lines of the screen are the 'diagnostic window' in which Mentor sends error messages and questions to be answered by the user.

At the end of the second line, a small window, the 'blip window', always contains the name of the current language and the type and language of the current highlighted expression. This type is the abstract syntax operator of the root of the subtree denoted by the current pointer (see multi-lang.i).

The rest of the screen, from the fourth line to the 24th line is the 'text window'. In that window is displayed the text representation of the structure you are editing: The text displayed in this window is the unparsing of the subtree denoted by the 'video pointer'. The video pointer is a predefined pointer in Mentor that you can move by means of special function keys described below. The video pointer is also automatically moved in order to respect the following rule: the tree denoted by the current pointer must always be a subtree of the tree denoted by the video pointer. In that manner, the text displayed on the screen always contains the current expression.

- Redisplay

.redisplay
This command is the Mentor redisplay. It is a Mentol procedure implemented using .invideo (see below).

The redisplay is called automatically by Mentor each time the user hits the <CR> key. This command can also be called from within Mentol procedures to force the redisplay at some point. A funny example (in Pascal) is a command such as

```
.foreach<@ident,.redisplay>
```

It make each identifiers belonging to the current subtree to be displayed.

- Window and buffer management commands (available in full screen mode only)

.do<.command>

The procedure '.do' first clears the screen, and then executes the mentol command given as argument. The video expression is not redisplayed so that results of the command remain displayed on screen until the next command.

Example:

```
.do<@aux p>
```

shows on the screen the expression denoted by the mentol variable @aux. The next carriage return will restore on the screen the video expression.

.hide, show — No more exist in the Version 5 of Mentor

.get<@variable>
.flip
.unget

These commands provide the user with a rought buffer mechanism. The .get command goes to a buffer containing the representation of the mentol variable passed as its parameter. This buffer is created if it does not yet exist. The .unget command goes back to the previous buffer. Buffers are stacked. The .get command pushes a new buffer on the buffer stack while .unget pops this stack. The .flip commands switches the two top buffers of the stack allowing easy switches and back between two buffers. The number of buffers in use is not limited: there is in fact one associated with each mentol variable. The default size of the stack is four.

.invideo

When called without parameter, this predefined procedure fails if the current mode is the teletype mode and succeeds if the current mode is full screen mode. It is useful to write mentol procedures whose behaviour has to be different depending on the current mode.

The command .invideo accept following parameters:

- .invideo<"clear> — Clears the screen
- .invideo<"display> — Calls the redisplay procedure
- .invideo<"isawit> — Succeeds if the current expression has been seen by the last call to the unparser

- .invideo<"notflash> Disables the redisplay
- .invideo<"flash> Reverses the effect of .invideo<"notflash>
- .invideo<"more> Puts the redisplay in "more" mode. Used in the implementation of the command .more.

.more<@addr>

To obtain unparsing of the tree denoted by address given as argument (by default the video expression). If the unparsed text has more then 21 lines, you are prompted with '--More--' when the 'text window' is full. If you hit the carriage return key, the next 21 lines are displayed; if you hit a 'q' followed by a carriage return, the command is interrupted. The last displayed page remains on screen until the next command is entered.

In 'text window' is displayed the video expression unparsed to the default holophrasting level. To change the default holophrasting level, use the command #n.

#n Modifies the holophrasting level. n is an integer (> 0) and becomes the default holophasting level. It is not safe to use a value less than or equal to 0 for n.

- USING FUNCTION KEYS :

The following commands can of course be entered directly as any other Mentol commands. Because of their functions, it is comfortable to use function keys of the terminal to call them. This will be possible on terminal that have programmable function keys by programming these keys to send the name of one of the following commands. It will be possible also on terminal that have function keys sending a sequence of characters of the form ESC followed only by letters or digits. In this case, the sequence sent by a function key must be taken as the name of a Mentol commands that calls one of the following already existing commands.

- Commands to modify the video expression (the prefixing '.' can be replaced by ESC (escape) when more convenient).

.p<CR> => To clear the screen.

.q<CR> => Makes the video expression equal to current expression. Then, only the current expression is displayed without context.

.r<CR> => Anti_zoom effect: The video pointer is moved up one level, so the displayed context around the current expression is extended if this is possible while keeping the current expression on the screen.

.s<CR> => Zoom effect: The video pointer is moved down one level toward current pointer, so the context is reduced.

- Commands to move the current pointer around in the tree: These commands should be attached to the arrow keys of the terminal.

.t<CR> => To perform the 'u' mentol command. (up arrow)

.u<CR> => To perform the 'l' mentol command. (left arrow)

.v<CR> => To perform the 'r' mentol command. (right arrow)

.w<CR> => To move current pointer to the next node in preorder tree traversal. (down arrow)

On HP2621 terminal these commands can be activated by keyboard function keys: f1 (for .p) , f2 (for .q) , ... , f8 (for .w) .

On terminals with programmable function keys, these keys must be initiated to send proper characters.

(see also mentor.gi.i)

pascal.procs.i

:Info: pascal.procs:

PASCAL ENVIRONNEMENT

B.Melese
(January 1984)

The Mentor procedures explained in this info segment are those written in the Mentol language. Other Mentor procedures, the system procedures, are described in the segment mentor.procs.info.

10/11/83 List of procedures available in MENTOR-PASCAL

(For more informations about a procedure during a mentor session, call the procedure '.details')

On_line Informations

.details	.ginfo	.list	.sysdetails	

Navigation

.body	.con	.cond	.ctx	.export
.fproc	.import	.lab	.leftms	
.proc	.typ	.up	.val	.var

Modifications

.case	.deccons	.decexport	.decforward	.decimport
.dectype	.decvar	.dvar	.isfirstlev	.label
.putdef	.putref	.putvar	.replaceall	.replaceid
.replaceidl	.toexport	.toimport	.wrap	

String substitutions

.sl	.sg	.sl

Call Graph Construction and Manipulation

.call	.closure	.crossrec	.directcall	.graph
.graphcomp	.whocalls			

Normalization

.clean	.combods	.comprocs	.normalize	.sat

Structured inclusion

.include

Trace, Execution Profile

.count	.profile	.strace	.supp	.trace

Tree Traversal

.apl	.enter	.forall	.foreach	.sort
.next	.prev	.rsearch		

System Commands

.compile	.comp_ld	.exit	.multics	.unix
.write				

Commands controlling the video display
(Information on these commands is in mentor.video.i)
.more .get .unget .do .hide
.show

:Info: pascal.procs.call:
11/18/80 .call

Asks for two procedure or function names (P and Q) and then indiquates if the first may call the second or not. The path found (if some) is kept in the variable '@path' and thus, may be used for some other purpose. Notice that '@path' is the first path found between P and Q rather than the shortest.

:Info: pascal.procs.sat:
a010c011R
11/18/80 .sat

Suppress the anonymous types that introduce new identifiers. The names of these types may be chosen by the user or generated by the system.

:Info: pascal.procs.crossrec:
11/18/80 .crossrec

Asks for two procedure or function names and then indiquates if they are mutually recursive or not. If P and Q are two procedures or functions, .crossrec<p,q> is equivalent to (.call<p,q> and .call<q,p>).

:Info: pascal.procs.case:
11/18/80 .case

If the current position is inside a 'case' statement, .case will indicate in what alternative the current position is. It fails if it do not find a 'case' operator when going up.

:Info: pascal.procs.var:
11/18/80 .var

Go to the 'var' declaration part of the current block . We call 'current block' the most internal block in which the current position is, according to the scope rules of Pascal. When the current position is at the top of some

block P, the current block is not P but the 'father' of P, unless P is the root of the whole program. When the current position is somewhere in the title of some block Q, the current block may be Q or its father according to the EXACT position of the current marker K, following the scope rules of Pascal.

:Info: pascal.procs.lab:
11/18/80 .lab

Go to the 'label' declaration part of the current block. (See also the details about '.var').

:Info: pascal.procs.con:
11/18/80 .con

Go to the 'const' declaration part of the current block .(See also the details about '.var').

:Info: pascal.procs.val:
11/18/80 .val

Go to the 'value' declaration part of the current block .(See also the details about '.var').

:Info: pascal.procs.typ:
11/18/80 .typ

Go to the 'type' declaration part of the current block .(See also the details about '.var').

:Info: pascal.procs.import:
10/11/83 .import

Go to the 'import' declaration part of the module (Dependent of the compiler separated compilat.on mechanism).

:Info: pascal.procs.export:
10/11/83 .export

Go to the 'export' declaration part of the module (Dependent of the compiler separated compilation mechanism).

:Info: pascal.procs.up:
11/18/80 .up{<@op>}

If called without any argument it is equivalent to 'u'. When called with an argument '@op', goes up until an operator @op is reached (Warning : it is equivalent to 'u*' if the operator @op cannot be reached).

Example : .up<@if>

:Info: pascal.procs.fproc:
11/18/80 .fproc{<.position>}

Asks for a procedure or function name and goes to it. When .fproc is called without parameters, the search begins at the current position. It is possible to give one parameter to .fproc to choose the point where the search will start. The parameter indiquates a move from the current position, in the same way as the parameters of '.decvar' do. (See the details about .decvar).

Examples:

.fproc	The search will begin at the current point,
.fproc<.proc>	Will begin the search from the top of the current sub-program,
.fproc<(u2;s1;r3)>	Will first move as indiquated by (u2;s1;r3) and the search will begin there (if this move does not fail !),
.fproc<.fproc>	Is very useful when the nesting of sub-programs is very deep.

:Info: pascal.procs.body:
11/18/80 .body

Go to the body of the current statement or block

:Info: pascal.procs.proc:
11/18/80 .proc

Go up to the top of the current block according with the scope rules of Pascal. If the current position is somewhere in the title of a procedure or a function, but not on the parameters, the current block is not that block but the external one.

:Info: pascal.procs.label:
11/18/80 .label

Asks for a label, that is, an integer constante, declares it in the current block and labels the current statement. Fails if the current position is not a statement. It verifies that the given label does not exist yet in the current block and asks for another one if necessary. If the label declaration part does not exist in the current block, it is created.

:Info: pascal.procs.decforward:
11/18/80 .decforward{<.position>}

This procedure creates a 'forward' declaration of a procedure or a function. The current marker must be somewhere inside a procedure or function declaration: this subprogram will be forward declared. The procedure '.decforward' can receive one parameter. When specified, this parameter indiquates a move to do on the current marker before doing the forward declaration.

The forward declaration is created before the first sub-program declaration of the corresponding level. The procedure '.decforward' verifies that the forward declaration to be created does not exist yet. Let PF be the newly forward declared procedure or function: If PF had parameters, .decforward put them in the forward declaration. The same is done for the type if PF is a function. Parameters and type of PF are kept as comments in the actual declaration of PF.

:Info: pascal.procs.decvar:
11/18/80 .decvar{<{.dl},{.vl}>}

Can be used to declare new variables in a program. It asks for a variables list and then for the type of these variables. If necessary, it creates the 'var' declaration part in the corresponding block. Before declaring variables it checks that they do not exist yet. Two parameters can be passed to .decvar:

- The first indiquates in which block the declarations should be done.
- The second controls the scope in which the verification will be done.

Both arguments may be omitted and, thus, they have a default value. (The second argument may be omitted even if the first is there). The default values are '.proc' for the first argument and 'u*' for the second. This means that calling .decvar is equivalent to .decvar<.proc,u*>, and will declare variables in the current block verifying that they do not exist yet in the most external scope. Examples:

.decvar<(.proc)2> Will declare variables in the block containing the current block verifying that they do not exist yet in the most external scope.

.decvar<.fproc,.proc> Will first ask for a procedure or function name (see .fproc info) and then declare the variables in the corresponding block verifying that they do not exist yet in THIS block.

:Info: pascal.procs.dectype:
11/18/80 .dectype{<{.dl},{.vl}>}

This procedure can be used to declare new types in a program. It asks for the name of the type to be declared, and then for the value of this type. Before declaring the type it checks that its name does not exist yet. The block in which the type will be declared and the scope in which the verification will be done may be controlled by parameters passed to .dectype. The use of these parameters is explained in the documentation about .decvar.

:Info: pascal.procs.deccons:
11/18/80 .deccons{<{.dl},{.vl}>}

This procedure can be used to declare new constantes in a program. It asks for the name of the constante to be declared, and then for its value. Before declaring the constante it checks that its name does not exist yet. The block in which the constante will be declared and the scope in which the verification will be done may be controlled by parameters passed to .deccons. The use of these parameters is explained in the documentation about .decvar .

:Info: pascal.procs.decimport:
10/11/83 .decimport

This procedure can be used to declare new variables as external references in the 'import' declaration part of a module. It asks for the name of the variables to be such declared, and then for their type. Before declaring the variables it checks that they does not exist yet at first level.
(Dependent of the compiler separated compilation mechanism).

:Info: pascal.procs.decexport:
10/11/83 .decexport

This procedure can be used to declare new variables as external definitions in the 'export' declaration part of a module. It asks for the name of the variables to be such declared, and then for their type. Before declaring the variables it checks that they does not exist yet at first level.
(Dependent of the compiler separated compilation mechanism).

:Info: pascal.procs.clean:
11/18/80 .clean

The procedure '.clean' cleans up the program by suppressing the empty

statements (ie. useless ';') and useless compound statements (ie. useless begin end).

:Info: pascal.procs.comprocs:
11/18/80 .comprocs

Comments the end of each block by the name of this block. If the end of a block has already a comment, nothing is done for that block.

:Info: pascal.procs.combods:
11/18/80 .combods

The procedure '.combods' puts a comment of the form 'BODY OF FOO' at the beginning of the body of procedures and functions. The comment is put only if the body is not empty and if the procedure has internal declarations of sub-programs.

:Info: pascal.procs.normalize:
11/18/80 .normalize

The procedure '.normalize' applies the three following procedures to the whole of the program: '.clean', '.combods', '.comprocs' .

:Info: pascal.procs.trace:
11/18/80 .trace

This procedure asks for a statement. This statement will be evaluated (in the Mentor sense, that is, by the Mentor primitive E) and then inserted at the beginning of every procedure or function of the program.If the given statement is 'writeln('entering ',$name)', the effect will be a trace of every procedure and function.(See a Mentor Manual for more information about E and the Mentor $-notation).

:Info: pascal.procs.strace:
11/18/80 .strace

This procedure asks for the names of procedures and functions to be traced. A tracing statement will be added at the begining of the body of each of them. All these extra statements are labeled (and the corresponding labels are declared : the program remains correct) and each label as a comment which is

himself. The main purpose of labeling and commenting this way is to make possible the automatic suppression of the tracing instuctions by the procedure '.supp'.

:Info: pascal.procs.count:
11/18/80 .count

Procedure very useful when somebody wants to know howmany times some part of the program is executed. It assumes that K denotes a statement, and adds what is needed to count howmany times this statement is reached during execution of the program. The following steps are processed by '.count':

-- Ask for a counter name,
-- Verify that this name does not exist yet in the program,
-- Declare this counter this type Integer,
-- Initialize it to 0 at the beginning of the program,
-- Increment it just before the statement to be counted,
-- Write the counter name and value at the end of the program.

Case of labeled statement not implemented. The program MUST be a MAIN program.
Example:
The following Mentor command will count executions of all While loops of the program: ' .forall<@while,(s2;.count)>' . All the statements created by .count may be automaticaly supressed by the procedure '.supp' . (All these statements were labeled as explained in the informations about the procedure '.strace').

:Info: pascal.procs.supp:
11/18/80 .supp

Is the procedure that suppress all the statements and label declarations created by procedures '.count' and '.strace'. After doing this, the procedure '.clean' is applied to the program.

:Info: pascal.procs.graph:
11/18/80 .graph

The procedure '.graph' computes the call-graph of the user defined procedures and functions. The graph is then printed in the following format: there is one output line per procedure and function; one output line of the form P(Q,R,S) means that the procedure (or function) P calls the three procedures or functions Q,R,S.

:Info: pascal.procs.graphcomp:
11/18/80 .graphcomp

The procedure '.graphcomp' computes the call-graph of all the procedures and

functions used in the program. The graph is then printed in the following format: there is one output line per user defined procedure and function; one output line of the form P(Q,R,S) means that the user defined procedure (or function) P calls the three procedures or functions Q,R,S.

:Info: pascal.procs.whocalls:
11/18/80 .whocalls

The procedure '.whocalls' asks for a procedure or function name (P say) and answer the names of all the procedures and functions that may call P. If a graph were already computed (ie. by '.graph' or by '.graphcomp'), the last computed graph is used. If no graph were already computed, '.whocalls' first computes the complete graph (ie. applies '.graphcomp' without printing the graph). Thus, the first call of '.whocalls' may be a little slower than the next ones.

:Info: pascal.procs.closure:
11/18/80 .closure

The procedure '.closure' computes the transitive closure of the last computed call-graph of a program. If no call-graph were already computed '.closure' first computes a graph by applying the procedure '.graph' to the program.

:Info: pascal.procs.directcall:
11/18/80 .directcall

Asks for a procedure or function name (PF) and answers the names of all the procedures and functions that PF may call at the first level.

:Info: pascal.procs.forall:
11/18/80 .forall<@patt,.action>

Search all occurences of the pattern @patt, and apply the action .action on each. The search is done in all the text of the program starting at the current position. If the action .action fails on some occurence of the pattern the execution stops on that occurence.

Examples:

.forall<@block,sl p>	Will print all procedures and functions headers which are 'after' the current position in preorder
.forall<@sch1,c @sch2>	Will replace, from the current position, all occurences of @sch1 by @sch2. These two schema must be defined before .

:Info: pascal.procs.foreach:
11/18/80 .foreach<@patt,.action>

Same syntax and effect than .forall but the search is done only in the subtree whose root is the current position. Thus, .forall and .foreach are equivalent when starting at the top of the program.

:Info: pascal.procs.apl:
11/18/80 .apl<.action>

Assumes that the current position is a list node (fails if it is not the case), and applies the action .action on all the elements of that list. Fails and stays here if .action fails on some elements.

:Info: pascal.procs.enter:
10/11/83 .enter<@list,@object>

Add the object at the end of the list, but only if the object was not yet contained in the list. The type of the object must correspond with the type of the list elements.

Example :

u*;.foreach<@ident,.enter<@allident,@k>>

insert all the identifiers of the program in the list @allident.

:Info: pascal.procs.exit:
11/18/80 .exit

This procedure may be used to terminate a Mentor session. It saves the current program (PRO) by overwriting (or creating) both files PRO.polish and PRO.pascal. The prefix comment is updated with the current user-name, date and hour.

:Info: pascal.procs.ctx:
11/18/80 .ctx

Goes up until it finds a while, for, repeat, case, block, or begin ... end statement.

:Info: pascal.procs.cond:
11/18/80 .cond

Go to the condition of a while, repeat, for, or if statement. Fails if the current position is not on such a statement.

:Info: pascal.procs.dvar:
11/18/80 .dvar

Suppress the 'var' option of the parameter declaration pointed out by the current marker. Thus, the corresponding parameter is now declared as a 'value' parameter.

:Info: pascal.procs.putvar:
11/18/80 .putvar

The procedure '.putvar' is the reverse operation of '.dvar': it transforms a 'value' parameter into a 'var' parameter.

:Info: pascal.procs.putdef:
11/18/80 .putdef

The procedure '.putdef' transforms a declaration into a 'def' declaration. That is, it adds the 'def' keyword of Pascal in the declaration if it is not already there. This procedure begins by verifying that the declaration to be transformed is at the first level in the Pascal program and it does nothing if it is not the case.
If the current position is not a declaration of variable, function, or procedure, an error will occur.
(Dependent of the compiler separated compilation mechanism).

:Info: pascal.procs.putref:
11/18/80 .putref

The procedure '.putref' is the dual operation of the procedure '.putdef'. It adds the identifier found by applying the procedure '.leftms' to the current pointer in the list of parameters of the program. Fails in some bad cases.
(Dependent of the separated compilation mechanism).

:Info: pascal.procs.toimport:
10/11/83 .toimport

The procedure '.toimport' moves a variable declaration contained in the 'var' declaration part of the first level of a module into the 'import' declaration part. So the variable is transformed into an external reference.
Before to call that procedure, the current pointer must design the name of the variable in its declaration.
(Dependent of the compiler separated compilation mechanism).

:Info: pascal.procs.toexport:
10/11/83 .toexport

The procedure '.toexport' moves a variable declaration contained in the 'var' declaration part of the first level of a module into the 'export' declaration part. So the variable is transformed into an external definition.
Before to call that procedure, the current pointer must design the name of the variable in its declaration.
(Dependent of the compiler separated compilation mechanism).

:Info: pascal.procs.isfirstlev:
11/18/80 .isfirstlev

Check if the current position is at the first level in the Pascal program. Succeed if it is, fail if it is not.

:Info: pascal.procs.wrap:
11/18/80 .wrap

The procedure '.wrap' transforms the current statement into a list of statements with only one son. Thus , in the Pascal program, the statement will be inserted within a begin....end pair (compound statement).

:Info: pascal.procs.multics:
11/18/80 .multics

The procedure '.multics' is an interface between Multics and Mentor. It asks for a line. This line will be passed to Multics as a command line, and then, will be interpreted by Multics .

(Only on Multics System).

:Info: pascal.procs.unix:
11/18/80 .unix

The procedure '.unix' is an interface between Unix and Mentor. It asks for a line. This line will be passed to Unix as a command line, and then, will be interpreted by Unix .
(Only on Unix System).

:Info: pascal.procs.list:
11/18/80 .list

The procedure '.list' gives the list of all the procedures that are written in Mentol for the Pascal environnement, and then all the predefined procedures of the Mentor system.

:Info: pascal.procs.ginfo:
11/18/80 .ginfo

Gives some general informations about the Mentor system.
- On Multics, these informations are taken from the file mentor.gi.info. After calling the procedure .ginfo the user is inside the help processor of Multics. When the user lives this processor, by answering 'no' or reaching the end of the file, he comes back to his interrupted Mentor session.
- On Unix, these informations are taken from the file mentor.gi.i by the Unix command 'more' . The user can type a space to continue or 'q' to return to the mentor session.

:Info: pascal.procs.replaceid:
12/04/80 .replaceid

Interactively replaces all occurences of one identifier with another. The command prompts for both identifiers, and then searches forward for each occurrence of the first identifier. It prints a little program fragment around this occurence and waits for one of the following responses:

$ -- replaces this particular occurrence of the first identifier with the second. hen searches for the next o ccurrence o f the first identifier and waits for a response again.
$- -- leaves this occurrence of the first identifier alone and searches for the next occurrence of the first identifier.
$-3 -- terminates the query replace without modifying this occurrence of

the first identifier.

:Info: pascal.procs.replaceidl:
10/11/83 .replaceidl

Interactively replaces all occurences of one identifier with another. The command prompts for both identifiers, and then searches forward for each occurrence of the first identifier. It prints a little program fragment around this occurence and waits for one of the following responses:

yes / y -- replaces this particular occurrence of the first identifier with the second. Then searches for the next occurrence of the first identifier and waits for a response again.
no / n -- leaves this occurrence of the first identifier alone and searches for the next occurrence of the first identifier.
q -- leaves the procedure without modify this occurrence.

:Info: pascal.procs.replaceall:
04/07/82 .replaceall

Inconditionnally replace all occurences of an identifier by another in the current subtree.

:Info: pascal.procs.write:
03/05/81 .write

This procedure may be used to save a Pascal program . It saves the current program (PRO) by overwriting (or creating) both files PRO.polish and PRO.pascal. The prefix comment is updated with the current user-name, date and hour.

:Info: pascal.procs.compile:
03/05/81 .compile

The procedure .compile first calls the procedure .write. Then, it calls the pascal compiler on the current program.

:Info: pascal.procs.comp_ld:
10/11/83 .comp_ld

(Only on Unix System).
The procedure '.comp_ld' first calls the procedure '.write'. Then, it calls

the pascal compiler on the current program. This procedure is to use when the current program is a complete program, and then the result of the compilation is an executable segment.

:Info: pascal.procs.details:
10/11/83 .details

The procedure .details give some informations about all the procedures available in the Pascal environnement (ie. procedures written in mentol, or predefined procedures of the Mentor system). The user has to enter the name of a procedure in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: pascal.procs.sysdetails:
10/11/83 .sysdetails

The procedure .sysdetails give some informations about all the predefined procedures of the Mentor system. The user has to enter the name of procedure in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: pascal.procs.leftms:
03/05/81 .leftms

Moves the current pointer to the first leaf (in preorder search) encountered in the current expression. Fails if it is not an identifier.

:Info: pascal.procs.next:
03/05/81 .next

Moves the current pointer to the next node (according the preorder search).

:Info: pascal.procs.prev:
03/05/81 .prev

Reverse of .next ; Moves the current pointer to the previous node (according the preorder search).

:Info: pascal.procs.rsearch:
03/05/81 .rsearch<@pattern>

Backwards search of patterns;
Moves the current pointer to the previous occurence of the pattern (according the preorder search).

:Info: pascal.procs.sort:
03/05/81 .sort{<{.key},{@list}>}

Sorts the list, by default the current expression, according a certain key.

Examples :

u*;f @const; .sort<sl>

Reorganizes the 'const' part of a program according the lexical order applied to the constantes identifier.

.foreach<@ident,.enter<@lid,@k>>;.sort<,@lid>

To arrange the identifiers list @lid in alphabetical order.

:Info: pascal.procs.sg:
7/3/84 .sg

Works only on COMMENTS and STRINGS.
Global string substitution.

:Info: pascal.procs.sl:
7/3/84 .sl

Works only on COMMENTS and STRINGS.
Local string substitution.

:Info: pascal.procs.sl:
7/3/84 .sl

Works only on COMMENTS and STRINGS.

One string substitution. Make the substitution only at the next occurence of the old string.

:Info: pascal.procs.include:
7/3/84 .include

The .include command is not loaded as part of the standard PASCALUSER environment. To use it you have to explicitely load the Mentol file PASINCLUDE.tol using the .xin command. If you use it often, add in your personnal .pre file the command .xin<@pasinclude>.

The .include command performs structured merging of pascal programs. It includes a module (that is a pascal program with no body) into the current program or module. The current program or module is saved in polish form before beeing modified by the include command. The include command merges corresponding declaration parts. No verifications are made on double declarations or type coherence.

When called without parameters, it asks first for the module to be included in the current program. This module must exist in polish form and is loaded from the corresponding file. Then the include command asks for the name of the program to be created after the merging. If the name given here is the name of the current program there is a risk of overwriting the initial form of this current program next time it will be saved into a file since file names used by mentor are names of programs. In all cases, the current program is lost in the current mentor session.

The include command can also be called with 2 arguments that must be Mentol variables denoting identifiers. In this case, the first argument is the name of the module to be included, and the second argument is the name of the program to be created after the inclusion.
If the include command is called with only one argument it asks for the second one.

:Info: pascal.procs.more:
7/3/84 .more

The more command is very usefull to scroll on the current (i.e. enhanced) part of the program on a page per page basis.
It writes --more-- on the top of the screen and waits for an answer.
Answer <CR> to see next page,
or q to quit the more command
After the q answer, or when the last page has been displayed,the next <CR> goes back to the point from where the more command were called.

---

rap.procs.i

RAPPORT ENVIRONNEMENT

B.Mélèse, D.Verove
(June 1984 updated November 84)

List of procedures available in MENTOR-RAPPORT

For more informations about a procedure during a mentor session, call the procedure '.details' or the procedure .ginfo that takes input in the file .../mentor/info/Rapport_gi.i (in french).

WARNING : Remember that all Rapport specific commands have french names.

On_line Informations

.details	.ginfo	.list	.sysdetails	

Navigation

.bib	.chercher	.corps	.debut	.paragraphe
.pder	.ppre	.psuiv	.section	.titre
.more	.progsuiv			

Modifications

.casser	.creer	.de	.detail	.df
.combiner	.indent	.programme	.telquel	.tribib
.normalpar	.emboiter	.remonter		
.sl	.sl	.sg	.sq	.sc

File Handling

.document_complet		.ecrire	.pdsa
.regrouper	.sauver	.separer	.sortir
.s_automatique			

Editor Calling

.edit	.emacs	.mentor

Text composition commands

.compose	.quickcomp

Unparsing Parameters

.en_tetes	.ne_pas_numeroter	.numeroter	.pc
.pl	.plan		

Tree Traversal

.apl	.forall	.foreach

System call

.unix

:Info: rapport.procs.forall:

11/18/80 .forall<@patt,.action>

Search all occurences of the pattern @patt, and apply the action .action on each. The search is done in all the text of the program starting at the current position. If the action .action fails on some occurence of the pattern the execution stops on that occurence.

Examples:
.forall<@section,(sl; .redisplay)> Will redisplay all chapter titles which are 'after' the current position in preorder
.forall<@schl,c @sch2> Will replace, from the current position, all occurences of @schl by @sch2. These two schema must have been defined before .

:Info: rapport.procs.foreach:
11/18/80 .foreach<@patt,.action>

Same syntax and effect than .forall but the search is done only in the subtree whose root is the current position. Thus, .forall and .foreach are equivalent when starting at the top of the program.

:Info: rapport.procs.apl:
11/18/80 .apl<.action>

Assumes that the current position is a list node (fails if it is not the case), and applies the action .action on all the elements of that list. Fails and stays here if .action fails on some elements.

:Info: rapport.procs.unix:
11/18/80 .unix

The procedure '.unix' is an interface between Unix and Mentor. It asks for a line. This line will be passed to Unix as a command line, and then, will be interpreted by Unix .

:Info: rapport.procs.list:
11/18/80 .list

The procedure '.list' gives the list of all the commands available in the Mentor-Rapport environment.

:Info: rapport.procs.ginfo:
11/18/80 .ginfo

Gives some general informations about the Mentor system.
On Unix, these informations are taken from the file mentor.gi.i through the 'more' command. The user can type a space to continue or 'q' to return to the mentor session.

:Info: rapport.procs.document_complet:
03/05/81 .document_complet

This procedure may be used to save a Document . It saves the current document by overwriting or creating the file FOO.rapport, where FOO is the name of your document.

:Info: rapport.procs.details:
10/11/83 .details

The procedure .details give some informations about all the procedures written in Mentol, available in the Rapport environnement. The user has to enter the name of a procedure in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: rapport.procs.sysdetails:
10/11/83 .sysdetails

The procedure .sysdetails give some informations about all the predefined Mentor system procedures. The user has to enter the name of a procedure in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: rapport.procs.bib:
7/2/84 .bib

Goes on the bibliography of the document if it exists

:Info: rapport.procs.chercher:
7/2/84 .chercher

Asks for an identifier. Goes to the section identified by this identifier if it exists.

:Info: rapport.procs.corps:
7/2/84 .corps

Goes to the body of the current part of the document.

:Info: rapport.procs.debut:
7/2/84 .debut

Goes back to the beginning of the document.

:Info: rapport.procs.paragraphe:
7/2/84 .paragraphe

To be written. Information (in French) can be found in the file .../mentor/info/Rapport_gi.i.

:Info: rapport.procs.pder:
7/2/84 .pder

Goes to the last paragraph.

:Info: rapport.procs.ppre:
7/2/84 .ppre

Goes to the previous paragraph.

:Info: rapport.procs.psuiv:
7/2/84 .psuiv

Goes to the next paragraph.

:Info: rapport.procs.progsuiv:

11/13/84 .pprogsuiv

Only in version 5 of Mentor. Goes to the next program

:Info: rapport.procs.section:
7/2/84 .section

Goes up to the surrounding section.

:Info: rapport.procs.titre:
7/2/84 .titre

Goes to the title of the current document, section, or bibliographie element.

:Info: rapport.procs.casser:
7/2/84 .casser<n>

The argument is an integer. Splits the current paragraph after its n-th line giving two adjacent paragraphs. This is useful when one wants to enter several paragraphs calling the text editor only once.
When splitting a special paragraph (detail, indent, telquel or programme) the created paragraph has the same attribute.

:Info: rapport.procs.creer:
7/2/84 .creer

To create new parts of the document (section, paragraph, figure, bibliography or programs)

:Info: rapport.procs.de:
7/2/84 .de

To initialize the creation of a new document in english.

:Info: rapport.procs.detail:
7/2/84 .detail

Transforms the current paragraph into a paragraph that will be itemized.

:Info: rapport.procs.df:
7/2/84 .df

Pour creer un nouveau document en francais.

:Info: rapport.procs.indent:
7/2/84 .indent

Transforms the current paragraph into a paragraph that will be indented.

:Info: rapport.procs.programme:
7/2/84 .programme

Same as .telquel in the current version.

:Info: rapport.procs.telquel:
7/2/84 .telquel

Transforms the current paragraph into a paragraph that will be output without modifications when composed.

:Info: rapport.procs.tribib:
7/2/84 .tribib

Sorts the bibliography.

:Info: rapport.procs.compose:
7/2/84 .compose

Creates a file named DOCCOMP.rapport, if the name of the document is DOC, that contains the input to be send to TROFF or NROFF.
Creates also a file called DOCPLAN.rapport that contains the table of contents of the document.

To process this files using NROFF or TROFF or any text processing command based on these, remember that the generated form uses the me macros. Thus, the command to process the file by NROFF is

```
nroff -me DOCCOMP.rapport
```

To compose french documents you have to send it to TROFF using the "versatec" command. This command assumes that following conventions are respected for french special symbols:

- all accents are put AFTER the character:
  e' e` e^ a` a^ u` u^ i^ o^

- Letters with "trema" accents must be given followed by ":"
  i: a: e: u:

- c/ for "c cedille" examples : fac/on, franc/ais

:Info: rapport.procs.quickcomp:
7/2/84 .quickcomp

Same effect than .compose but does not save the current state of the document. Thus, after this command the document not more exists in the current session and must be reloaded from the corresponding polish file if more edition has to be done on it.
The advantage of .quickcomp is to be faster than .compose.

:Info: rapport.procs.ecrire:
7/2/84 .ecrire

To be written. Information (in French) can be found in the file .../mentor/info/Rapport_gi.i.

:Info: rapport.procs.pdsa:
7/2/84 .pdsa

Suppresses the automatic saving of the document. (see the command s_automatique).

:Info: rapport.procs.regrouper:

7/2/84 .regrouper

Reverse operation of .separer. See .separer.

:Info: rapport.procs.sauver:
7/2/84 .sauver

Save the document in the Mentor polish form.

:Info: rapport.procs.separer:
7/2/84 .separer

Used to split big documents into multiple files. Asks for a file name and put the current chatper in the corresponding file. Only chapters can be separated.

:Info: rapport.procs.sortir:
7/2/84 .sortir

Leaves Mentor after having saved the document.

:Info: rapport.procs.s_automatique:
7/2/84 .s_automatique

Enters a mode in which the document is saved after each modification. (this is an expensive mode !!).

:Info: rapport.procs.edit:
7/2/84 .edit

Calls the tex editor (emacs by default) on the current paragraph. If the current paragraph is a program paragraph (progpar), switch the mentor editing mode to the appropriate language. The program is then edited under Mentor as usual.

:Info: rapport.procs.emacs:
7/2/84 .emacs

Sets emacs as current text editor.

:Info: rapport.procs.mentor:
7/2/84 .mentor

Sets mentor as current text editor (for debugging purposes only).

:Info: rapport.procs.en_tetes:
7/2/84 .en_tetes

Like .plan but shows also identifiers of sections. Usefull to remenber these identifiers.

:Info: rapport.procs.ne_pas_numeroter:
7/2/84 .ne_pas_numeroter

Reverse of .numeroter.

:Info: rapport.procs.numeroter:
7/2/84 .numeroter

Enters a mode in which sections are always shown with their (relative) number.

:Info: rapport.procs.pc:
7/2/84 .pc

Enters the "short-paragraphs" mode. In this mode, paragraphs longer than 4 lines are abbreviated to their two first lines.

:Info: rapport.procs.pl:
7/2/84 .pl

Enters the "long-paragraphs" mode. In this mode, all paragraphs are always fully shown.

:Info: rapport.procs.plan:
7/2/84 .plan

Shows the table of contents of the current document.

:Info: rapport.procs.sg:
7/3/84 .sg

Global string substitution. Make the substitution in the rest of the document, starting at the current position.

:Info: rapport.procs.sl:
7/3/84 .sl

Local string substitution. Make the substitution only inside the current part of the document.

:Info: rapport.procs.sl:
7/3/84 .sl

One string substitution. Make the substitution only at the next occurence of the old string.

:Info: rapport.procs.sq:
7/3/84 .sq

Query substitution on strings. Like .sl but on each occurrence of the string to be substituted asks for confirmation before doing the substitution.

:Info: rapport.procs.sc:
7/3/84 .sc

Scroll.

:Info: rapport.procs.more:
7/3/84 .more

The more command is very usefull to scroll on the current (i.e. enhanced) part of the text on a page per page basis.
It writes --more-- on the top of the screen and waits for an answer.
Answer <CR> to see next page,
or q. to quit the more command
After the q answer, or when the last page has been displayed,the next <CR> goes back to the point from where the more command were called.

:Info: rapport.procs.combiner:
7/3/84 .combiner

The current position must be a normal or special paragraph. The following paragraph must have the same type (i.e normal, indent, detail, telquel, programme). The .combiner command makes the two paragraphs into one of the same type.
(implemented by Barbara Staudt, july 1984)

:Info: rapport.procs.normalpar:
7/3/84 .normalpar

The current position must be a special paragraph,
(i.e normal, indent, detail, telquel, programme).
The .normalpar command makes it into a normal paragraph.
(implemented by Barbara Staudt, july 1984)

:Info: rapport.procs.emboiter:
7/3/84 .emboiter

The .emboiter command in useful to make one section a subsection of another.
It first asks for the section into which the subsection should be placed. Then it asks for the subsection. The subsection becomes the last subsection of the section.
(implemented by Barbara Staudt, july 1984)

:Info: rapport.procs.remonter:

7/3/84 .remonter

The .remonter command takes a subsection and puts it at the outermost level.
It prompts for the name of the section.
(implemented by Barbara Staudt, july 1984)

---

ada.procs.i

:Info: ada.procs:

ADA ENVIRONNEMENT

The Mentor procedures explained in this info segment are those written in the Mentol language. Other Mentor procedures, the system procedures, are described in the segment mentor.procs.info.

04/07/84 List of procedures available in MENTOR-ADA

(For more informations about a procedure during a mentor session, call the procedure '.details')

On_line Informations

.details	.ginfo	.list	.sysdetails

Navigation

.leftms	.next	.up

Modifications

.enter	.rename	.sort

Tree Traversal

.apl	.forall	.foreach

System Commands

.multics	.unix	.write

(on Multics System only)
For each one of these procedures, say .foo, there is an info segment called ada.procs.foo.info that you can see by typing the command:

help ada.procs.foo

:Info: ada.procs.forall:
04/07/84 .forall<@patt,.action>

Search all occurences of the pattern @patt, and apply the action .action on each. The search is done in all the text of the program starting at the current position. If the action .action fails on some occurence of the pattern the execution stops on that occurence.

Example:
.forall<@sch1,c @sch2> Will replace, from the current position, all occurences of @sch1 by @sch2. These two patterns must be defined before .

:Info: ada.procs.foreach:
04/07/84 .foreach<@patt,.action>

Same syntax and effect than .forall but the search is done only in the subtree whose root is the current position. Thus, .forall and .foreach are equivalent when starting at the top of the program.

:Info: ada.procs.apl:
04/07/84 .apl<.action>

Assumes that the current position is a list node (fails if it is not the case), and applies the action .action on all the elements of that list. Fails and stays here if .action fails on some elements.

:Info: ada.procs.enter:
04/07/84 .enter<@list,@object>

Add the object at the end of the list, but only if the object was not yet contained in the list. The type of the object must correspond with the type of the list elements.

Example :

```
u*;.foreach<@id,.enter<@listnames,@k>>
```

insert all the identifiers of the program in the list @listnames.

:Info: ada.procs.multics:
04/07/84 .multics

The procedure '.multics' is an interface between Multics and Mentor. It asks for a line. This line will be passed to Multics as a command line, and then, will be interpreted by Multics .
(Only on Multics System).

:Info: ada.procs.unix:
04/07/84 .unix

The procedure '.unix' is an interface between Unix and Mentor. It asks for a line. This line will be passed to Unix as a command line, and then, will be interpreted by Unix .
(Only on Unix System).

:Info: ada.procs.list:
04/07/84 .list

The procedure '.list' gives the list of all the procedures that are written in Mentol for the ADA environnement, and then all the predefined procedures of the Mentor system.

:Info: ada.procs.ginfo:
04/07/84 .ginfo

Gives some general informations about the Mentor system.
- On Multics, these informations are taken from the file mentor.gi.info. After calling the procedure .ginfo the user is inside the help processor of Multics. When the user lives this processor, by answering 'no' or reaching the end of the file, he comes back to his interrupted Mentor session.
- On Unix, these informations are taken from the file mentor.gi.i by the Unix command 'more' . The user can type a space to continue or 'q' to return to the mentor session.

:Info: ada.procs.write:
04/07/84 .write

This procedure may be used to save an ADA program . It saves the current program by overwriting or creating both files L.polish (L.po on Unix system) and L.ada , where L is a name that you give.

:Info: ada.procs.details:
04/07/84 .details

The procedure .details give some informations about all the procedures available in the ADA environnement (ie. procedures written in mentol, or predefined procedures of the Mentor system). The user has to enter the name of a procedure in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: ada.procs.sysdetails:
04/07/84 .sysdetails

The procedure .sysdetails give some informations about all the predefined procedures of the Mentor system. The user has to enter the name of a procedure

in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: ada.procs.leftms:
04/07/84 .leftms

Moves the current pointer to the first leaf (in preorder search) encountered in the current expression.

:Info: ada.procs.next:
04/07/84 .next

Moves the current pointer to the next node (according the preorder search).

:Info: ada.procs.sort:
04/07/84 .sort{<{.key},{@list}>}

Sorts the list, by default the current expression, according a certain key. The dot key argument denotes a path in the tree from list element node to the root of a sub_tree that will be taken as the key for sorting.

:Info: ada.procs.up:
04/07/84 .up{<@op>}

If called without any argument it is equivalent tu 'u'. When called with an argument '@op', goes up until an operator @op is reached (Attention : it is equivalent ot 'u*' if the operator @op cannot be reached).

Example : .up<@var>

:Info: ada.procs.rename:
04/07/84 .rename

Inconditionnally replace all occurences of an identifier by another in the current subtree.

---

mentorkit.i

How to introduce a user-defined langage inside Mentor-V5

B. Mélèse, D. Verove

INRIA
November, 13, 84

1- We assume that the name of the language you want to create and introduce under Mentor-V5 is "foo".

First, create a new directory in the directory .../mentor/public.
Copy is this directory all the files that are in .../mentor/mentorkit.

You have now in your directory .../mentor/public/foo the following files:

makefile — is the makefile that make a mentorkit for a language called "foo". You will have to modify it to change foo and FOO by the actual name of your language (in respecting the upper and lower cases of each occurence of foo or FOO).

yaccbib.c — is a library needed to make the parser. You do not have to modify it.

foo.c — will be the main program of the parser of foo. You do not have to modify it, just change its name.

SYSUSER.p (.po) — This is the Pascal module in which to wire the the pascal written unparser (see below) of your language. You just have to change FOO by the name of your language. There are three places where to make the change:

In the declaration of the procedure DECFOO
In the call of the procedure DECFOO
In the expression 'FOO ' . The actual name of your language must be no longer than 8 characters. In this expression it must by padded to 8 characters.

DECSKELETON.p (.po) : Is the skeleton of unparser to start with to write in Pascal the unparser of your language. Look at it (and see mentor.dec.i) , it is heavely commented.

2- Now, define your language in METAL, following instructions found in mentor/info/metal.i and mentor/info/metal.traps.i.

Examples of Metal programs may be found in the directory mentor/metal and in the file mentor/info/metal.i.

To develop and compile a Metal program you have to use the

version of MENTOR called GENERATOR which is the only version of Mentor-V5 containing the Metal compiler (see mentor.gi.i).

When entering GENERATOR answer "metaluser" when generator asks for "user name".

Documentation about the MENTOR-Metal environment can be found in mentor/info/metal.procs.i or can be read on line using the .list and .details commands during a GENERATOR session.

The compilation of a metal program under GENERATOR, by the command .compile, creates the following files (assuming the defined language is called FOO):

-- FOO.t : contains the abstract syntax tables

-- FOOCODE.t : contains the tree generation code

These two files (FOO.t and FOOCODE.t) are the files loaded by Mentor-V5 to work in language FOO. The user never have to look at them.

-- FOOKW.tol : It is a Mentol file to be loaded (by the .xin command) in a Mentor-pascal session. It contains a Pascal constant declaration part to be further inserted in the unparser of FOO (that has to be written in Pascal). In this constant declaration part, there is one constant declared for each OPERATOR and each KEYWORD of FOO. The integer values of these constants are the internal code of operators and keywords (see mentor.dec.i and section 2 below)

-- yaccfoo.y : The yacc definition of your language generated from the productions of the METAL program. The user does not have to modify this file: It is ready to be processed by yacc.

-- lexfoo.metal: A partial definition of the scanner of your language. The scanner will be generated using LEX. This file contains the tokens of the language as they appear in yaccfoo.y.
This file must be completed by the user by writing the regular expressions recognizing the generic tokens of the defined language (see the LEX manual).

3- Write in Pascal the unparser of your language.
(see mentor/public/mentorkit/DECSKELETON.p and mentor/info/mentor.dec.i)

The unparser can be (in fact MUST be) developped under Mentor-pascal starting from the segment mentor/public/mentorkit/DECSKELETON.po.

To access the keywords and the operators of your language from within the Pascal unparsing program you have to insert in the constant declaration part of this program the constant declaration part generated by the METAL compiler in the file FOOKW.tol. The file FOOKW.tol can be loaded under MENTOR-PASCAL by the command .xin. Loading this Mentol file creates a

Mentol variable called @kwlist that contains the constant declaration part defining the keywords of your language.

An example of unparser written from DECSKELETON can be found in mentor/public/asple under the name DECASPLE.p.

4- Then wire this unparser in the module mentor/public/mentorkit/SYSUSER.p (obvious by looking at this module).
To build your own SYSUSER.p take a copy of mentor/public/mentorkit/SYSUSER.po and modify it under MENTOR-Pascal.

5- If you are working on SM90, wire the parser of your language in SYSUSER.p. (forget about this point if you are working on VAX).

6- Modify the makefile found in mentor/public/mentorkit/makefile to take in accout your unparser (and your parser if you are working on SM90).
The parser will be created by calling LEX and YACC, these calls already exist in the mentorkit/makefile. Modify a copy of the file mentor/public/foo.c .

8- then use the "make" command to make a mentorkit for the language FOO.

Note 1 : the directory mentor/public/asple contains an example of a mentorkit for a user generated language called ASPLE.

- Its unparser is DECASPLE.p (and DECASPLE.po is its polish form)

- SYSUSER.p (and SYSUSER.po in polish form) is the file in which the ASPLE unparser is wired to be reachable by Mentor-V5.

- Its Metal definition is ASPLE.metal
(and ASPLE.po is its polish form)

- Its keywords are in ASPLEKW.tol

- Its YACC definition (generated by the metal compiler) is in yaccasple.y

- Its initial LEX definition is in lexasple.metal while its COMPLETE LEX definition (completed by hand) is in lexasple.l.

- asple.c is the main program of the YACC-LEX generated parser

- ASPLE is the parser generated from the YACC and LEX definitions. This parser will be called by mentorkit as a sub-process, thus, it must be reachable as any executable command.

- ASPLE.t and ASPLECODE.t are the tables to be used by MENTOR-ASPLE. The user never has to look at them.

- The makefile adapted for ASPLE is also in this directory.

Note 2 : By following these operation you will get a file called mentorkit, which is Mentor-V5 to which your language has been added. If you want

to add more than one language into Mentor-V5, the principle are the same except that instead of creating a new SYSUSER.p you just have to complete the one you have made for your first language.

Note 3 : Once a new language has been introduced this way in Mentor-V5, no Mentol environment exists for it. Such an environment will have to be written in Mentol by the implementor. However, it is possible to take advantage of the minimum language independant Mentol environment given in the Mentol library in the file COMMONENV.tol. The annotation environment (in the file ANNOTTOOL.tol) is also language independant. Thus A possible prelude file to start a Mentor-V5 session with a newly user defined language call "userlang" is:

```
%USERLANG
.xin<@commonenv>          loads the language independant mentol
                          environment COMMONENV.tol.
.xin<@annottool>          loads the annotation library
.p
.xin
```

Of course, all Mentor-V5 predefined commands also are language independant. A description of all these commands can be found in mentor.procs.i.

See multi-lang.i for more information on Mentor-V5 prelude files.

:Info: metal.procs:

METAL ENVIRONNEMENT

Bertrand Mélèse, Denis Verove
(October 1983)

The Mentor procedures explained in this info segment are those written in the Mentol language. Other Mentor procedures, the system procedures, are described in the segment mentor.procs.info.

10/11/83 List of procedures available in MENTOR-METAL

(For more informations about a procedure during a mentor session,
call the procedure '.details')

On_line Informations
.details .ginfo .list .sysdetails

Navigation
.abstract .chapt .fchapt .frule .leftms
.next .rule .up

Tree Traversal
.apl .forall .foreach .occur

Learning Mode
.create .menu

System Commands
.cnv .compile .multics .unix .write

Miscellaneous
.buildevery .enter .sort .version_number .comment_format

(on Multics System only)
For each one of these procedures, say .foo, there is an info segment called metal.procs.foo.info that you can see by typing the command:
help metal.procs.foo

:Info: metal.procs.forall:
11/18/80 .forall<@patt,.action>

Search all occurences of the pattern @patt, and apply the action .action on each. The search is done in all the text of the program starting at the current position. If the action .action fails on some occurence of the pattern the execution stops on that occurence.

Examples:

.forall<@rule,sl p>	Will print all syntactic production of the rules which are 'after' the current position in preorder
.forall<@schl,c @sch2>	Will replace, from the current position, all occurences of @schl by @sch2. These two schema must be defined before .

:Info: metal.procs.foreach:
11/18/80 .foreach<@patt,.action>

Same syntax and effect than .forall but the search is done only in the subtree whose root is the current position. Thus, .forall and .foreach are equivalent when starting at the top of the program.

:Info: metal.procs.apl:
11/18/80 .apl<.action>

Assumes that the current position is a list node (fails if it is not the case), and applies the action .action on all the elements of that list. Fails and stays here if .action fails on some elements.

:Info: metal.procs.enter:
10/11/83 .enter<@list,@object>

Add the object at the end of the list, but only if the object was not yet contained in the list. The type of the object must correspond with the type of the list elements.

Example :

```
u*;.foreach<@rule,.enter<@lnonterm,@k sl sl>>
```

insert all non_terminals of the program in the list @lnonterm.

:Info: metal.procs.multics:
11/18/80 .multics

The procedure '.multics' is an interface between Multics and Mentor. It asks for a line. This line will be passed to Multics as a command line, and then, will be interpreted by Multics .
(Only on Multics System).

:Info: metal.procs.unix:
11/18/80 .unix

The procedure '.unix' is an interface between Unix and Mentor. It asks for a line. This line will be passed to Unix as a command line, and then, will be interpreted by Unix .
(Only on Unix System).

:Info: metal.procs.list:
11/18/80 .list

The procedure '.list' gives the list of all the procedures that are written in Mentol for the Metal environnement, and then all the predefined procedures of the Mentor system.

:Info: metal.procs.ginfo:
11/18/80 .ginfo

Gives some general informations about the Mentor system.
- On Multics, these informations are taken from the file mentor.gi.info. After calling the procedure .ginfo the user is inside the help processor of Multics. When the user lives this processor, by answering 'no' or reaching the end of the file, he comes back to his interrupted Mentor session.
- On Unix, these informations are taken from the file mentor.gi.i by the Unix command 'more' . The user can type a space to continue or 'q' to return to the mentor session.

:Info: metal.procs.write:
03/05/81 .write

This procedure may be used to save a Metal program . It saves the current program by overwriting or creating both files L.polish (L.po on Unix system) and L.metal , where L is the name of your language.

:Info: metal.procs.compile:
03/05/81 .compile{<.compact>}

To compile a metal program. When using the '.compact' argument, the tables built by the metal compiler in the file LCODE.langtbl (LCODE.t on Unix) are generated in a compact form.

The parser MUST be re-created the first time the Metal program is compiled in "compact" mode that is using the command ".compile<.compact>" instead of ".compile" without parameters EVEN IF NO CHANGES HAVE BEEN MADE IN THE METAL PROGRAM.

:Info: metal.procs.cnv:
03/05/81 .cnv{<.compact>}

To compile a metal program. When using the '.compact' argument, the tables built by the metal compiler in the file LCODE.langtbl (LCODE.t on Unix) are generated in a compact form. Warning : this procedure does not make any verification on the phylum EVERY .

The parser MUST be re-created the first time the Metal program is compiled in "compact" mode that is using the command ".compile<.compact>" instead of ".compile" without parameters EVEN IF NO CHANGES HAVE BEEN MADE IN THE METAL PROGRAM.

:Info: metal.procs.details:
10/11/83 .details

The procedure .details give some informations about all the procedures available in the Metal environnement (ie. procedures written in mentol, or predefined procedures of the Mentor system). The user has to enter the name of a procedure in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: metal.procs.sysdetails:
10/11/83 .sysdetails

The procedure .sysdetails give some informations about all the predefined procedures of the Mentor system. The user has to enter the name of a procedure in lowercases without the prefixing dot. The list of all the procedures is obtained by '.list' .

:Info: metal.procs.leftms:
03/05/81 .leftms

Moves the current pointer to the first leaf (in preorder search) encountered in the current expression.

:Info: metal.procs.next:
03/05/81 .next

Moves the current pointer to the next node (according the preorder search).

:Info: metal.procs.sort:
03/05/81 .sort{<{.key},{@list}>}

Sorts the list, by default the current expression, according a certain key. The dot key argument denotes a path in the tree from list element node to the root of a sub_tree that will be taken as the key for sorting.

Examples :

```
f @rule_s; .sort<sl sl>
```

Reorganizes a rules list according the lexical order applied to the left hand_side productions non_terminals.

:Info: metal.procs.frule:
10/14/83 .frule

Asks for a non_terminal, then moves the current pointer on the next rule that has this non-terminal as left hand side of its production.
Does not move @k if that rule does not exist.

:Info: metal.procs.occur:
10/14/83 .occur

Asks for a non_terminal and shows all the rules in which this non_terminal occurs. The pointer @listoccur denotes the list of these rules.
Does not move the current pointer.

:Info: metal.procs.chapt:
10/14/83 .chapt

Goes up to the enclosing chapter. Does not move if fails

:Info: metal.procs.rule:

10/14/83 .rule

Goes up to the enclosing rule. Does not move if fails.

:Info: metal.procs.abstract:
10/14/83 .abstract

Goes at the 'abstract syntax' part associated with the current position. Does not move if such a part does not exist.

:Info: metal.procs.fchapt:
10/14/83 .fchapt

Asks for a chapter name and goes to it.
Does not move if this chapter does not exist.

:Info: metal.procs.up:
10/14/83 .up{<@op>}

If called without any argument it is equivalent tu 'u'. When called with an argument '@op', goes up until an operator @op is reached (Attention : it is equivalent to 'u*' if the operator @op cannot be reached).

Example : .up<@rule>

:Info: metal.procs.buildevery:
10/14/83 .buildevery

To build the EVERY phylum in a metal program. Used by the .compile and .cnv commands. In normal cases the user does not have to call this command.

:Info: metal.procs.create:
10/14/83 .create

To create a Metal program in a learning mode. In this mode, the creation of a metal program is driven by a menu system.
(to get this procedure you must first load the mentol file METALMENU.tol using the command .xin when you are inside mentor. The command .xin asks for a file name where you answer "metalmenu")

:Info: metal.procs.menu
10/14/83 .menu

To create a Metal program in a learning mode. In this mode, the creation of a metal program is driven by a menu system.
This is an EXPERIMENTAL learning mode .....

:Info: metal.procs.version_number:
11/15/84 .version_number

To create an annotation which contains the version number of the language

---

:Info: metal.procs.comment_format:
11/15/84 .comment_format

To create an annotation which contains the delimitors of comments of the language and the level of visible recursive comments

---

metal.traps.i

KNOWN TRAPS IN USING METAL

and new features of Metal

MENTOR-V5

Bertrand Mélèse
Nov, 12, 1984

In this info file are recorded known difficulties encountered by users when defining a new language in Metal, known bugs of Metal and limitations imposed by the Metal compiler on Metal programs.

Section 8 and 9 describe new features of Metal in Mentor-V5:
- How to specify a version number of a language defined in Metal
- How to specify unparsing format for comments.

This file should be carefully read by new users of Metal as well as the Metal user's manual which is in the file metal.i. Before trying to introduce a new language inside Mentor-V5, users should also read the info file mentorkit.i.

1-- Syntax of productions

1.1- The maximum number of non terminals in the right hand side of a production is 9. This is not checked during a Generator (i.e. Mentor-Metal) session. It is checked at compile time by the Metal compiler that send the error message

"Number of Non_terminals exeeds MAX_NON_TERMINAL : user error"

on productions with more that 9 non-terminals in their right-hand side. The total maximum size of a right-hand side of a production is 20 elements.

All limitations imposed by Metal are listing in section 7.

1.2- Terminals of the defined language whose names are also Metal reserved names or symbols must be preceded by a # sign. Other terminals can also be preceded by a # sign.

A list of Metal reserved names and symbols can be found in the Metal program defining Metal given in the metal user's manual (metal.i).

1.3- When a GENERIC appears in the right hand side of a production it MUST be the only element of this right hand side:

example:

(1) <ident>::= %IDENT ; is correct

(2) <call> ::= %IDENT (<params>) ; is NOT correct in Metal

and should be replaced by

(3) <call> ::= <ident> (<params>) ; and the production (1) above must be present.

***WARNING: This restriction is NEVER CHECKED by MENTOR-METAL or by the Metal COMPILER. It is a bug that I hope to remove some day .....

2-- Tree building functions

2.1- List matching

The matching of lists was not implemented in previous versions of Mentor. This section is then a presentation of this "new" feature of Mentor-V5.
The only case where list matching did run in previous versions is shown by the second example below. Take care of this example if you already have Metal programs running under Mentor version 4.n: The behaviour of the matching has changed in this case since there was a bug in previous versions that has been fixed in Mentor-V5.

Only ONE meta-variable that stands for a sub-list is allowed in patterns. The fact that a meta-variable stands for a sub-list or a single element is deduced by the Metal compiler from the position of that Meta-variable in the pattern.

When the pattern does not contain a sub-list meta-variable, the match can succeed only when the pattern and the list to be matched have EXACTLY the same number of elements.

When a meta-variable that stands for a sub-list has been bound to an empty sub-list, it CANNOT be re-used to build a new tree.

Examples:

(1)
```
let fool-list(X) = <NT>
    in foo2-list(X)
```

X is a sub-list meta-variable because it is the parameter of a "list" operator; it will be bound to the whole list denoted by <NT>.
This first example is thus a way to rename a list in Metal.
(see below the section about Renaming of lists)

```
(2)     let fool-list((X)) = <NT>
            in foo2-list((X))
```

X is meta-variable of element because in this example the parameter of the "list" operator is (X) which stands for a list with exactly one element. In this case, the match will succeed when the list denoted by <NT> has exactly ONE element.

**** WARNING: This point is a difference between the Metal compiler of Mentor-V5 and previous versions: In previous versions, this second example did match correct even if the matched list had more than one element. This WAS A BUG OF THE PREVIOUS VERSION. It has been fixed. Metal program that relies on this bug will no more run under Mentor Version 5 !!!!

```
(3)     let fool-post(X,Y) = <NT>
            in fool-pre(Y,X)
```

X is a sub-list meta-variable because it is the first parameter of a "post" operator. Y in an element meta-variable because it is the second parameter of a "post" operator. In this example, X will be bound to the sub-list composed of the list denoted by <NT> without its last element that will be bound to Y. This example takes a list and re-builds a list which is the first one in which the first element has been put at the end of the list.

If the list to be matched has only one element, X is bound to an empty sub-list and MUST NOT BE REUSED in the tree building function that follows the "in" keyword of a "let" operator or the "=>" keyword of a "when" alternative in a "case" statement of Metal.

If there is a risk of getting an empty sub-list in X, this Metal "let" statement must be re-written to trap this case separately as follows:

```
case <NT>
  when fool-list((X)) => fool-list ((X))
  when fool-post(X,Y) => fool-pre(Y,X)
end case
```

(4)

```
case <NT>
     when fool-list((X1,X2,X3,X4,X5,X6,X7))
         => fool-list((X7,X6,X5,X4,X3,X2,X1))
     when fool-list((X1,X2))
         => foo2-list((X2,X1))
     when fool-post(Y,binop(Z,string-atom('aaa')))
         => foo2-pre (binop(number-atom('333'),string-atom('bbbbb')),Y)
     when fool-post(Y,binop(number-atom('1000'),Z))
         => foo2-pre (binop(number-atom('2222'),Z),Y)
     when fool-pre (binop(X1,Y1),fool-post(Y,binop(X2,Y2)))
```

```
        => fool-post(Y,fool-pre (binop(X1,Y2),fool-list((binop(X2,Y1)))))
end case
```

This Metal "case" statment could have a meaning in a language that has the following operators defined in its abstract syntax:

"fool" and "foo2" are list operators
"string" is an atomic operator "implemented as STRING"
"number" is an atomic operator "implemented as INTEGER"
"binop" is a binary operator that can have a number as first son and a string as second son.

2.2- Renaming of list.

Since renaming of lists has to be done very often, Metal provides a short way to do it: The example (1) in the previous section could be written shortly as:

```
foo2-list(<NT>)
```

which means: Take the list denoted by the non-terminal <NT> and change its operator by foo2.

2.3- Specifying building of empty trees.

The Metal predefined operator "voidbid" can be used in the tree building functions to specify the construction of an empty tree. It is a nullary operator. It will NOT appear in the resulting trees: the Metal interpretor builds an empty tree in place of it. This operator must NOT appear in the definition of the abstract syntax of the language. Thus, the name "voidbid" is a reserved name of Metal.

Example: Consider the following Metal rule and suppose that foo has been defined as a binary operator.

```
<left>  ::= <rl> ;
    foo(voidbid,<rl>)
```

The tree built there is a binary tree with the operator foo. its first son is empty and its second son in the tree associated to the non-terminal <rl>.

The operator "voidbid" is useful to build empty parts of fixed arity operators. To build empty list, it is enougth to put an empty list as argument to the "list" constructor of Metal:

Example: Building of possibly empty lists (foo is now supposed to be a list operator)

```
<list1> ::= ;                       First case: the list can be empty
    foo-list(())
<list1> ::= <list1> <el1> ;         Second case: add an element at the
    foo-post(<list1>,<el1>)         end of the list.
```

3-- Definition of the Abstract syntax of the language:

3.1- The following operators MUST appear in the abstract syntax of the defined language:

```
meta     -> ;
comment  -> implemented as STRING ;
comments -> COMMENT * ... ;
```

as well as the following phylum

```
COMMENT  ::= comment ;
```

It is envisaged that the Metal compiler could generate these definitions automaticcaly in subsequent versions of METAL.

3.2- The node "meta" mentionned in the previous section MUST NOT BE the first atomic operator encountered in the abstract syntax of the language. Remember that the abstract syntax of the language is built by the Metal compiler by concatenating all the "abstract syntax" zones of the Metal program.
(see also the section about the phylum EVERY)

3.3- The FIRST atomic operator encountered in the abstract syntax of the language is used by Mentor-V5 to define various implicit Mentol variables of a Mentor-V5 session (like @UNDEF, @NULL, @USERNAME ...).
This first atomic operator should then be an operator "implemented as IDENTIFIER".
(see also the section about the phylum EVERY)

3.4- The following operator names are reserved and cannot be names of user defined operators:

meta, comment, comment_s, free, voidbid

4-- The phylum EVERY

A special phylum called EVERY is used by the Metal compiler to associate an internal code to each operator of the language defined by a Metal program.

When this phylum does not exist in a Metal program, it is automatically created by the Metal compiler. Then, in normal use, the phylum EVERY will be created by the Metal compiler the FIRST time this program has been compiled. Then, it remains in the Metal program in a separate "abstract syntax" zone.

The phylum EVERY is built by the Metal compiler as follows:

All operators defined in all "abstract syntax" zones of the Metal program are gathered in five distinct lists:

list1 contains all NULLARY operators
list2 contains all UNARY operators
list3 contains all BINARY operators
list4 contains all LIST operators
list5 contains all TERNARY operators

Inside each of these five lists, the order of operators is their order of definition in the Metal program.

The list of operators appearing in the definition of EVERY is the concatenation of these five lists. In the final list holes are inserted to mark the separation between arities. These holes appears in EVERY as operators called "free".

To be sure that the two conditions 3.2 and 3.3 are satisfied you just have to check what the first operator of the phylum EVERY is: It MUST NOT be "meta" and must be something implemented as IDENTIFIER. If it is not the case it is NOT sufficient to change that in the phylum EVERY: you have to change the order of definition of atomic operators in the "abstract syntax" zones of the Metal program, delete the "abstract syntax" zone created by the Metal compiler to contain EVERY and compile again your Metal program.

The changes to make in your "abstract syntax" zones are easy since the first operator that appears in the definition of the generated phylum EVERY will be the FIRST atomic operator encountered in these "abstract syntax" zones.

5-- Updating the phylum EVERY when changing a language definition

When you modify the definition of a Metal defined language, you can take profit of the phylum EVERY to keep compatibility between the previous versions and the new version of your language. This is the reason why this phylum is kept in the Metal program and not recomputed each time by the Metal compiler.

When a language has been used under Mentor-V5, polish files have been created and you want to be able to load them even if modifications have been made in the language definition. This will be possible only if the modifications made are PURE EXTENSIONS of the language: new abstract syntax operators or new rules have been added without changing existing ones.

If you make some actual modifications in your language, like for example, modifying the arity of an operator or canceling an operator, all the already existing polish files become unreadable in the new version of your language. In this case, you can always restart from their corresponding concrete representations using the .parse command instead of the .load command.

5.1- How to keep compatibility when adding new operators

The newly defined operators have to be added MANUALLY AT THE END of the list of operators defining the phylum EVERY, no matter what their arity is.

5.2- How to keep compatibility when adding new rules

Nothing special to do here. The compatibility of polish

files is not affected when new rules are added.
But, in this case the PARSER of the language has to be re-generated (using yacc and lex) to take in account the added concrete syntax features (see 6-- below)

If you are not concerned in keeping compatibility with old versions of your language, the best way to recreate it is in deleting by hand the "abstract syntax" zone that contains the phylum EVERY: the Metal compiler will then recreate it the next time your Metal program will be compiled.

5.3- Compiling a modified Metal program

If you get the error message

"the phylum EVERY may be inconsistent"

this means that you have added new operators in the abstract syntax of the defined language but they have not been added in the phylum EVERY. In this case you can add them by hand as explained in 5.1 above to keep compatilibily or delete this phylum to let the Metal compiler create it again.

5.4- Keeping compatibility with an already written unparser.

Another compatibility problem that arises when modifying a Metal definition is the compatibility of the unparser of the language. The unparser has to be written in Pascal (see mentorkit.i). In a Pascal written unparser, operators and keywords of the language are declared as constants of the Pascal program. The constant zone to insert in the Pascal unparser is generated by the Metal compiler (see mentorkit.i). When modifications made in your language modifie the number or THE ORDER of operators or KEYWORDS, you have to change this constant declaration part of your unparser by the new one generated by the Metal compiler during the last compilation of the Metal program.

6-- When does one have to re-create the parser of a language.

```
**********************************************************
* Each time a modification has been made in the definition *
*      of the concrete syntax of the language.           *
**********************************************************
```

-- by adding new rules

-- by modifying the production of existing rules

-- by cancelling rules

The parser MUST also be re-created the first time the Metal program is compiled in "compact" mode that is using the command ".compile<.compact>" instead of ".compile" without parameters (see description of the

.compile command in metal.procs.i) EVEN IF NO CHANGES HAVE BEEN MADE IN THE METAL PROGRAM.

It is useless to recreate the parser when only the abstract syntax of the language or the tree building functions have been modified.

In any case, the parser has to be re-created when you get the error message "incompatibility between MENTOR tables and PARSER tables" when trying to parse something in your language during a Mentor-V5 session in your language.

7-- Various limitations imposed by the Metal compiler

These limitations should not disturb users even when developping large languages in Metal. The limits are 20% larger than the needed size in the Metal program that defines Ada.

7.1- The maximum number of RULES in a Metal program is 770.

7.2- The maximum number of NON_TERMINALS in the right-hand side of a production is 9.

7.3- The maximum number of ELEMENTS (non_terminals + terminals) in the right-hand side of a production is 20.

7.4- The maximum number of TERMINALS in a Metal program is 200.

7.5- The maximum number OPERATORS in the abstract syntax of a language defined by a Metal program is 220.

7.6- The maximum number of NON SINGLETON PHYLA in the abstract syntax of a language defined by a Metal program is 100.
(a NON SINGLETON PHYLUM is a phylum that contains at least two operators)

7.7- The maximum number of SINGLETON PHYLA in the abstract syntax of a language defined by a Metal program is 64.
(a SINGLETON PHYLUM is a phylum that contains only ONE operator)

7.8- The maximun number of IMMEDIATE STRINGS is 70.
(IMMEDIATE STRINGS are possible immediate values given as parameter of the "atom" construct of Metal)

8-- Specifying the Version number of the defined language in Mentor-V5

In Mentor-V5 it is possible to specified a Version number of the language defined by the Metal program. This version number is recorded in the polish files allowing Mentor-V5 to check it when loading a polish file.

A polish file can then be loaded if the version number it contains not greater than the current version number of the language (the current version number of a language is given by Mentor-v5 when this language is loaded). This means that, increasing the version number of a language always keeps previous polish files loadable but decreasing this version number disallows the loading

of polish files that were created by the previous version of the language.

***WARNING : The fact that a polish file remains "loadable" when the version number has been increased just means that mentor will accept to try to load it. It DOES NOT MEAN that the corresponding tree will always remain meaningful !!! The corresponding tree remains meanungful ONLY if the compatibility between versions of the defined language is kept by taking care of remarks of previous sections 5.n.

The version number is specified in a special annotation called VERSION hooked at the root of the Metal program. (see annotation.i for explanation of what an annotation is) of the Metal program. It can be created or changed by using the command

.version_number

of the metaluser environment (see metal.procs.i).

9-- Specifying unparsing of COMMENTS of the defined language in Mentor-V5

In Mentor-V5, a minimal format for unparsing of comments of the defined language has to be specified. This format is kept in an annotation called ANNOT hooked to the definition of the "comment" node (see section 3.1 above) of the abstract syntax.

This annotation can be created interactively by calling the command

.comment_format

of the Metaluser environment. (see metal.procs.i)

This format contains:

The comment delimiters
The maximun visible recursive levels of comments.

Comments of level 1 are comments (or annotations of course) hooked directly to a node of the main tree. Comments of level n+1 are comments hooked to a tree which is a comment of level n.

Examples:

In Pascal the comments delimiters are (* and *) by default and the maximum visible recursive levels of comments is 1 because in most pascal compilers, nested comments are prohibited.

In Ada the comments delimiters are -- and EOL (end_of_line) and the maximum allowed level of comments is unlimited.

The ANNOT annatation described here is actually very minimum and is just a quick adaptation of Metal to cope with new features about comments and annotations needed in Mentor-V5. It will be developped from now on to become a complete annotation definition language.

For more information about comments, annotations and the multi-language facility of Mentor-V5, see the files annotation.i and multi-lang.i.

10-- A strange point: the operator "terminal_s_op"

This point concerns the abstract syntax of Metal itself as it is defined by the Metal program METAL.metal.

The operator "language" of the abstract syntax of Metal is TERNARY.

Its first son is the name of the defined language
Its second son is a list of zones.
Its third son is called "terminal_s_op".

Only the first two sons are useful for a user who defines a language in Metal using Mentor-V5 under Unix. The third son has been kept for compatibility with the version of Mentor-V5 running on Multics where the parser generator it uses needs specific information to optimize the error-recovery mechanism. This information is a list of terminals of the language and is given in the third son of the Metal program. This list can of course be empty and is in fact always empty as far as the Unix version in concerned.

When working on Unix, the only case where the user of Metal has to know about this ghost operator is when entering a Metal using the menu mode of input (i.e the .menu command: see beginners.i). The menu mode will ask of this third son at the end of the process of entering a Metal program. Put an empty list there if you are working on Unix.

mentor.dec.i

How to write an Unparser in Pascal for a new language defined in Metal

Bertrand Mélèse
(November 1984)

In the current state of the Mentor system, one has to write in Pascal the unparser of his/her new defined language. To do that he/she is advised to start from the unparser skeleton given below. The polish form of this skeleton, ready to be loaded under Mentor-pascal, will be found in the file DECSKELETON.po in the info directory of Mentor.

You will find examples of unparsers written from this skeleton in mentor/public/asple/DECASPLE.p and mentor/listing/DECRAPPORT.p which are the unparsers for the ASPLE and the RAPPORT language.

```
 (* All along this program skeleton, the name of your language
   will be refered as NAME
   ******************
   .LAST UPDATE :    .
   15 Nov84  11:11:47
   .BY :             .
   VALERIE
   ******************
   *)
module DECSKELETON (* change DECSKELETON by DECNAME*);
     const
          PRE      =0;
          POST     =1;
          FUNNY    =-87654321;
          IDENT16 =16;
          MAXSTRING=160;
          IDENTFONT=0;
          KWFONT   =1;
          COMMON   =4;
          COMMOFF =5;
          FLCOMMENT=2;
          MAXINAMES=2500 (*A good idea is to declare here a constant for
                           each node and each keyword of your language.
                           This part of the constants declaration zone
                           is generated during the compilation of the
                           Metal program defining your language in the
                           segment NAMEKW.tol . This file is a mentol file
                           that you can load (by .xin command) during a
                           MENTOR-Pascal session and that creates the
                           operators and keywords constants list under
                           the variable @kwlist. Then the constants can
                           be used in Case statements of procedures
                           DECLIST,DECTAO... and for Keyprint orders.
                           Constants identifiers are automaticaly
```

```
                    generated; for example constants for operators
                    are generated with the prefix 'op' and the name
                    of the operator without underscore character;
                    so conflicts with others identifiers may appear.*);
  (* Nothing to change in the type declaration part *)
 type ALFA    =packed array[1..8]of CHAR;
      OPERATORS=0..255;
      KINDS   =INTEGER;
      SMALLBUFS=array[1..IDENT16]of CHAR;
      BIGBUFS =array[1..MAXSTRING]of CHAR;
      OPERKIND=(KOPIDENT,KOPSTRING,KOPNUMBER,KOPVOID,KOPCHAR,KOPINT,KOPGATE
                ,KOPTRUC,KOPUNARY,KOPBINARY,KOPTERNARY,KOPSTARLIST,
                KOPPLUSLIST);
      TREES   =^INTEGER (* private type *);
      LANGUAGE=^INTEGER (* private type *);
      SYMBOL  =INTEGER;
      TTABNAMES=array[0..MAXINAMES]of SYMBOL;
  (* Nothing to change in the import declaration part *)
 import
      TABNAMES:TTABNAMES;
      CURRENTP:TREES (* tree denoted by the current pointer @k *);
 var  WASLETTER:BOOLEAN (* true if the last written character is a letter
       *);
      VISCOMM:BOOLEAN (* true if comments has to be unparsed *);
      WASCOM:BOOLEAN (*true if comment has been unparsed *);
      PADIDENT:INTEGER (* minimum size of identifiers output *);
      LOCALANG:LANGUAGE;
      BASEKEYWORD:INTEGER (* used to reach keywords symbols *);

  (* Returns true if the operator OP is a list operator *)
 function ISALIST(OP:OPERATORS;L:LANGUAGE):BOOLEAN;external;

  (* Return the structure of the operator OP *)
 function OPKINDOF(OP:OPERATORS;L:LANGUAGE):OPERKIND;external;

  (* Returns the value of the language KIND parameter *)
 function LGPARAM(LANGUE:LANGUAGE;KIND:INTEGER):INTEGER;external;

  (* Returns true is the FLAG is set *)
 function FLAGON(FLAG:INTEGER):BOOLEAN;external;

  (* In video mode, returns true when the screen is full;
     used in USERDECTABLE to stop unparsing *)
 function STOPDIS:BOOLEAN;external;

  (* Returns the code of the root operator of T *)
 function OPER(T:TREES):OPERATORS;external;

  (* The TREE L must be a list.
     Returns the first element of this list *)
 function HEAD(L:TREES):TREES;external;

  (* The TREE L must be a list.
     Returns a list which is L without its first element *)
 function TAIL(L:TREES):TREES;external;
```

```
 (* The TREE L must be a list.
    Returns true is this list is empty.*)
function ISEMPTY(L:TREES):BOOLEAN;external;

 (* Returns the N th son of the tree T *)
function CHILD(N:INTEGER;T:TREES;L:LANGUAGE):TREES;external;

 (* The first argument of STRINGOF must be an atomic tree,
    that is a tree with a nullary operator as root operator.
    It gives back the corresponding string and its length in B and LNG.
    Very usefull to get the atomic objects like identifiers, strings...*)
procedure STRINGOF(T:TREES;var B:BIGBUFS;var LNG:INTEGER;L:LANGUAGE);
 external;

 (* Takes a character an gives it back in upper cases *)
function UPPERC(CH:CHAR):CHAR;external;

 (* Takes a character and gives it back in lower cases *)
function LOWERC(CH:CHAR):CHAR;external;

 (* Takes a buffer and gives it back in lower cases *)
procedure LOWERB(var B:BIGBUFS;L:INTEGER);external;

 (* Takes a buffer and gives it back in upper cases *)
procedure UPPERB(var B:BIGBUFS;L:INTEGER);external;

 (* Returns in BUF the text of the symbol SYMB. Used in KEYPRINT *)
procedure PRNAME(SYMB:INTEGER;var BUF:SMALLBUFS;var LNG:INTEGER);external;

 (*Unparsing of comments *)
function DECCOMMS(WHERE:TREES;PREFIX:BOOLEAN;HOLO:INTEGER);external;

function HASVISCOMMS(WHERE:TREES;PREFIX:BOOLEAN):BOOLEAN;external;

 (* In video mode, to enter in bold fount *)
procedure BEGEMPH;external;

 (* In video mode, to come back in normal fount *)
procedure ENDEMPH;external;

 (* Writes a space separator *)
procedure GLUE;external;

 (* Begins a new line *)
procedure LINE;external;

 (* Marks the current column; the next line will start at that column *)
procedure MARK;external;

 (* To forget the previous MARK *)
procedure RELEASE;external;

 (* Tabulation from the current column.
    Will be effective on the next line *)
```

```
procedure TAB;external;

 (* Back tabulation: comes back to the previous position.
    Will be effective only on the next line *)
procedure BACKTAB;external;

 (* Writes a single character on the current output device *)
procedure CHARWRITE(CH:CHAR);external;

 (* Writes the buffer CH of length LNG on
    the current output device without splitting in font FONT.
    Use this procedure to write strings no longer than 16 *)
procedure WRITEON(CH:SMALLBUFS;N:INTEGER;FONT:INTEGER);external;

 (* Write the buffer BUF of length LNG on the current output
    device without splitting.
    PAD is the minimum size of the output. If  PAD < LNG  the
    size will be LNG characters. If  PAD > LNG  the size will
    be PAD characters long ending with PAD-LNG blank characters.
    PAD may be used to make alignments.
    Use this procedure to output buffer no longer than 160 *)
procedure BIGWRITE(var BUF:BIGBUFS;LNG,PAD:INTEGER);external;

 (* Procedure that writes a comment line
    on the current output device.*)
procedure PRINTCOMM(P:TREES);
     var  BB:BIGBUFS;
          L,I:INTEGER;
     begin
     STRINGOF(P,BB,L,LOCALANG);
     BIGWRITE(BB,L,PADIDENT)
     end;

procedure WLINE;
     begin
     WASLETTER:=FALSE;
     LINE
     end (*WLINE*);

 (* P may be any atomic tree. Its corresponding
    string will be written without splitting *)
procedure PRINTATOM(P:TREES);
     var  BB:BIGBUFS;
          L,I:INTEGER;
     begin
     STRINGOF(P,BB,L,LOCALANG);
     if WASLETTER then GLUE;
     BIGWRITE(BB,L,PADIDENT);
     WASLETTER:=TRUE
     end;

 (* P may be any atomic tree. Its corresponding
    string will be written without splitting but preceded by a given
    character ( here a $ ). Used here to write meta-variables *)
procedure PRINTMETA(P:TREES);
```

```
     var  BB:BIGBUFS;
          L,II:INTEGER;
     begin
     STRINGOF(P,BB,L,LOCALANG);
     if WASLETTER then GLUE;
     for II:=L+1 downto 2 do BB[II]:=BB[II-1];
     BB[1]:='$';
     BIGWRITE(BB,L+1,PADIDENT);
     WASLETTER:=TRUE
     end;

 (* To write a keyword. TOK is the constant declared for the keyword.*)
procedure KEYPRINT(TOK:INTEGER);
     var  BUFFER:SMALLBUFS;
          LNG:INTEGER;
     begin
     if WASLETTER then GLUE;
     PRNAME(TABNAMES[BASEKEYWORD+TOK],BUFFER,LNG);
     WRITEON(BUFFER,LNG,KWFONT) (*
       You could here set the variable WASLETTER to true for keywords
       ending with a letter: something like
        WASLETTER : = TOK in [....] *)
     end;

 (* To write holophrasting characters *)
procedure PRINTHOLO(OP:OPERATORS);
     var  BUFFER:SMALLBUFS;
          LNG:INTEGER;
     begin
     if WASLETTER then GLUE;
     if ISALIST(OP,LOCALANG) then
          begin
          BUFFER[1]:='.';
          BUFFER[2]:='.';
          BUFFER[3]:='.';
          LNG:=3
          end
     else begin
          BUFFER[1]:='#';
          LNG:=1
          end;
     WRITEON(BUFFER,LNG,KWFONT);
     WASLETTER:=TRUE
     end;

 (*USERDELTABLE is the unparsing procedure.
   *)
procedure USERDECTABLE(P:TREES;HOLO:INTEGER);

      (* Unparsing of list nodes *)
     procedure DECLIST(P:TREES);

           (* Vertical unparsing of a list node (exple: statements list).
              If the list needs a separator, the corresponding keyword
              must be put at the place of the meta-variable $SEPARATOR *)
```

```
        procedure VERTICDECLST(PTEMP:TREES);
             label
                  1;
             begin
             MARK;
             1: USERDECTABLE(HEAD(PTEMP),HOLO-1);
             PTEMP:=TAIL(PTEMP);
             if PTEMP<>nil then
                  begin
                  KEYPRINT($SEPARATOR);
                  if HOLO<>1 then WLINE;
                  goto 1
                  end;
             RELEASE
             end;

        (*
         Horizontal unparsing of a list node (exple: identifiers list).
         If the list needs a separator, the corresponding keyword
         must be put at the place of the meta-variable $SEPARATOR *)
        procedure HORIZDECLST(PTEMP:TREES);
             label
                  1;
             begin
             MARK;
             1: USERDECTABLE(HEAD(PTEMP),HOLO-1);
             PTEMP:=TAIL(PTEMP);
             if PTEMP<>nil then
                  begin
                  KEYPRINT($SEPARATOR);
                  goto 1
                  end;
             RELEASE
             end;

    (* Body of DECLIST *)
        begin
        case OPER(P) of
              (* Put here a case for each list node *)
             $LISTNODE1: $ACTION1 (*
               Is a call to a list unparsing procedure such as
               VERTICDECLST or HORIZDECLST *)
        end
        end (*DECLIST*);

    (* Unparsing of nullary nodes *)
    procedure DECTAO(var P:TREES);
        begin
        case OPER(P) of
              (* Put here a case for each nullary arity operator.*)
             $NULLARYNODE1: $ACTION1
        end
        end (*DECTAO*);

    (* Unparsing of unary nodes *)
```

```
    procedure DECTA1(var P:TREES);
         begin
         case OPER(P) of
               (* Put here a case for each unary node *)
              $UNARYNODE1: $ACTION1 (*

              Contains recursive call to USERDECTABLE(CHILD(1,P),HOLO-1)
                and writing commands such as keyprint, line, tab...
                Look at DECASPLE.p for examples *)
         end
         end (*DECTA1*);

     (* Unparsing of binary nodes *)
    procedure DECTA2(var P:TREES);
         begin
         case OPER(P) of
               (* Put here a case for each binary operator*)
              $BINARYNODE1: $ACTION1
         end
         end (*DECTA2*);

     (* Unparsing of ternary nodes *)
    procedure DECTA3(var P:TREES);
         begin
         case OPER(P) of
               (* Put here a case for each ternary operator.*)
              $TERNARYNODE1: $ACTION1
         end
         end (*DECTA3*);

(*Body of USERDECTABLE*)
    begin
    if(P<>nil)and not STOPDIS then
         begin
         if P=CURRENTP then BEGEMPH;
         if HOLO>0 then
              begin
              if VISCOMM then
                   begin
                   WASCOM:=DECCOMMS(P,TRUE,HOLO);
                   if WASCOM then WLINE
                   end
              else WASCOM:=FALSE;
              case OPKINDOF(OPER(P),LOCALANG) of
                   KOPIDENT,KOPSTRING,KOPNUMBER,KOPVOID,KOPCHAR,KOPINT,
                    KOPGATE,KOPTRUC: DECTA0(P);
                   KOPUNARY: DECTA1(P);
                   KOPBINARY: DECTA2(P);
                   KOPTERNARY: DECTA3(P);
                   KOPSTARLIST,KOPPLUSLIST: DECLIST(P)
              end;
              if VISCOMM then
                   begin
                   if HASVISCOMMS(P,FALSE) then WLINE;
                   WASCOM:=DECCOMMS(P,FALSE,HOLO);
```

```
                    if WASCOM then WLINE
                    end
               else WASCOM:=FALSE
               end
          else  (* Holophrast *)
               PRINTHOLO(OPER(P));
          if P=CURRENTP then ENDEMPH
          end
     end (*USERDECTABLE*);

  (*This procedure will be the entry point of your unparser.
    Then, choose its name (with the name of your language in it
    to avoid name conflicts).
    The only change you have to make in the body is to change
    USERDECTABLE by the name you have chosen for the unparsing
    procedure if you changed it.*)
procedure def YOURLANGDECOMP(X:TREES;HOLOPH:INTEGER;L:LANGUAGE);
     begin
     WASLETTER:=FALSE;
     PADIDENT:=0;
     VISCOMM:=FLAGON(FLCOMMENT);
     LOCALANG:=L;
     BASEKEYWORD:=LGPARAM(LOCALANG,1);
     if HOLOPH=-1 then HOLOPH:=10000 (* P* COMMAND*);
     USERDECTABLE(X,HOLOPH)
     end;
```

interface.i

TREEFRONT.p

How to use the Mentor interface

Bertrand Mélèse

November 1984

The Mentor interface is in the files TREEFRONT.p (the Pascal code) and TREEFRONT.po (the polish form). Have a look at this program: it is heavily commented and self-contained.

In these files are declared some Pascal types, procedures and functions header through which it is possible to manipulate the internal Mentor data structures.

The implementation of most of the procedures and functions of TREEFRONT.p is in the module TREEFACE.p (.po). In principle, it is useless to known about this implementation to use the interface.

Elements of the interface are used, for example, in the unparsers written for the unparser skeleton DECSKELETON.p (.po).

1-- Procedures and functions found in the interface permit to:

-- Obtain trees associated with Mentol variables of the current Mentor session (GETVARIABLE, RECEIVE ...)

-- Associate trees to Mentol variables (SEND ...)

-- Build new trees in a given abstract syntax
(all MAKE functions, PRE, POST APPEND ...)

-- Take sub-trees and move around in a tree (CHILD, FATHER, TAIL, HEAD ...)

-- Get the printname of atomes (STRINGOF ...)

-- ....

2-- Types

The type VARIABLE corresponds to Mentol variables

The type TREES corresponds to Mentor trees

The type LANGUAGE corresponds to the internal representation of languages

```
(*******************
  .LAST UPDATE :    .
  14 Nov84  10:49:04
  .BY :             .
  BERTRAND
  *******************
  *)
module TREEFRONT;
     const
          MAXSTRING=160;
     type PRIVATE =^INTEGER;
          OPERATORS=0..255;
          BIGBUFS =array[1..MAXSTRING]of CHAR;
          OPERKIND=(KOPIDENT,KOPSTRING,KOPNUMBER,KOPVOID,KOPCHAR,KOPINT,KOPGATE
                    ,KOPTRUC,KOPUNARY,KOPBINARY,KOPTERNARY,KOPSTARLIST,
                    KOPPLUSLIST);
          TREES    =PRIVATE (* private type *);
          LANGUAGE=PRIVATE (* private type *);
          VARIABLE=PRIVATE;
          SYMBOL   =INTEGER;

      (* Returns true if the operator OP is a list operator *)
     function ISALIST(OP:OPERATORS;L:LANGUAGE):BOOLEAN;external;

      (* Return the structure of the operator OP *)
     function OPKINDOF(OP:OPERATORS;L:LANGUAGE):OPERKIND;external;

      (* Returns the code of the root operator of T *)
     function OPER(T:TREES):OPERATORS;external;

      (* The TREE L must be a list.
        Returns the first element of this list *)
     function HEAD(L:TREES):TREES;external;

      (* The TREE L must be a list.
        Returns a list which is L without its first element *)
     function TAIL(L:TREES):TREES;external;

      (* The TREE L must be a list.
        Returns true is this list is empty.*)
     function ISEMPTY(L:TREES):BOOLEAN;external;

      (* Returns the N th son of the tree T *)
     function CHILD(N:INTEGER;T:TREES;L:LANGUAGE):TREES;external;

      (* The first argument of STRINGOF must be an atomic tree,
        that is a tree with a nullary operator as root operator.
        It gives back the corresponding string and its length in B and LNG.
        Very usefull to get the atomic objects like identifiers, strings...*)
     procedure STRINGOF(T:TREES;var B:BIGBUFS;var LNG:INTEGER;L:LANGUAGE);
      external;

      (* Takes a character an gives it back in upper cases *)
```

```
function UPPERC(CH:CHAR):CHAR;external;

 (* Takes a character and gives it back in lower cases *)
function LOWERC(CH:CHAR):CHAR;external;

 (* Takes a buffer and gives it back in lower cases *)
procedure LOWERB(var B:BIGBUFS;L:INTEGER);external;

 (* Takes a buffer and gives it back in upper cases *)
procedure UPPERB(var B:BIGBUFS;L:INTEGER);external;

 (*Returns the Mentol variable of name NAME*)
function GETVARIABLE(var NAME:BIGBUFS;SIZE:INTEGER):VARIABLE;external;

 (*Makes the Mentor variable V denote the tree T*)
procedure SEND(V:VARIABLE;T:TREES);external;

 (*Returns the tree denoted by the Mentol variable V*)
function RECEIVE(V:VARIABLE):TREES;external (*



 The function EXAMPLE shows how to get the tree associated
 to a Mentol variable, and how to attach a tree to a Mentol
 Variable

 The four first assignments are clear: they build a BUFF containing
 characters and a LNG that is the lenght of BUFF

 Then, TFOO becomes the tree denoted by the Mentol variable @FOO,
 and the call to procedure SEND puts back that tree in the
 Mentol variable @FOO


 *) (*
function EXAMPLE:SOMETHING;
     var  BUFF:BIGBUFS;
          LNG:INTEGER;
          TFOO:TREES;
     begin
     BUFF[1]:='F';
     BUFF[2]:='O';
     BUFF[3]:='O';
     LNG:=3;
     TFOO:=RECEIVE(GETVARIABLE(BUFF,LNG));
     ALOHOFCODE;
     SEND(GETVARIABLE(BUFF,LNG),TFOO)
     end*);

 (*Builds an EMPTY tree*)
function NULLTREE:TREES;external;

 (*Builds an atomic tree (implemented as IDENTIFIER) with the
   operator OP and the value found in NAME. SIZE is the
   number of characters in NAME*)
```

```
function ATOMIDENT(OP:OPERATORS;var NAME:BIGBUFS;SIZE:INTEGER):TREES;
 external;

 (*Builds an atomic tree (implemented as STRING) with the
   operator OP and the value found in NAME. SIZE is the
   number of characters in NAME*)
function ATOMSTRING(OP:OPERATORS;var NAME:BIGBUFS;SIZE:INTEGER):TREES;
 external;

 (*Builds a unary tree with operator OP (that must be unary)
   and s1 as son*)
function MAKEUNARY(OP:OPERATORS;S1:TREES):TREES;external;

 (*Builds a binary tree with operator OP ands sons s1 and s2*)
function MAKEBINARY(OP:OPERATORS;S1,S2:TREES):TREES;external;

 (*Builds a ternary tree with OP and sons s1, s2 s3 *)
function MAKETERNARY(OP:OPERATORS;S1,S2,S3:TREES):TREES;external;

 (*Builds a list tree with operator OP (that must be a list
   Operator) and 1 element T *)
function MAKELIST(OP:OPERATORS;T:TREES):TREES;external;

 (*Adds T in front of the list L*)
function PRE(T,L:TREES):TREES;external;

 (*Adds T at the end of the list L*)
function POST(L,T:TREES):TREES;external;

 (*Appends the lists L1 and L2 *)
function APPEND(L1,L2:TREES):TREES;external;

 (*Returns the tree associated to the father of the tree T *)
function FATHER(T:TREES;LOCALANG:LANGUAGE):TREES;external;
```

toggs–dials.i

DIALS and TOGGLES

Denis Verove
November 1984

The behaviour of some Mentor commands and functions can be modified interactively in changing values of system parameters.

There are two kinds of system parameters :

- Dials
  These are integer parameters of Mentor. They are defined by a name, an internal code, a default value and a range of allowed variations.

- Toggles (or Flags)
  These are boolean parameters of Mentor. They are defined by a name, an internal code and a default value (set, or reset).

1 - ON LINE HELP ABOUT DIALS AND TOGGLES.

Four help keywords of the Mentor '.help' command concern dials and toggles :

- The keyword 'dials' lists the names of dials.

- The keyword 'toggles' lists the names of toggles.

- The keyword 'togs_on' lists the names of those toggles that are set, that is toggles whose current value is 'true' .

- The keyword 'togs_off' lists the names of those toggles that are reset, that is toggles whose current value is 'false' .

2 - MENTOR COMMANDS ON DIALS AND TOGGLES.

- The command '.test' asks for a dial or toggle name. When a dial name is answered, the command sends a message containing the current value of the dial. When a toggle name is answered, the command succeeds if the toggle is set, or fails if the toggle is reset.

- The command '.set' asks for a dial or a toggle name. When the given name is a dial name, the user is requested for the new

value to give to this dial. When the answered name is a toggle name, this toggle is set.

- The command '.reset' asks for a dial or a toggle name. When the given name is a dial name, the command assigns as current value to the dial its initial default value. When the answered name is a toggle name, this toggle is reset.

3 - THE MENTOR DIALS.

All the Mentor dials are listed there with a short explaination of their functions. For each dial information given is :

- its name
- its internal code
- its default value
- its range of authorized values

Dial names followed by USER are dials that any user can modify. Dial names followed by EXPERT are dials that are reserved to expert users. Dial names followed by IMPLEMENTOR are dials that are reserved to Mentor implementors.
Modifying EXPERT or IMPLEMENTOR dials is not save and not encouraged: they tune internal file coding and symbol table implementation. The modification of their values will result in sensible modifications in the behaviour of the system.
EXPERT and IMPLEMENTOR reserved dials are listed there only for completness of the Mentor documentation.

ALFAMAX, LINEMAX, LINECUT, IDENTMAX, TERMTYPE, POLMAXID dials are not currently used and are reserved for a future use.

-- ALFACUT - Implementor

internal code	5		
default value	8		
constraint range	1	-	160

Modifies the be. aviour of the '.coerce' command.
When using the coerce command with a list operator as first argument, ALFACUT gives the size of the strings in the resulting list of atomic elements. Example (in Pascal) :

```
@k c &
line]
aaaabbbbccccddddeeeeffffgggg
.coerce<@lline,@k,@v>;@v p
aaaabbbb
```

```
ccccdddd
eeeeffff
gggg
```

-- ALFASIZE - Implementor

internal code	3
default value	8
constraint range	1 - 160

Has an effect only when the ALFA8 toggle is set (ALFA8 is reset by default). Indicates the maximum size of string atomic values. When ALFA8 is set, strings are truncated during tree generation at the ALFASIZE length.

-- COMLEVEL - User

internal code	60
default value	32000
constraint range	0 - 32000

Maximum number of annotations level displayed by unparsing command. See annotation.i and multi-lang.i .

-- HRECALER - User

internal code	55
default value	0
constraint range	0 - 32000

Modifies the behaviour of the Mentor display algorithm.
When the current expression is moved out of the text displayed on the screen, the display procedure automatically computes a new video context. The value of this dial is the number of levels above the current expression in the tree from which the redisplay procedure starts.
Must be used carefully. Giving large values to HRECALER make the Mentor display both more clever and more expensive.

-- IDENTCUT - Implementor

internal code	9
default value	8
constraint range	1 - 160

Modifies the behaviour of the '.coerce' command.
When using the coerce command with a list operator as first argument, gives the size of the identifiers in the resulting list of atomic elements. Example (in Pascal) :

```
@k c &
```

```
line]
aaaabbbbccccddddeeeeffffgggg
.coerce<@lident,@k,@v>;@v p
(aaaabbbb,ccccdddd,eeeeffff,gggg)
```

-- IDENTPAD - User

internal code	10
default value	8
constraint range	0 - 32000

Modifies behaviour of unparsers.
This dial is used in unparsers to align vertical lists. It gives the minimal padding width of identifiers.
In the Pascal unparser, it is used to make alignments in the constant and type declaration parts.
In the Metal unparser, it is used to make alignments on right hand sides of productions and on right hand sides of definitions of operators and phyla in "abstract syntax" zones. Its value is changed to 19 in the Metal standard environment.

-- MARGIN - Expert

internal code	2
default value	20
constraint range	0 - 32000

The value of MARGIN dial is used by the currently used prettyprinter. When the successive prettyprinting indentations have reduced the useful part of a line to a length smaller than the value of MARGIN, horizontal wrap-round mechanism is simulated, cancelling temporarely existing indentation.

-- MORE - User

internal code	61
default value	32000
constraint range	0 - 32000

In teletype mode, commands that take long outputs (p , p-2 , .help) stop every N output lines (where N is the MORE value), waiting for a user answer to continue. The possible answers are :

- carriage return (to see the N next lines)
- an integer I (to see the I next lines)
- a quit command (0 or any string beginning with n or q) to stop the command.

-- POLLMAX - Implementor

internal code	15
default value	1

constraint range 0 - 32000

Maximum language codes allowed for current polish file.

-- POLSBMAX - Implementor

internal code 13
default value 500
constraint range 0 - 32000

Size of the symbol coding tables in polish files.

-- POLSYMBOLS - Implementor

internal code 12
default value 1
constraint range 0 - 2

Index to method used for encoding symbols in polish files.

-- POLWIDTH - Implementor

internal code 63
default value 80
constraint range 4 - 32000

Line size in polish files.

-- TAB - User

internal code 1
default value 5
constraint range 0 - 32000

Modifies the behaviour of the display by indicating the indentation size.
Set to 3 in the Rapport standard environment.
Set to 4 in the Metal standard environment.

-- WIDTH - User

internal code 0
default value 80
constraint range 1 - 32000

Modifies the behaviour of the display. The value of WIDTH is the line length.
This dial has an effect only when the WIDELINE toggle is reset (which is the default).
The line length used by the Mentor display is the minimum between the

value of WIDTH and the "co" termcap parameter of the user terminal. The major use of this dial is to make listings with a convenient line length.

4 - THE MENTOR TOGGLES.

All the Mentor toggles are listed there with a short explaination of their functions. For each toggle information given is :

- its name
- its internal code
- its default value (set or reset)

Toggle names followed by USER are toggles that any user can modify. Toggle names followed by EXPERT are toggles that are reserved to expert users. Toggle names followed by IMPLEMENTOR are toggles that are reserved to Mentor implementors.
Modifying EXPERT or IMPLEMENTOR toggles is not save and not encouraged : they tune internal file coding and symbol table implementation. The modification of their values will result in sensible modifications in the behaviour of the system.
EXPERT and IMPLEMENTOR reserved toggles are listed there only for completness of the Mentor documentation.

F57, HEXASH, INITPROC, SKIPEOF, WPROTECT toggles are not currently used, but are reserved for future use.

Toggles are regrouped in sections according to their functionality. Toggles listed in the first four sections are language independant. Toggles listed in the last three sections affect parser or unparser of the specified language.

Section 4.1 lists toggles whose value affects the behaviour of the Mentor display functions.

Section 4.2 lists toggles whose value affects the tree generation.

Section 4.3 lists toggles usable to trace commands execution.

Section 4.4 lists toggles that concern polish files handling.

Section 4.5 lists toggles available in Mentor-Pascal.

Section 4.5 lists toggles available in Mentor-Metal.

Section 4.5 lists toggles available in Mentor-Rapport.

4.1 - DISPLAY FUNCTION TOGGLES

-- COMMENT - User

internal code 2
default value set

Comments and annotations are visible only when the COMMENT toggle is set.

Affects also tree copying operation : if COMMENT is reset then the tree obtained by copy is the initial tree without any annotations.

-- PHOTOCMP - User

internal code 42
default value reset

PHOTOCMP can be used in other languages than Rapport, to generate TROFF input text with different fonts for identifiers, keywords and comments.

PHOTOCMP is also used in Mentor-Rapport.

-- PLOTTER - User

internal code 43
default value reset

Used only when PHOTOCMP is reset.
When PLOTTER is set, the unparsed text is in input format suitable for HP-7220 Plotter, with different colors for identifiers, keywords and comments.

-- SHOWMESS - Expert

internal code 48
default value reset

When SHOWMESS is set, executable fragments of messages are displayed at the current cursor position between two vertical bars, and they are not sent to the operating system command interpreteur.
See mentor.mess.i for more informations about the Mentor messages system.

-- SMARTDIS - Expert

internal code 47
default value reset

When SMARTDIS is set, backspace characters are interpreted by display functions.

-- VERBOSE - Expert

internal code 61
default value reset

When Mentor is used in a non_interactive mode, outputs (such as system requests, messages, input entry points, ...) are written on the output file only when VERBOSE is set.

VERBOSE is also used in interactive mode to control messages sending. When VERBOSE is set, messages are interpreted without testing any system toggle (see mentor.mess.i).

-- WIDELINE - User

internal code 10
default value reset

When WIDELINE is set, maximum size of output lines is 129 characters. WIDELINE is only used when output is performed on permanent files (with the '.unparse' command) to make listings.

4.2 - TREE BUILDING TOGGLES

-- ALFA8 - Implementor

internal code 62
default value reset

When set, strings are truncated at a length defined by the value of the ALFASIZE dial.
Operators used for internal representation of those strings must be implemented as strings (in their Metal definition).

-- COMMENT - User

internal code 2
default value set

Affects tree copying operation. If COMMENT is reset, the tree obtained by copy is the initial tree without any comment or

annotation.

COMMENT is also used in unparsers to make comments and annotations visible when it is set, or invisible when it is reset.

-- ECOMMENT - User

internal code 3
default value reset

When ECOMMENT is set, instantiation of schemas attaches the name of substituted metavariables as prefix comment of corresponding sub_tree in the resulting schema.

4.3 - TRACING TOGGLES

-- DEBUG - Implementor

internal code 0
default value reset

When set, Mentor runs in debug mode. Debugging messages are displayed on the terminal.

-- ECHO - Expert

internal code 4
default value reset

When ECHO is set, Mentor commands are echoed. Useful to debug mentol files when mentol syntax errors are reported at loading.

-- INTERRUP - Implementor

internal code 8
default value reset

When an interrupt occurs due to the use of the break key, the toggle INTERRUP is set and the interrupt routine is executed. If a new break key interrupt occurs when INTERRUP is set, the default interrupt routine '.user' is executed. Thus if the interrupt routine loops, it it is still possible to break the loop with a second break key. The user can reset INTERRUP during execution of the interrupt routine in order to disable this protection mechanism.
See mentor.wr_pr.i for more informations about interrupt routine and tracing facility.

-- NOSTEP - Implementor

internal code 7
default value set

Inhibits the tracing facility of Mentor (see mentor.wr_pr.i) when NOSTEP is set.

-- POSTSTEP - Implementor

internal code 6
default value reset

When POSTEP is set, the interrupt routine is called after executing each mentor statement. See mentor.wr_pi.i for informations about tracing facility.

-- PRESTEP - Implementor

internal code 5
default value set

When PRESTEP is set, the interrupt routine is called before executing each mentor statement. See mentor.wr_pr.i for informations about tracing facility.

-- STEPETAT - Implementor

internal code 63
default value set

When an interrupt occurs in the execution of a mentor program, some information is printed on the terminal if the flag STEPETAT is set. This information consists of :

- the nature of the interrupt (prestep, poststep, use of break key)
- whether the last statement excuted succeded or failed (and the nature of the failure)
- when in poststep, the number of compound statement still to be exited if an exit statement $n has been executed.

-- STEPOK - Implementor

internal code 60
default value reset

The toggle STEPOK is used in the tracing facility of Mentor (see the info file mentor.wr_pr.i). Before the execution of the interrupt routine, which is activated then the toggles PRESTEP or POSTSTEP are set, the flag STEPOK is set if and only if the instruction executed just before was successfull. The user can change stepok during the

interrupt routine so as to transmit either success or failure to the instruction executed after the interrupt.

4.4 - POLISH FILES TOGGLES

-- F39 - Implementor

internal code 39
default value reset

When F39 is set, Mentor performs statistics on polish files.

-- MONOLANG - User

internal code 45
default value set

When MONOLANG is set, a warning message is displayed on the screen when the user loads a polish file in which a specified language is not the current language.

-- XVERSION - Implementor

internal code 46
default value reset

Used to test the polish coding versions.
When XVERSION is set and the Mentor version is i.j , the polish files are created in version i.j+1 .

4.5 - MENTOR-PASCAL TOGGLES

-- AMPERBAR - User

internal code 21
default value reset

Affects boolean operators unparsing.
When AMPERBAR is set and ANDOR is reset, boolean operators ('or' and 'and') are unparsed as vertical bar (|) and ampersand (&) respectively . Otherwise, keywords 'or' and 'and' are used.

-- ANDOR - User

internal code 20
default value reset

Affects boolean operators unparsing.
When ANDOR is reset and AMPERBAR is set, boolean operators ('or' and 'and') are unparsed as vertical bar (|) and ampersand (&) respectively . Otherwise, keywords 'or' and 'and' are used.

-- AROB - User

internal code 16
default value set on MULTICS , reset on VAX and SM90

Affects pointer dereferencing symbol unparsing.
When AROB is set and UPARROW is reset, pointer dereferencing symbol is unparsed as '@' . Otherwise it is unparsed as '^' .

-- BRACE - User

internal code 12
default value reset

When BRACE is set, `}` is interpreted as right delimiter of comments in input mode. So input mode with [1line] entry point ends at first `}` encountered character. See also PERCENT and PARSTAR.

-- DECHEAD - User

internal code 32
default value reset

When DECHEAD is set, program header has the folowing synctatical form during parsing and unparsing :

```
program NAMEPROG , PROGPAR1 , PROGPAR2 , PROGAPR3
```

When reset, the equivalent usual form is :

```
program NAMEPROG(PROGRPAR1,PROGRPAR2,PROGPAR3)
```

-- HEXAMOT - User

internal code 41
default value reset

Affects hexadecimal constants unparsing.
Used only when the toggles OCTALB, HEXAX and HEXAQQ are reset. When HEXAMOT is set, the hexadecimal constants are unparsed as 16#FFFF . When HEXAMOT is reset, they are unparsed as 'FFFF'X .

-- HEXAQQ - User

internal code 26
default value reset

Affects hexadecimal constants unparsing.
Used only when the toggles OCTALB and HEXAX are reset. When HEXAQQ is set, hexadecimal constants are unparsed as "FFFF . When HEXAQQ is reset and HEXAMOT is set, they are unparsed as 16#FFFF . When HEXAQQ is reset and HEXAMOT is reset, they are unparsed as 'FFFF'X .

The HEXAQQ toggle is also used in Mentor-Metal.

-- HEXAX - User

internal code 25
default value set

Affects hexadecimal constants unparsing, only when OCTALB is reset. When HEXAX is set, hexadecimal constants are unparsed as 'FFFF'X . When HEXAX is reset and HEXAQQ is set, they are unparsed as "FFFF . When HEXAX and HEXAQQ are reset and HEXAMOT is set, they are unparsed as 16#FFFF . When HEXAX, HEXAQQ and HEXAMOT are reset, they are unparsed as 'FFFF'X .

-- IDLC - User

internal code 40
default value reset

When IDLC is set, identifiers are unparsed in lowercases.

Used in others unparsers of Mentor.

-- IDSTAR - User

internal code 36
default value reset

When IDSTAR is set, the Pascal parser accepts constructions such as NAME* in identifiers lists.

-- INFSUP - User

internal code 14
default value set on VAX and SM90 , reset on MULTICS

Affects inequality operator unparsing.
When INFSUP is set , inequality operator is unparsed as <> .
When INFSUP is reset and SHARP is set, it is unparsed as # .
When INFSUP and SHARP are reset and TILDEQ is set, it is unparsed

as ~= .
When INFSUP , SHARP and TILDEQ are reset, it is unparsed as <> .
The symbol ~= is never allowed by the Pascal parser.

-- INIVALUE - User

internal code 35
default value set

When INIVALUE is reset, "value" is parsed as an identifier, so value initialization zones are not allowed.

-- KWUC - User

internal code 9
default value reset

When KWUC is set, keywords are unparsed in uppercases.

Used also in others unparsers of Mentor.

-- METAKEY - Expert

internal code 31
default value reset

When METAKEY is set, metavariables are unparsed without '$' prefix and are affected by the KWUC toggle as pascal keywords.

METAKEY is also used in Mentor-Metal.

-- MODULSEM - User

internal code 33
default value set

When MODULSEM is set, a semicolon is added at the end of declaration zones list when the edited module is not a main Pascal program.

-- NOTNOT - User

internal code 24
default value reset

Affects unparsing of the logical operator 'not'.
When NOTNOT is set or toggle TILDNOT is reset, the logical operator is unparsed as 'not' keyword. Otherwise, it is unparsed as ~ .
The symbol ~ is never allowed by the Pascal parser.

-- OCTALB - Expert

internal code 37
default value reset

When OCLATB is set, octal constants (with lexical form 7777B) may be used during parsing instead of hexadecimal constants. Internal representation of octal constants use the same operator than hexadecimal constants. So, be carefull to unparse such operators with the same value for OCTALB toggle, because it will be unparsed as hexadecimal constants if OCTALB is reset as described for HEXAX toggle.

-- PARDOT - User

internal code 19
default value reset

Affects unparsing of array indexation symbols.
When PARDOT is set and SQUARE is reset, array indexation symbols are unparsed as (. and .) . Otherwise they are unparsed as [and].

-- PARSTAR - User

internal code 13
default value set

When PARSTAR is set, comment delimiters (* and *) are allowed during parsing. See also PERCENT and BRACE.
The three toggles BRACE, PARSTAR and PERCENT control allowed comment delimiters during parsing and during interactive input of comments.
When these three toggles are simultaneously reset, NO comment delimiters are recognized in input mode. In this situation, it becomes IMPOSSIBLE to end a comment entered interactively.

-- PERCENT - User

internal code 11
default value set

When PERCENT is set, comment delimiter % is allowed during parsing.
See also PARSTAR and BRACE.

-- SHARP - User

internal code 15
default value set on MULTICS , reset on VAX and SM90

Affects inequality operator unparsing when INFSUP toggle is reset.
When SHARP is set, inequality operator is unparsed as # .
When SHARP is reset and TILDEQ is set, inequality operator is

unparsed as ~= .
When SHARP and TILDEQ are reset, inequality operator is unparsed as <> .

-- SQUARE - User

internal code 18
default value set

Affects unparsingof array indexation symbols.
When SQUARE is set or PARDOT is reset, array indexation symbols are [and] . Otherwise, indexation symbols are (. and .) .

-- SYNT_ERR - User

internal code 58
default value set

When SYNT_ERR is set, Mentor sends an explicit error message in case of syntax error detected during Pascal parsing.

-- TILDEQ - User

internal code 22
default value reset

Affects inequality operator unparsing when INFSUP and SHARP are reset.
When TILDEQ is set, inequality operator is unparsed as ~= .
When TILDEQ is reset, inequality operator is unparsed as <> .

-- TILDNOT - User

internal code 23
default value reset

Affects unparsingof the logical operator 'not'.
When NOTNOT is set or TILDNOT is reset, the logical operator 'not' is unparsed as 'not' keyword. Otherwise, it is unparsed as ~ .

-- UPARROW - User

internal code 17
default value set on VAX and SM90 , reset on MULTICS

Affects pointer dereferencing symbol unparsing.
When UPARROW is set or AROB is reset, pointer dereferencing symbol is unparsed as ^ . Otherwise, it is unparsed as @ .

-- XEXTERN - User

internal code 28
default value set on SM90 , reset on MULTICS and VAX

Affects parsing and unparsing of external procedures and functions.
When XEXTERN is set, the body of external is unparsed as 'extern' .
The Pascal parser admits in this case the keyword 'extern'.
When XEXTERN is reset, unparsing of the body of external procedures and functions depends on the operator used for its internal representation :

- The operator is 'metasym' :
  When XXTERNAL is set, it is unparsed as 'external' ;
  when XXTERNAL is reset, it is unparsed as 'metasym' .
- The operator is 'pascal' :
  When XPASCAL is set, it is unparsed as 'pascal' .
  When XPASCAL is reset and XTERNAL is set, it is unparsed as 'external'.
  When XPASCAL and XTERNAL are reset, it is unparsed as 'extern'.

-- XPASCAL - User

internal code 27
default value set on MULTICS , reset on VAX and SM90

Affects the unparsing of body of external of procedures and functions only when XEXTERN is reset and when its internal representation uses the 'pascal' atomic operator.
When XPASCAL is set, the body of external procedures and functions is unparsed as 'pascal' .
When XPASCAL is reset and XTERNAL is set, it is unparsed as 'external'.
When XPASCAL and XTERNAL are reset, it is unparsed as 'extern' .

The 'pascal' keyword is allways allowed by the Pascal parser.

-- XTERNAL - User

internal code 29
default value set on VAX and SM90 , reset on MULTICS

Affects unparsing of the body of external procedures and functions, only when XEXTERN and XPASCAL are reset, and when its internal representation use the 'pascal' atomic operator.
When XTERNAL is set, the body of external procedures and functions is unparsed as 'external' .
When XTERNAL is reset, it is unparsed as 'extern'.

The 'external' keyword is allowed by the Pascal unparser when XTERNAL or XXTERNAL are set.

XTERNAL is also used in Mentor-Rapport.

-- XXTERNAL - User

internal code 38
default value set

Affects unparsing of the body of external procedures and functions, only when XEXTERN is reset and when its internal representation use the 'metasym' atomic operator.
When XXTERNAL is set, the body of external procedures and functions is unparsed as 'external' .
When XXTERNAL is reset, it is unparsed as 'metasym' .

The 'external' keyword is allowed by the Pascal parser when XXTERNAL or XTERNAL are set.

Also used in Mentor-Metal.

4.6 MENTOR-METAL TOGGLES

-- DECOMP2 - Expert

internal code 50
default value reset

When DECOMP2 is reset, the Metal standard unparser is called.
When DECOMP2 is set, another special Metal unparser is called to generate input text for Yacc and Lex.
DECOMP2 is automatically modified by the '.compile' procedure of the Metal standard environment.
DECOMP2 cannot be used for outputs on screen.

DECOMP2 is also used in Mentor-Rapport and may be used for any other language to call two different unparsers.

-- HEXAQQ - Expert

internal code 26
default value reset

Used by the Mentor-Metal compiler to produce compact code.
This toggle is automatically set and reset when the '.compile' Metal command is call with the argument '.compact'.
(See informations about the Metal environment).

HEXAQQ is also used in Mentor-Pascal.

-- IDLC - User

internal code 40
default value reset

Used in the Metal unparser.
When IDLC is set and IDMC is reset, non-terminals are unparsed in lowercases. IDLC is set in the Metal standard environment.

IDLC is also used by other language unparsers.

-- IDMC - User

internal code 44
default value reset

Used in the Metal unparser.
When IDMC is set, non_terminals are unparsed in mixed_cases (i.e. in lowercases , except initial letter and all letters following an underscore character).

IDMC is also used by Ada and Rapport unparsers.

-- KWUC - User

internal code 9
default value reset

Used in the Metal unparser.
When KWUC is set, the Metal keywords are unparsed in uppercases.

KWUC is also used by other language unparsers.

-- METAKEY - Expert

internal code 31
default value reset

Used in the Metal unparser.
When METAKEY is set, the Metal unparser generates input text for the SYNTAX parser generator. On the Multics version of Mentor, METAKEY is automatically set and reset by the '.compile' procedure of the Metal standard environment.

METAKEY is als used in Mentor-Pascal.

-- TERMLC - User

internal code 51
default value reset

Used in the Metal unparser.
When TERMLC is set, the terminal elements in productions are unparsed

in lowercases.

-- XXTERNAL - Expert

internal code 38
default value set

Used in the Metal unparser when DECOMP2 is set.
When XXTERNAL is set, Metal unparser generates input text for Yacc.
When XXTERNAL is reset, Metal unparser generates input text for our local Cxyacc parser generator (which is a Yacc written in Lisp).

XXTERNAL is also used in Mentor-Pascal.

4.7 - MENTOR-RAPPORT TOGGLES

-- IDLC - User

internal code 40
default value reset

Used in the Rapport unparser when IDMC is reset.
When IDLC is set, identifiers are unparsed in lowercases.

IDLC is also used in other language unparsers.

-- IDMC - User

internal code 44
default value reset

Used in the Rapport unparser.
When IDMC is set, identifiers are unparsed in mixed_cases (i.e. in lowercases, except for the initial letter and all letter following an underscore character). IDMC is set in the Rapport standard environment.

IDMC is also used in other language unparsers.

-- NOIDENT - Expert

internal code 55
default value reset

Used in the Rapport unparser.
When NOIDENT is set, sections identifiers are not unparsed.

-- NOMETA - Expert

internal code 54
default value reset

Used in the Rapport unparser.
When NOMETA is set, metavariables are not unparsed.

-- PHOTOCMP - Expert

internal code 42
default value reset

Used in the Rapport unparser.
When PHOTOCMP is set, the RAPPORT unparser generates TROFF input text.

PHOTOCMP can be used in other languages to generates TROFF input text in permanent files, with different fonts for identifiers, keywords and comments.

-- SECTNUMS - Expert

internal code 53
default value reset

Used in the Rapport unparser.
When SECTNUMS is set, sections are numbered during unparsing.

-- SHORTPART - User

internal code 56
default value reset

Used in the Rapport unparser.
When SHORTPAR is set, paragraphs are unparsed in abbreviated form : only the two first lines are displayed, next lines are represented by holophrast symbols . SHORTPAR is set in the Rapport standard environment.

-- TABMAT - User

internal code 52
default value reset

Used in the Rapport unparser.
When TABMAT is set, the table of contents of the document is displayed.

-- WIZARD - Expert

internal code 49
default value reset

Used in the Rapport unparser when PHOTOCMP is set.
When WIZARD is set, spaces and dots at the beginning of lines are not deleted during the generation of TROFF input text. So the user can include in the text his own composition directives.

-- XTERNAL - Expert

internal code 29
default value set on VAX and SM90 , reset on MULTICS

Used in the Rapport unparser when PHOTOCMP is set.
When XTERNAL is set, the TROFF input generated will produce a more compact output

XTERNAL is also used in Mentor-Pascal.

mentor.break.i

November 13 1984:
mentor.interrupt_facility

When you hit the INTR key during a mentor session the current computation is interrupted and you go up one level in Mentor, as if you had invoked the Mentor system procedure .user. You can come down with $ or $-.

The INTR key is the key defined by the "intr" argument of the stty Unix command. It is the key you use to interrupt any command under Unix.

****Warning for installations on Berkeley Unix 4.1 *****

Interrupting a listing (i.e. the execution of a print (p) command) in tty mode often makes the system crash because of a bad recovery of interruptions by the Pascal write procedure.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique