The (ihs) Reference Manual

Larsson, Jan Eric; Persson, Per

1987

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7341)/1-121/(1987)

The (ihs) Reference Manual

Jan Eric Larsson & Per Persson

Department of Automatic Control
Lund Institute of Technology
March 1987

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Technical report

Date of issue
March 1987

Document Number
CODEN: LUTFD2/(TFRT–7341)/1–121/(1987)

Author(s)
Jan Eric Larsson
Per Persson

Supervisor

Sponsoring organisation
The National Swedish Board for Technical Development, (STU), contract no. 85–3042

Title and subtitle
The `(ihs)` Reference Manual

Abstract

The `(ihs)` Reference Manual gives a technical description of the help system `(ihs)`. It is described how new scripts and rules are introduced, and how the system is started. A listing of the complete source code for the system is also given.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
121

Recipient's notes

Security classification

The `(ihs)` Reference Manual

Yet another notch in the belt for AI.

The (ihs) Reference Manual

Jan Eric Larsson
Per Persson

Department of Automatic Control
Lund Institute of Technology
March 1987

Department of Automatic Control
Lund Institute of Technology
Box 118
S-221 00 LUND
Sweden

Published 1987
Printed in Sweden

Contents

1

Introduction

This report gives a technical description of a help system based on expert system techniques. The help system works as an interface to Idpac, an interactive program for system identification, i.e., as an intelligent front-end.

The intention of this manual is to give a complete description of the expert interface, exactly as it was implemented. This means that there is no talk of the ideas behind it, of possible extensions, nor of the knowledge database. Basically, this report contains two parts. The first is a reference manual, giving a technical overview of the system, as well as providing information on how to build new knowledge databases and adding commands and macros to the command grammar. The second part is the complete source code of the system, including some hopefully useful comments.

We are better programmers now, than we were when the project started. There are some bugs and some strikingly inefficient code, but we have chosen not to hack forever on the program, and to accept that its main purpose is for demonstrations and for showing ideas.

The ideas underlying the expert interface, a general discussion about the implementation, the knowledge database, and the possibility of further developments, are all treated in our thesis, Larsson and Persson [1987 a]. The knowledge database that has been developed during the project is found in Larsson and Persson [1987 c].

For readings on Idpac, see Wieslander and Elmqvist [1978], Wieslander [1979 a, b, c, 1980].

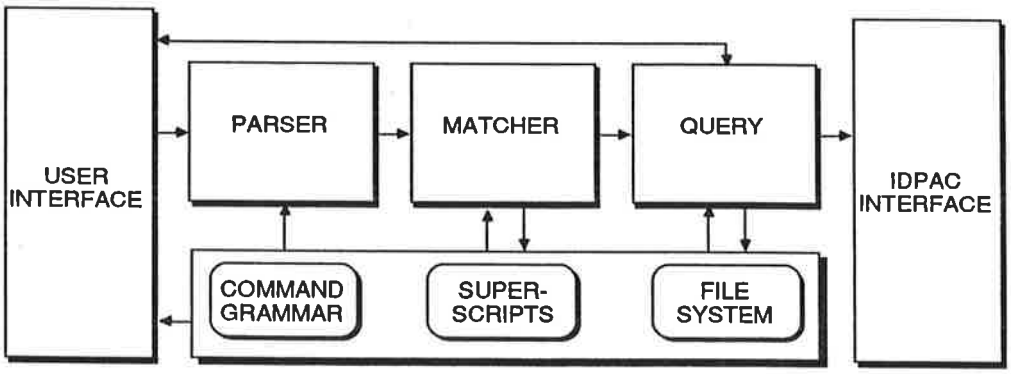
2

Overview of the System

The implementation of the expert interface has formed a crucial part of our project. It contains all essential parts, as a command parser, script matcher, and production rule system. In addition to this there are several utilities that must be present if the program is to work as a realistic example. These are a query module, a file system, and interfaces to the user and Idpac. There are some limitations, though. The interface currently handles only a subset of all the Idpac commands, and there are some irregularities in the syntax of some Idpac commands that are not fully supported.

General Layout

The expert interface is made up from several parts. Most of the parts work on a common database.



Layout of the system.

The user interface reads a command from the user and transforms it into a Lisp list. It provides all the input and output functions used by the other parts of the interface. In this way, all of the system's dependence on terminal types, graphics, etc., is collected in one place.

The command parser checks the commands for syntactical correctness and supplies defaults in the same way that the parser of Idpac does. In this process it transforms the commands into a more convenient form. The parser accepts commands with arguments left out, as the other routines will fill information in, by defaulting from scripts or asking the user. This is a useful feature in itself, since short commands are desirable in the dialog, but complete commands are more useful in documentation.

The script matcher incrementally keeps track of the script data structures and updates them according to the incoming commands. The commands are transformed, and files may be defaulted with the help of knowledge from the scripts. The details of this can be found in the thesis.

Each script object inherits a YAPS database. When a command has been successfully matched against a script and the script is updated, facts may be put in its database. This takes care of all information that is not directly available in the scripts, e.g., the results of different commands, etc.

The system allows any number of different scripts to be followed in parallel. The superscripts are used to accomplish this. Each superscript contains the current state of a session with one or several scripts. One of the superscripts is currently active and the others, if any, are waiting in a suspended state. When a command does not match any script in the current superscript, the system tries to find a new current superscript by testing the suspended superscripts and also a superscript in the initial state.

The query module goes through the command description and tries to fill in the remaining unknown entries by asking the user about them. In this way the user may give only the command name, and then he will be prompted for all the arguments left out. The query module also sends messages to the file system about created and deleted files.

The file system keeps track of all the files created and used during an Idpac session. It does this by storing data about the files in a directed graph structure. This enables the file system to show the ancestors or descendants of a file, i.e., the files used in the creation of and the files created with the use of a specific file.

The database contains the command grammar used by the parser, the scripts and rules used by the script matcher, the file tree of the file system, and state variables for keeping track of the user state, internal tracing, and so on.

The Idpac interface handles the communication with Idpac. It transforms the commands delivered by the query module into text strings which are read by Idpac. The expert interface and Idpac reside in two different VMS processes. The Idpac interface sends the processed commands to Idpac via a VMS mailbox. In this way no changes had to be done to the Idpac program itself. The routines for interprocess communication are written in C.

The system is written in Franz Lisp, Foderaro and Sklower [1981], extended with Flavors, Allen *et al* [1984], and YAPS, Allen [1983]. It consists of about 6000 lines of code and runs under VMS, Digital [1984], and Eunice, Kashtan [1982], on a VAX 11/780.

3

The Knowledge Database

The knowledge database of the system is partitioned into scripts, rules, and command descriptions. In addition to these, there are several other small knowledge databases, e.g., the explanations of the on-line dictionary, the file system, the Idpac state-tracking system, etc. The latter are all of a trivial nature, and we will not give any closer description of them here.

The Script Grammar

The script data structure was defined as a language for describing command sequences. A script consists of a series of clauses. Each clause either describes a command, sends facts to a production rule system, or affects the structure of the script. A formal description of the script language in Extended Bacchus-Naur Form is given below, followed by an explanation of the clauses.

```
SCRIPT              ::=  (SCRIPTCLAUSE [SCRIPTCLAUSE...])
SCRIPTCLAUSE        ::=  COMMAND | ASSIGN | KSCALL | SCRIPTMACRO |
                           REPETITION | REPETITION* | OR | ALL | EMPTY
COMMAND             ::=  (command COMMANDNAME [[OUTFILEDESCRIPTOR...] | [INFILEDESCRIPTOR...] |
                           [GLOBFILEDESCRIPTOR...] | [PARAMETER...]...])
COMMANDNAME         ::=  <identifier>
OUTFILEDESCRIPTOR   ::=  (outfile <identifier>)
INFILEDESCRIPTOR    ::=  (infile <identifier>)
GLOBFILEDESCRIPTOR  ::=  (globfile <identifier>)
PARAMETER           ::=  (TYPE <identifier>)
TYPE                ::=  number | numlist | numlist1 | symbol | symlist | symlist1
ASSIGN              ::=  (assign NEWNAME OLDNAME)
NEWNAME             ::=  <identifier>
OLDNAME             ::=  <identifier>
KSCALL              ::=  (kscall FACTLIST)
FACTLIST            ::=  ([<identifier> | <Lisp list> | PARAMETERVALUE ...])
PARAMETERVALUE      ::=  (parameter <identifier>)
SCRIPTMACRO         ::=  (scriptmacro MACRONAME INCLAUSE OUTCLAUSE)
MACRONAME           ::=  <identifier>
INCLAUSE            ::=  (in [<identifier>...])
OUTCLAUSE           ::=  (out [<identifier>...])
REPETITION          ::=  (repeat SCRIPT)
REPETITION*         ::=  (repeat* SCRIPT)
OR                  ::=  (or SCRIPT [SCRIPT...])
ALL                 ::=  (all SCRIPT [SCRIPT...])
EMPTY               ::=
```

A script-macro is defined by the following grammar.

```
SCRIPTMACRODEF      ::= (SCRIPTMACRONAME [MACROINCLAUSE] [MACROOUTCLAUSE] [SCRIPTCLAUSE...])
SCRIPTMACRONAME     ::= <identifier>
MACROINCLAUSE       ::= (in [<identifier>...])
MACROOUTCLAUSE      ::= (out [<identifier>...])
```

An internal name is the name of a variable in a script. Examples of internal names are the identifiers in the **infile** and **outfile** clauses. An external name is a name of a file or a value typed by the user. Each internal name is associated with an external one.

The **command** clause describes a command that the script should match. The name of the command and some of its parameters appear in sub-clauses of the command clause. The names in any file or parameter clause are associated with the corresponding external filenames and parameter values. If a file is declared **infile**, the internal name must have been used earlier in the script. The actual external filename must match the old internal value in order for the command to match. If the external filename is left out, it can be defaulted from the script. A file declared **outfile** is a file that is created by the command. Its name must be given by the user or created automatically. A **globfile** corresponds to a file that contains indata to the session, i.e., a filename that must be given. A sub-clause with any other kind of parameter has the effect of catching the value of the corresponding actual parameter. In this way parameter values can be transferred from the command via script matching to the fact databases of the production systems.

The **assign** clause is used to associate a new name with an old one. This is useful in the case of an alternative. An example might look like this.

```
(command plot (globfile DATA))
(or
 ((command trend (outfile NEW-FILE) (infile DATA)))
 ((assign NEW-FILE DATA)))
(command plot (infile NEW-FILE))
```

First, a new file is plotted. Then either trends are removed from it, which creates a new file, or they are not. The second alternative means that the name **NEW-FILE** must be associated with **DATA**, otherwise the next **plot** command would not work. The **assign** clause has this effect.

The **kscall** clause (**kscall** stands for Knowledge Source Call) contains facts which should be added to the database of the production system in a script, when the previous command has been matched. This is used to fire rules in the production systems that are associated with each of the scripts. The facts consist of Lisp lists and are added to the YAPS database. A clause in a fact list that has the form **(parameter <identifier>)** is substituted for the value associated with the identifier. This is used to transmit a parameter value from a command to the rulebases.

Some pieces of scripts may be so useful and occur so frequently that one would like to make subroutines of them. The **scriptmacro** clause calls such a macro. The **in** and **out** sub-clauses are lists of the in and out parameters, respectively. When a **scriptmacro** clause is reached in a script, a list of all defined script-macros is checked to see if there is one with the correct name and correct number of in and out parameters. If so, the script-macro is textually inserted in the script, i.e., a true macro-type call is used. The names of any files created in script-macros bubble out to the surrounding level. There are no local variables in the macros. A good programming practice when writing script-macros would be to mention the names of all created files in the **out** parameter list and the names of all files used by the macro, but not created by it, in the **in** parameter list.

Four different constructs are used to control the sequencing in the scripts. The keyword **repeat** means that the script mentioned in the clause may be repeated one or more times. **repeat*** means repetition zero or more times. The **or** clause lists several scripts of which one must be chosen, and the **all** clause demands that all the listed scripts must be matched, but not in any particular order.

The Rule Grammar

The system uses the expert system shell YAPS for its rulebases, and the rules must thus obey the grammar of YAPS rules. This is described in the YAPS manual, Allen [1983], and will not be treated here.

There are several ways of entering rules in YAPS, such as **defp**, **p**, etc. The expert interface handles this at startup. Therefore the rules should be entered in the following way.

For every script there is list of associated rules. Each rule is contained in a Lisp list of the following form.

```
((fact 1)
 (fact 2)
 ...
 -->
 (action 1)
 (action 2)
 ...
)
```

As can be seen, there is neither a **defp** nor a name. The system will create names and put the rules in the appropriate YAPS databases when it is initialized. There is an additional rule group, for rules that should be available in all scripts. These rules are placed in the **global-rules** list.

Building a New Knowledge Database

The construction of a new script requires two different things. The command sequence must be entered, and rules associated with it must be written.

Before the actual implementation of a new knowledge database, several phases, such as problem definition and knowledge acquisition, must have taken place. These matters will not, however, be dealt with here.

The first thing that the knowledge engineer should do is to decide upon a few command sequences that will achieve the goal in question. There is no need to find all possible command sequences, a selection will often do. These command sequence is entered into the system in the script language. All scripts should be appended to the **scripts** list, found in the file **scripts.l**. The list of scripts is an association list, consisting of pairs of the name of the script and the actual command sequence "code." Also, there is a short and a long help text for each script. These texts will be put into the on-line dictionary. The script can now be subjected to test runs.

Next, knowledge source calls, **kscalls**, should be put in at the right places. After most commands there will be a need for a small help text, giving advice on what the next command could be, and the knowledge source calls will usually be numerous.

This is an example of the layout of a script.

```
(setq scripts '(

  (example
    (script example
      ("EXAMPLE"
       "The EXAMPLE script does nothing important.")
      (
       (command conv (outfile WRK) (globfile ASCII))
       (kscall (file (parameter WRK) converted))
       (command plot (infile WRK))
       (command stop))))

  (ml
    (script ml
      ("ML"
```

```
    "The ML (Maximum Likelihood) estimation script computes and"
    "verifies a model of a transfer function that could have"
    "produced the output (given) from the input (also given).")

   (
    (kscall (system-list ()))
    (kscall (bode-plot-list-ba ()))
    ...
    )))))
```

After this, rules to take care of the facts in the knowledge source calls should be entered. All rules are found in the **rules** list, in the file **rules.l**. This list is also an association list, consisting of pairs of the name of the script and a list of the rules belonging to it. After each command sent to Idpac, facts will be fed to the rulebases and YAPS run. The rules may then use the facts freely. If a rule puts a fact looking like **(message <list of strings>)** into the YAPS database, the strings will be concatenated and appear on the screen when the user types '?'.

An example of the **rules** list is the following.

```
(setq rules '(

  (example

    ((file -x converted)
     -->
     (fact message ("The file" -x "has been converted.")))

    ...)

  (ml

    ((fact length -f1 -l1)
     (fact length -f2 -l2)
     test
     (and
       (not (equal -f1 -f2))
       (not (equal -l1 -l2)))
     -->
     (fact message
       ("The files" -f1 "and" -f2 "are not equally long. You must"
        "CUT one or both before using an identification algorithm.")))

    ...)))
```

There is a rudimentary check of the syntax of a command sequence, but no check of the rules. Also, no semantic check is done, so the knowledge engineer is entirely responsible for keeping track of all the facts used. Our experience is that this may become quite complicated, and a lot of test runs will generally be needed during the development of a larger script.

4

The Command Grammar

The parser of the expert interface checks the commands for syntactical correctness and turns the commands into a more convenient form. In order to do this, it must have descriptions of the different commands. These descriptions, or command definitions, are found the **external-commands** list, placed in the file **commands.l**. Currently, not all Idpac commands will be accepted by the expert system. To enable the system to parse a previously undefined command, or a newly written macro, a definition of the command or macro should be put into the **external-commands** list.

The **external-commands** list is an association list consisting of pairs of the following form.

```
(name (command <description>) (strings <help strings>))
```

The description of the command is made up of clauses describing the corresponding parameter. Each clause has two or more help text strings to match it. The shorter help text is used by the Query module when asking for missing parameters, and should the user want help with understanding such a question, the longer help text (the butfirst texts concatenated) is also output.

Adding a New Command or Macro

When a new command or macro should be added, a definition of it must be put into the commands list. Each parameter of the command or macro should have a corresponding description clause and one short and one long help string. When all this has been put in, the system will be able to handle the command or macro.

The descriptions can be one of the following: **infile, outfile, number, numlist, numlist1,** (which means a list of exactly one number), **symbol, symlist,** or **symlist1**. The descriptions are usually made up of a list containing the appropriate word, e.g., **(number)**. This will match a corresponding number in the input command. If a parameter should have a default, this is written as, e.g., **(number (default 1.0))**, or, if the default is blank, as **(number (default))**. Idpac works with its data in files, and most commands create new files for their results. An already existing file should be described with an **infile** clause, and a file created by a command with an **outfile** clause. An infile or outfile descriptor may have an extension telling the type of the data in the file, **.d** or **.t**. The default extension is **.d**.

This is an example of the structure of the list **external-commands** and of some command definitions. The **internal-commands** list has the same structure.

```
(setq external-commands '(

  (acof
   (command acof
            (outfile)
            (numlist1 (default (1)))
            <
```

```
            (infile)
            (numlist1 (default (1)))
            (number)
            (symbol (default)))

    (strings ("the ACOF command"
              "The ACOF command computes the autocorrelation function of"
              "a time series.")
             ("auto covariance outfile"
              "The outfile will contain the auto covariance function. Use"
              "the BODE command to plot it.")
             ("outfile column number"
              "The column of the outfile where the auto covariance function"
              "will be put. Defaults to (1).")
             ("<"
              "Less than, 'Fedtmule'!")
             ("time series infile"
              "The infile should contain data for a time series. From this"
              "the auto covariance function will be computed.")
             ("infile column number"
              "The column of the infile where the time series is to be found."
              "Defaults to (1).")
             ("number of lags"
              "The number of lags for computing the auto covariances."
              "If you don't know what it should be, try 10 or 100.")
             ("extension tag"
              "If set, this will be the name of a variable, set to the"
              "variance of the data in the outfile.")))

   (alter
    (command alter
             (number))

    (strings ("the ALTER command"
              "The ALTER command is used as a subcommand to PLMAG."
              "It alters the current value of a data point.")
             ("data point number"
              "The number of the data point to alter."
              "Idpac will prompt you for the new value")))

    ...
))
```

If one wants to add a command that should go into the interface itself and not to Idpac, i.e., an internal command, the description of this command should be put in the **internal-commands** list, which is also found in the file **commands.l**. The method that implements the action of the internal command should be named **:internal-command-** and the name of the command. There is a special section in the database where these methods are located. The **Lisp** command, e.g., has the following method associated to it.

```
(defmethod (database-flavor :internal-command-lisp) (command)
  (setq user-top-level nil))
```

When the parser sees an internal command, the corresponding method is called. The method is always given the complete command, in the form that comes from the parser, as argument.

There are yet two other lists, where the name of the command might appear. If the command is of a general nature and may be used outside of scripts, as, e.g., the **plot** command, it should be put in the **allowed-commands** list. Commands that need to have special procedures called when they are executed, should be put in the **special-commands** list. The corresponding procedure should be put into the database module. Currently, the special commands are **delet, free,** and **let**, all of which affect data internal to the expert interface. The name of the method for the special command should be **:special-command-** and the name of the command. The let command, e.g., has the following method associated to it.

```
(defmethod (database-flavor :special-command-let) (command)
  (<- self ':add-intrac-variable (cadadr command) (cadadddr command)))
```

Unsupported Idpac Features

Some of the grammar of Idpac's commands is currently not accepted by the parser of the expert interface. Among other things, this includes the use of '/' and ':' in the plot command. In the present implementation, these characters are matched as **symbols**, which means that they will be confused with file names. Our remedy has been to give advice on how to type the command through the rule-based help. If the user should plot the data files **a** and **b**, the rule system could say **Plot the files a and b by giving the command PLOT A / B.**

Another thing that is currently not handled very well by the system is when several different files are created by the same command, in a repetition. Only the latest created file may be explicitly used later in the script. The following situation is an example.

```
(repeat
 ((command mlid (outfile SYST) (infile IN) (infile OUT))))
 ...
 (command residu (outfile RES) (globfile SYST) (infile IN) (infile OUT))
 ...
```

The idea is that any of the created system files should be allowed in the **residu** command. A clause **(infile SYST)** would only match the latest created system file. The good solution is to introduce a special **one-of** file clause, but this has not been done. The trick solution that has been used by us, is to use a **globfile** clause. This makes it possible for the user to type in any of the previously created files. There is no check, however, that he does not type in a completely new file, which could considerably mess things up. A suggestion could, of course, be made by the rule system. This is the approach that we have taken when developing knowledge databases. A help text could, e.g., say **Compute the residuals from one of the pair of files IN1 - OUT1, IN2 - OUT2, ...**

Idpac sometimes uses subcommands. These are treated as ordinary commands by the expert system. The only drawback with this is that the system does not keep track of whether it is in a subcommand mode. Instead, subcommands will be accepted at any time. Also, the subcommands needed in a script must be explicitly given, just as any other commands.

5

The On-line Dictionary

The on-line dictionary is used to explain the different commands, scripts, and concepts used in system identification. The user types the command **explain** and the beginning of the word he wants to have explained. The system outputs explanations for all words starting with the given letter combination.

Every command and script has a few help texts associated to it, and these texts are automatically fed to the dictionary. In addition to this, concepts mentioned in the rule-based help should be given explanations. This is done by entering a clause in the list `other-dict-words` in the file `dictwords.l`. This is how this list looks.

```
(setq other-dict-words '(
  (internalcommands
    ,(cons
       "internal commands is obtained by asking about"
       (mapcar 'car internal-commands)))
  (fruit
    ("fruit" "The idpac fruit can be set to POTATIS."))
  (lags
    ("lags" "This term signifies a number of data points to be"
            "involved in a computation. It may be the size of"
            "the time window for a FFT or convolution computation."
            "Don't be afraid to experiment with a few different"
            "values if you don't know what number to give."))
  (larsson
     ("Larsson" "Larsson and Persson wrote the program."))
  (persson
     ("Persson" "Larsson and Persson wrote the program."))
  (outliers
    ("outliers" "An outlier is a data point, which is far out."
                "The reason for this may be some kind of measurement"
                "error. If you want to avoid outliers, try to use"
                "other parts of data. If that is not possible, you"
                "may replace an erratic data point with interpolation."
                "This is done with the PLMAG command."))
  (trends
    ("trends" "A trend is a finite order polynomial forming part of"
              "a data signal. The most usual kind of trend is one of"
              "zero order, i.e., a bias. Trends should always be"
              "removed before an estimation is started, or the results"
              "will be in error. A bias, e.g., will show up as an"
              "integrator. It is very difficult to remove higher order"
              "trends, so it is recommended that you try to use other"
              "data instead."))
  (help
    ("Ask for internal commands."))))
```

6

Starting the System

In order to run the intelligent help system, you must have the Lisp source code, the compiled C code, and a version of Franz Lisp with YAPS and Flavors. The startup relies on some intricate jumps between processes, and we recommend that the method described below is used.

First, some keys are defined to handle the skipping between the different processes. For this, the two command files `key1.com` and `key2.com` are used. The expert system is started automatically by the include file `lisprc.l`, and Idpac is started with the command file `idpac.com`. Here is an example of how the system is started. The process that is originally running is called "PERP."

```
$ spawn                                          ! Create the Idpac process
Process PERP_1 spawned
Terminal now attached to process PERP_1
$ attach perp                                    ! Change process
Control returned to process PERP
$ yaps                                           ! Start the Lisp
Franz Lisp, Opus 38.79 with YAPS and FLAVORS
-> (load "inc.l")
...
File number 25, ihs.l loaded.
-> (ihs)                                         ! Initialize the system
...
Script ml is OK.
Ready
IHS> lisp                                        ! Exit the (ihs) command loop
-> <CTRL-Y>                                      ! Break the Lisp
*INTERUPT*
$ attach perp_1                                  ! Change process
Control returned to process PERP_1
$ @idpac                                         ! Start Idpac
Process PERP_2 spawned
$ attach perp                                    ! Get back to Lisp process
Control returned to process PERP
$ continue                                       ! Continue Lisp execution
(igs)                                            ! Restart the command loop
Ready
IHS>                                             ! Prompt from (ihs)

IDPAC    V7A                                     ! Idpac comes to life


Copyright (c) Department of Automatic Control
Lund Institute of Technology, Lund, SWEDEN 1986
All Rights Reserved

>
>
>IHS>                                            ! (ihs) and Idpac synchronized
>
>
```

Now, you are hopefully running the system. To exit, type `stop` to stop Idpac, and exit the Lisp.

7

References

ALLEN, E. M. (1983): "YAPS: Yet Another Production System," Technical report, TR–1146, Department of Computer Science, University of Maryland, Baltimore County, Maryland.

ALLEN, E. M., R. H. TRIGG and R. J. WOOD (1984): "The Maryland Artificial Intelligence Group Franz Lisp Environment," Technical report, TR–1226, Department of Computer Science, University of Maryland, Baltimore County, Maryland.

DIGITAL EQUIPMENT CORPORATION (1984): *Introduction to VAX/VMS System Routines*, VAX/VMS Version 4.0, Digital Equipment Corporation, Maynard, Massachusetts.

FODERARO, J. K. and K. L. SKLOWER (1981): *The Franz Lisp Manual*, University of California, Berkely, California.

KASHTAN, D. L. (1982): "EUNICE: A system for porting UNIX programs to VAX/VMS," Artificial Intelligence Center, SRI International, Menlo Park, California.

LARSSON, J. E. and P. PERSSON (1987 a): *An Expert Interface for Idpac*, Licentiate thesis, TFRT–3184, Department of Automatic Control, Lund Institute of Technology, Lund.

LARSSON, J. E. and P. PERSSON (1987 b): "The `(ihs)` Reference Manual," Technical report, TFRT–7341, Department of Automatic Control, Lund Institute of Technology, Lund.

LARSSON, J. E. and P. PERSSON (1987 c): "A Knowledge Database for System Identification," Technical report, TFRT–7342, Department of Automatic Control, Lund Institute of Technology, Lund.

WIESLANDER, J. and H. ELMQVIST (1978): "INTRAC, a communication module for interactive programs. Language manual.," Technical report, TFRT–3149, Department of Automatic Control, Lund Institute of Technology, Lund.

WIESLANDER, J. (1979 a): *Interaction in Computer-Aided Analysis and Design of Control Systems*, Doctorial Dissertation, TFRT–1019, Department of Automatic Control, Lund Institute of Technology, Lund.

WIESLANDER, J. (1979 b): "Design Principles for Computer-Aided Design Software," *Preprints of the IFAC Symposium on CAD of Control Systems*, Zürich.

WIESLANDER, J. (1979 c): "Idpac User's Guide," Technical report, TFRT–7605, Department of Automatic Control, Lund Institute of Technology, Lund.

WIESLANDER, J. (1980): "Idpac Commands—User's Guide," Technical report, TFRT–3157, Department of Automatic Control, Lund Institute of Technology, Lund.

8

The Source Code

This manual is not intended to be useful for any than the simplest needs. The user who wants to learn more deeply about the system, must go to the thesis, Larsson and Persson [1987], and, ultimately, to the source code. For this reason, the complete source code is included here. The code has quite a few comments in it, and is supposed to be self documenting to a large degree. Still, we are acutely aware of the difficulties that must meet anyone, who tries to understand how the system really works. We would like to wish them good luck, at the very least.

The Lisp Code

Listing of the file abbrev.l

```
; Abbreviate flavor is a mixin which allows the use of shortforms of commands.
; The method :get-abbreviated-word is used externally.
;
; (<- some_object ':get-abbreviated-word 'ab '(abc aef ghi))
;   => (success abc)
;
; (<- some_object ':get-abbreviated-word 'a  '(abc aef ghi))
;   => (ambiguous abc aef)
;
; (<- some_object ':get-abbreviated-word 'b  '(abc aef ghi))
;   => (fail)
;
; (<- some_object ':get-abbreviated-word 'aef '(abc aef ghi))
;   => (success aef)
;

(defflavor abbreviate-flavor
  ()
  ())

; ------------------------------------------------------------------------

(defmethod (abbreviate-flavor :get-abbreviated-word) (word word-list)
  (let ((x (<- self ':match-list word word-list)))
    (cond
      ((null x)
        (list 'fail))
      ((equal (length x) 1)
        (list 'success (cadar x)))
      ((assq 'exact-match x)
        (list 'success (cadr (assq 'exact-match x))))
      (t
        (cons 'ambiguous (mapcar 'cadr x))))))

; -- Internal methods ----------------------------------------------------

(defmethod (abbreviate-flavor :match-word) (word exact-word position)
  (let
    ((x-w  (getchar word position))
     (x-ew (getchar exact-word position)))
    (cond
```

```
      ((and (null x-w) (null x-ew))
        (list 'exact-match exact-word))
      ((null x-w)
        (list 'partial-match exact-word))
      ((equal x-w x-ew)
        (<- self ':match-word word exact-word (1+ position)))
      (t
        nil))))

(defmethod (abbreviate-flavor :match-list) (word word-list)
  (let
    ((x (cond (word-list (<- self ':match-word word (car word-list) 1)))))
    (cond
      ((null word-list) ())
      (x (cons x (<- self ':match-list word (cdr word-list))))
      (t (<- self ':match-list word (cdr word-list))))))
```

Listing of the file commands.l

```
;
; This file contains the grammar definition of the Idpac
; commands with help textstrings --- Jan Eric --- 30 juni '86
;
; Lots of revisions later on
;

; Some commands are allowed anywhere in any script. The name
; of such a command should be put in the list allowed-commands.

(setq allowed-commands
  '(plbeg alter kill let plot write x save page look hcopy switch))

; This is a list of commands that can have subcommands. The
; underlined subcommands are currently known by the system.
;
; plmag: block plbeg alter page delet kill x
;               ----------------       ------
; plot:  kill page skip
;        ---------
; insi:  prbs norm rect sine zero step ramp pulse srtw look kill x
;                                 ----                  -----------
; resid: kill page table
;        ---------
; ls:    save kill x
;        -----------
; ml:    inval fix save look kill x
;                 ----------------
; struc: revert na nu nb kb fix unfix kill x
;                                     ------
; fhead: look kill x
;        -----------

; Some commands have procedures associated with them in the database.
; These commands must appear in the list of special commands. See
; also the 'special commands' part of the database.

(setq special-commands '(delet let free))

; Commands which are internal to (ihs) and do not exist in Idpac.
; These are mainly help commands, etc.

(setq internal-commands '(

  (?
   (command ?)

   (strings ("the ? command"
             "The ? command is used to get information from the rules"
             "associated with each matching script.")))

  (??
   (command ??)

   (strings ("the ?? command"
             "The ?? command is used to get information about the next"
             "command in each matching script.")))
```

```
(children
 (command children
          (symlist1 (default (d)))
          (infile))

 (strings ("the CHILDREN command"
           "The CHILDREN command types out the names of all files that have"
           "been created using the argument file.")
          ("default extension"
           "The default extension determines which file to write."
           "The default is (d).")
          ("parent infile"
           "The infile is the file whose children the command will"
           "type out.")))

(dir
 (command dir)

 (strings ("the DIR command"
           "The DIR command types the names of all the files in the"
           "current directory.")))

(dumpfiles
 (command dumpfiles
          (infile))

 (strings ("the DUMPFILES command"
           "The DUMPFILES command dumps the file system on a mass storage"
           "disk file.")
          ("dump file"
           "The file where the file system will be dumped.")))

(explain
 (command explain
          (symbol))

 (strings ("the EXPLAIN command"
           "The EXPLAIN command uses an on-line dictionary."
           "It can give help on scripts, commands and other things.")
          ("item"
           "The item you want explained.")))

(lisp
 (command lisp)

 (strings ("the LISP command"
           "The LISP command is used to get back to the standard Lisp"
           "toplevel.")))

(menu
 (command menu)

 (strings ("the MENU command"
           "The MENU command is used to set internal states.")))

(parents
 (command parents
          (symlist1 (default (d)))
          (infile))

 (strings ("the PARENTS command"
           "The PARENTS command types out the names of all files that was"
           "used, directly or indirectly, when creating the argument file.")
          ("default extension"
           "The default extension determines which file to write."
           "The default is (d).")
          ("child infile"
           "The infile is the file whose parents the command will"
           "type out.")))

(restorefiles
 (command restorefiles
          (infile))

 (strings ("the RESTOREFILES command"
           "The RESTOREFILES command restores the file system"
           "from a disk file.")
```

```
            ("dump file"
             "The file from which the file system will be restored.")))
  (show
   (command show
            (symbol))

   (strings ("the SHOW command"
             "The SHOW command shows values of the system.")
            ("symbol"
             "Name for the thing you want to know more about.")))

  (think
   (command think)

   (strings ("the THINK command"
             "The THINK command sets the user state to beginner if it"
             "was previously expert. This may start a session of questions"
             "and answers.")))

))

; The definitions of some of the Idpac commands. These definitions are not
; always exactly the same as the commands in Idpac/Intrac. In some cases
; their definitions are somewhat restricted. This is the place where new
; commands and macros should be placed.

(setq external-commands '(

  (acof
   (command acof
            (outfile)
            (numlist1 (default (1)))
            <
            (infile)
            (numlist1 (default (1)))
            (number)
            (symbol (default)))

   (strings ("the ACOF command"
             "The ACOF command computes the autocorrelation function of"
             "a time series.")
            ("auto covariance outfile"
             "The outfile will contain the auto covariance function. Use"
             "the BODE command to plot it.")
            ("outfile column number"
             "The column of the outfile where the auto covariance function"
             "will be put. Defaults to (1).")
            ("<"
             "Less than, 'Fedtmule'!")
            ("time series infile"
             "The infile should contain data for a time series. From this"
             "the auto covariance function will be computed.")
            ("infile column number"
             "The column of the infile where the time series is to be found."
             "Defaults to (1).")
            ("number of lags"
             "The number of lags for computing the auto covariances."
             "If you don't know what it should be, try 10 and 100.")
            ("extension tag"
             "If set, this will be the name of a variable, set to the"
             "variance of the data in the outfile.")))

 (alter
  (command alter
           (number))

  (strings ("the ALTER command"
            "The ALTER command is used as a subcommand to PLMAG."
            "It alters the current value of a data point.")
           ("data point number"
            "The number of the data point to alter."
            "Idpac will prompt you for the new value")))

 (aspec
  (command aspec
           (outfile)
           (numlist1 (default (1)))
```

```
          <
          (infile)
          (numlist1 (default (1)))
          (number)
          (symbol (default)))

 (strings ("the ASPEC command"
           "The ASPEC command is used to compute the autospectrum,"
           "(powerspectrum), of a time series.")
          ("frequency response outfile"
           "The outfile will contain the autospectrum of the data"
           "in the infile. Use the BODE command to plot it.")
          ("outfile column number"
           "The column number for the autospectrum. Defaults to (1).")
          ("<"
           "No comment.")
          ("time series infile"
           "The infile should contain the time series data for which"
           "the autospectrum should be computed.")
          ("infile column number"
           "The column number of the time series infile. Defaults to"
           "(1).")
          ("number of lags"
           "The number of lags for the computation. If you don't know"
           "what it should be, use 10 and 100.")
          ("frequency data file"
           "The frequency data file (if given) should contain points"
           "in the frequency domain to be chosen for the computation."
           "If it is not given, a logarithmic distribution is used."
           "The frequency points must lie in the first column"
           "of the file.")))

(block
 (command block
          (number))

 (strings ("the BLOCK command"
           "The BLOCK command is used as a subcommand to PLMAG."
           "It defines the block length of the data file. In PLMAG"
           "the data file will be plotted one block at a time.")
          ("the block length"
           "The number of data points plotted at a time.")))

(bode
 (command bode                                    ; Incomplete !
          (symlist1 (default (AP)))
          (infile)
          (numlist1 (default (ALL)))
          (symbol (default))
          (numlist1 (default))
          (symbol (default))
          (numlist1 (default))
          (symbol (default))
          (numlist1 (default)))

 (strings ("the BODE command"
           "The BODE comamnd is used to plot frequency response files"
           "in Bode diagram format.")
          ("page switch"
           "The page switch can be AP to plot amplitude and phase"
           "together (the default), AO for the amplitude only,"
           "P for the phase only and A for first plotting the amplitude"
           "and then entering a subcommand.")
          ("frequency response file"
           "The infile with the frequency response data. Usually"
           "produced by another command, like ACOF, ASPEC or DFT.")
          ("column numbers"
           "The columns to be plotted. Defaults to all.")
          ("another frequency response file"
           "The infile with the frequency response data. Usually"
           "produced by another command, like ACOF, ASPEC or DFT.")
          ("column numbers"
           "The columns to be plotted. Defaults to all.")
          ("another frequency response file"
           "The infile with the frequency response data. Usually"
           "produced by another command, like ACOF, ASPEC or DFT.")
          ("column numbers"
```

```
            "The columns to be plotted. Defaults to all.")
           ("another frequency response file"
            "The infile with the frequency response data. Usually"
            "produced by another command, like ACOF, ASPEC or DFT.")
           ("column numbers"
            "The columns to be plotted. Defaults to all.")))

 (ccoff
  (command ccoff
           (outfile .d)
           <
           (infile .d)
           (infile .d)
           (number))

  (strings ("the CCOFF macro"
            "The CCOFF macro computes and plots the cross correlation"
            "of two signals.")
           ("cross correlation outfile"
            "The outfile will contain the computed correlation values.")
           ("<"
            "Will Danilov be released?")
           ("first insignal file"
            "The insignal files should contain one of the signals you want"
            "to compute the cross correlation for.")
           ("second insignal file"
            "The insignal files should contain one of the signals you want"
            "to compute the cross correlation for.")
           ("number of lags"
            "Use 10 or 100 if you don't know anything better.")))

 (coh
  (command coh
           (outfile .d)
           <
           (infile .d)
           (infile .d)
           (number))

  (strings ("the COH macro"
            "The COH macro is used to compute the coherence between two"
            "signals. It also plots the result in a logarithimic scale.")
           ("coherence outfile"
            "The file where the squared coherence function will be put.")
           ("<"
            "When will Spectre ever be stopped?")
           ("first infile"
            "This file should contain one of the insignals.")
           ("second infile"
            "This file should contain another insignal.")
           ("number of lags"
            "The number of lags for the computation. If you have no better"
            "idea, use 20 to 25% of the number of points in the insignals.")))

 (conv
  (command conv
           (outfile .d)
           <
           (infile .t)
           (numlist (default ALL))
           (number)
           (number (default delta)))

  (strings ("the CONV command"
            "The CONV command is used to convert an ASCII data file"
            "to a file using the internal Idpac data representation.")
           ("floating point data outfile"
            "The outfile will contain data in internal Idpac representation,"
            "i.e. Fortran floating point.")
           ("<"
            "This doesn't mean less or equal, clobber!")
           ("ASCII indata file"
            "The infile should contain data in ASCII format, to be"
            "converted to floating point.")
           ("columns to be converted"
            "A list of the form (C1...), specifying what columns in"
            "the infile that should be converted. If not given, this"
```

```
          "will be defaulted to all columns.")
         ("number of columns"
          "The total number of columns in the infile.")
         ("sample interval"
          "The sample interval in seconds to be associated with the"
          "data in the outfile. If not given, it will be defaulted to"
          "the value of the variable DELTA.")))

(cut
 (command cut
          (outfile .d)
          <
          (infile .d)
          (number)
          (number))

 (strings ("the CUT command"
          "The CUT command cuts out a part of a time series file.")
         ("resultfile"
          "The resultfile in which to place the part cut out.")
         ("<"
          "Agent 006 was killed in Berlin...")
         ("time series infile"
          "The file containing the time series from which you want to"
          "cut out a part.")
         ("number of first record"
          "This is the starting point of the resulting time series.")
         ("number of records"
          "This is the number of records, counted from the starting"
          "point, that you want to cut out.")))

(deter
 (command deter
          (outfile)
          <
          (infile .t)
          (infile)
          (number))

 (strings ("the DETER command"
          "The DETER command is used to simulate a discrete system"
          "with a given insignal.")
         ("outfile"
          "File for the simulated signal.")
         ("<"
          "Mathias Rust landed on the Red Square.")
         ("system"
          "The system to be simulated.")
         ("infile"
          "The file containing the insignal.")
         ("number of points"
          "The number of points wanted in the outfile.")))

(dft
 (command dft
          (symlist1 (default (AMP)))
          (symlist1 (default (BC)))
          (outfile)
          <
          (infile)
          (numlist1 (default (1)))
          (number (default 1))
          (number (default (ALL))))

 (strings ("the DFT command"
          "The DFT command is used to compute the discrete Fourier"
          "transform of a time series. Everybody wants to know this"
          "for the different time series they have in their pockets.")
         ("result switch"
          "Can be either (AMP) for amplitude and phase spectrum,"
          "(the default), or (POW) for the power spectrum.")
         ("time window switch"
          "Can be either of (BH) when the data file is multiplied"
          "with Blackman-Harris' time window, or (BC) for a simple"
          "Box-Car window, (which is the default).")
         ("frequency response outfile"
          "The outfile will contain the frequency response. Use the"
```

```
            "BODE command to plot it.")
           ("<"
            "James Bond 007 is on another dangerous mission...")
           ("Time series infile."
            "The infile should contain data for which you are just"
            "dying to see the discrete Fourier transform.")
           ("infile column number"
            "The column in the infile where the data resides. Defaults"
            "to (1).")
           ("starting point"
            "The first data point to be used. Defaults to the first in"
            "the infile.")
           ("number of samples"
            "The number of samples to transform. Defaults to all.")))

(fix
 (command fix
          (symbol)
          (numlist1)
          (number (default 0)))

 (strings ("the FIX subcommand"
           "The FIX subcommand is used to give a fixed value to a"
           "parameter in the A, B, or C polynomial in an ARMA model.")
          ("A, B, or C polynomial"
           "In what polynomial should a parameter be fixed?")
          ("parameter number"
           "The number of the coefficient in the polynomial.")
          ("parameter value"
           "The value to which the parameter should be set.")))

(hcopy
 (command hcopy)

 (strings ("the HCOPY command"
           "The HCOPY command dumps the current plot on the graphics"
           "screen to a hardcopy printer.")))

(insi
 (command insi
          (outfile)
          (number)
          (number (default delta.)))

 (strings ("the INSI command"
           "The INSI command generates a signal on a file.")
          ("outfile"
           "The file where the result is placed.")
          ("number of points"
           "The length of the generated file.")
          ("sampling time"
           "The sampling time of the signal, default is DELTA.")))

(kill
 (command kill)

 (strings ("the KILL command"
           "The KILL command is used to exit from subcommand levels.")))

(let
 (command let
          (symbol)
          =
          (number))     ; Should be (or (number) (symbol)), which,
                        ; according to Larsson, should be named (numbol)
 (strings ("the LET command"
           "The LET command is used to assign values to Intrac variables.")
          ("symbol"
           "The name of the symbol to which you want to assign a value.")
          ("="
           "I only knew there had to be a shredding party.")
          ("number"
           "The numerical value to assign. Can't be a symbol.")))

(loss
 (command loss
          (infile))
```

```
 (strings ("the LOSS macro"
           "The LOSS macro computes the loss function of a file of"
           "residuals (really just the sum of the squares).")
          ("data file"
           "The sum of the squares of the values of this file"
           "will be computed.")))
(ml
 (command ml
          (symlist (default (VOID)))
          (outfile .t)
          (symlist (default))
          <
          (infile .d)
          (numlist (default (1 2)))
          (number)
          (symbol (default)))

 (strings ("the ML command"
           "The ML command is used to obtain a Nth order model"
           "from a data file containing insignal and outsignal"
           "in two of its columns.")
          ("subcommand switch"
           "This switch can be either (SC), activating the subcommand"
           "facility, or (VOID), avoiding it. Defaults to (VOID).")
          ("system outfile"
           "The outfile will contain the resulting model in text form."
           "It can later be used to simulate the behaviour of the model.")
          ("section in outfile"
           "This should be of the form (NAME), where NAME specifies the"
           "name of the section in the resulting outfile. It defaults"
           "to nothing.")
          ("<"
           "Comment at this place will never be used.")
          ("indata file"
           "The infile should contain the insignal and outsignal to"
           "be identified.")
          ("insignal and outsignal columns"
           "This should be a list of the form (1 2), where the first"
           "number specifies the insignal column and the second the"
           "outsignal column. Defaults to (1 2).")
          ("model order"
           "The order of the transfer function fitted to the indata.")
          ("EXT marker"
           "If the EXT marker is present, the global variables V.EXT"
           "and AIC.EXT will be set to the values of the loss function"
           "and Akaike's test quantity, respectively, iff those variables"
           "exists as reals. (I.e. have already been given a value.")))

(mlid
 (command mlid
          (outfile .t)
          <
          (infile .d)
          (infile .d)
          (number)
          (symbol (default)))

 (strings ("the MLID macro"
           "The MLID macro is used to obtain a Nth order model"
           "from two data files containing insignal and outsignal.")
          ("system outfile"
           "The outfile will contain the resulting model in text form."
           "It can later be used to simulate the behaviour of the model.")
          ("<"
           "These comments form an interesting story, don't they?")
          ("insignal file"
           "The insignal file should contain the insignal to be used in the"
           "identification.")
          ("outsignal file"
           "The outsignal file should contain the outsignal to be used in"
           "the identification.")
          ("model order"
           "The order of the transfer function fitted to the indata.")
          ("EXT marker"
           "If the EXT marker is present, the global variables V.EXT"
           "and AIC.EXT will be set to the values of the loss function"
```

```
            "and Akaike's test quantity, respectively, iff those variables"
            "exists as reals. (I.e. have already been given a value.")))

 (mlidsc
  (command mlidsc
           (outfile .t)
           <
           (infile .d)
           (infile .d)
           (number)
           (symbol (default)))

  (strings ("the MLIDSC macro"
            "The MLIDSC macro is used to obtain a Nth order model"
            "from two data files containing insignal and outsignal."
            "It differs from plain MLID by exiting to a subcommand"
            "level. This makes it useful for FIXing parameters.")

           ("system outfile"
            "The outfile will contain the resulting model in text form."
            "It can later be used to simulate the behaviour of the model.")
           ("<"
            "Pointdexter was to be the fall guy.")
           ("insignal file"
            "The insignal file should contain the insignal to be used in the"
            "identification.")
           ("outsignal file"
            "The outsignal file should contain the outsignal to be used in"
            "the identification.")
           ("model order"
            "The order of the transfer function fitted to the indata.")
           ("EXT marker"
            "If the EXT marker is present, the global variables V.EXT"
            "and AIC.EXT will be set to the values of the loss function"
            "and Akaike's test quantity, respectively, iff those variables"
            "exists as reals. (I.e. have already been given a value).")))

(move
 (command move
          (outfile)
          (numlist (default ()))
          <
          (infile)
          (numlist (default ())))
 (strings
          ("the MOVE comamnd"
           "The MOVE command moves one file, or part of a file in"
           "the directory.")
          ("outfile"
           "The target file of the move.")
          ("outfile column"
           "The column of the target file. Default is (1).")
          ("<"
           "Nick Daniloff was here.")
          ("infile"
           "The origin file to move from.")
          ("infile column"
           "The column number in the infile. Default is (1).")))

(page
 (command page)

 (strings ("the PAGE command"
           "The PAGE command is a subcommand used to get the next"
           "page of information in, e.g., the RESID command.")))


(pick
  (command pick
           (outfile)
           <
           (infile)
           (number))
  (strings ("the PICK command"
            "The PICK command picks out equidistant samplies of a"
            "datafile.")
           ("outfile"
```

```
            "The resulting data file, after pick.")
           ("<"
            "Spies in the night.")
           ("infile"
            "The file to be picked.")
           ("each n:th record"
            "Each n:th record of the infile will be sampled and transferred"
            "to the outfile.")))

(plbeg
 (command plbeg
          (number))

 (strings ("the PLBEG command"
           "The PLBEG command is used as a subcommand to PLMAG."
           "It plots one block of data. The length of one data block"
           "is defined by the subcommand BLOCK.")
          ("data point number"
           "The number of the first data point to plot.")))

(plmag
 (command plmag
          (infile)
          (numlist1 (default (1))))

 (strings ("the PLMAG command"
           "The PLMAG command is used to look at data files containing"
           "floating point data. It can also be used to alter data points"
           "in the file.")
          ("indata file"
           "The indata file should contain data in floating point"
           "format. This is the data you are just dying to see!")
          ("columns to be plotted"
           "This should be a list of the form (C1) where C1 is"
           "the column of the infile that you want to see. It defaults"
           "to (1).")))

(plot
 (command plot                                  ; Simplified
          (symlist1 (default (M)))
          (numlist1 (default (NPLX.)))
          (infile)
          (numlist (default (1)))
          (symbol (default))                    ; to take care of plot a/b
          (symbol (default))                    ; is used instead of a file
          (numlist (default)))                  ; extremely ugly ...

 (strings ("the PLOT command"
           "The PLOT command is used to look at data files containing"
           "floating point data. It can also be used to invent the plot"
           "in computer generated detective stories.")
          ("mark option"
           "This should be either (M) for enumeration of the curves in"
           "a plot of multiple curves, (the default), or (NM), which"
           "specifies no marking of curves.")
          ("data points per page"
           "The number of data points to be plotted per page. This"
           "defaults to (NPLX.), which initially is 200.")
          ("indata file"
           "The indata file should contain data in floating point"
           "format. This is the data you are just dying to see!")
          ("indata file"
           "The indata file should contain data in floating point"
           "format. This is the data you are just dying to see!")
          ("columns to be plotted"
           "This should be a list of the form (C1...), where Ci are"
           "the columns of the infile that you want to see. It defaults"
           "to (1).")))

(prewhite
 (command prewhite
          (outfile)
          (outfile)
          <
          (infile)
          (infile)
          (number (default)))
```

```
  (strings
           ("the PREWHITE macro"
            "The PREWHITE macro is used for prewhitening before,"
            "e.g., cross correlation analysis.")
           ("the input signal after prewhitening"
            "The produced (and filtered) input signal.")
           ("the output signal after prewhitening"
            "The produced (and filtered) output signal.")
           ("<"
            "Fawn Hall was here.")
           ("the input signal to be prewhitened"
            "The input signal which will be filtered. Input to PREWHITE.")
           ("the output signal to be prewhitened"
            "The output signal which will be filtered. Input to PREWHITE.")
           ("order of AR filter"
            "This parameter is optional. Default 10.")))

 (randstep
  (command randstep
           (outfile)
           <
           (infile .t)
           (infile .d)
           (number)
           (number (default nplx.)))

  (strings ("the RANDSTEP macro"
            "The RANDSTEP macro generates and plots a number of Monte"
            "Carlo simulated step responses from an insignal and"
            "a system file with covariance matrix.")
           ("outfile"
            "A file containing the simultaions.")
           ("<"
            "Fawn Hall hid letters under her blouse.")
           ("system file"
            "The original system.")
           ("insignal"
            "The file containing the insignal, e.g., a step.")
           ("number of step simulations"
            "The number of step responses to be Monte Carloed.")
           ("plot width"
            "The value of nplx., i.e., the number of points per plot window."
            "The default is the value of the Idpac variable nplx.")))

(randtf
 (command randtf
          (outfile)
          <
          (infile .t)
          (number))

 (strings ("the RANDTF macro"
           "The RANDTF macro produces a set of transfer functions by"
           "Monte Carloing.")
          ("data outfile"
           "The data file where the plots should be put.")
          ("<"
           "Cicciolina was elected to the Italian parliament.")
          ("system file"
           "The text file containing the system to be checked.")
          ("number of simulations"
           "The number of different new systems that should be"
           "produced by the Monte Carlo methods.")))

(residu
 (command residu
          (outfile)
          <
          (infile .t)
          (infile .d)
          (infile .d))

 (strings ("the RESIDU macro"
           "The RESIDU macro takes a system infile and data files"
           "containing output and input signals and computes the"
           "residuals, i.e. the differences between the simulated and"
           "the real output.")
```

```
          ("residuals outfile"
           "The outfile will contain the computed residuals.")
          ("<"
           "Everybody knows what this means, eh?")
          ("system infile"
           "This file should contain a model in text form. The model"
           "will be used in the simulation and the results compared"
           "to the real output.")
          ("insignal file"
           "")
          ("outsignal infile"
           "This file should contain the insignal and outsignal to"
           "be used in computing the residuals.")))

(resume
 (command resume)

 (strings ("the RESUME subcommand"
           "The RESUME subcommand restarts the execution of a macro."
           "Viva Wieslander!")))

(sclop
 (command sclop
          (outfile)
          (numlist1 (default (1)))
          <
          (infile)
          (numlist1 (default (1)))
          (symbol)
          (number))

 (strings ("the SCLOP command"
           "The SCLOP command performs scalar operations on each"
           "point in a data file.")
          ("outfile"
           "The result of the operation.")
          ("column number in outfile"
           "The output column number. It will defaulted to (1).")
          ("<"
           "Come on! Kick Kroonstein in the leg!")
          ("infile"
           "Input data to be operated on.")
          ("column number in infile"
           "Input file column. Default (1).")
          ("operator (+, -, * or /)"
           "Any of the single characters +, -, *, or /.")
          ("number"
           "Number to be added, subtracted, etc.")))

(slid
  (command slid
           (outfile)
           (outfile)
           <
           (infile)
           (infile)
           (number))
  (strings
    ("the SLID macro"
     "The SLID macro slides two files relative to each other")
    ("outfile 1"
     "")
    ("outfile 2"
     "")
    ("<"
     "Go, Ollie, go, go, go!")
    ("infile 1"
     "")
    ("infile 2"
     "")
    ("the relative timedelays for the files"
     "The number of timedelays the files will be shifted."
     "A positive value results in an upward shift of the column"
     "or a left shift in the plot.")))

(sptrf
  (command sptrf
```

```
            (symlist1 (default (AMP)))
            (outfile)
            (numlist1 (default (1)))
            <
            (infile .t)
            (symlist1 (default))
            (symbol)
            (numlist1 (default (1)))
            /
            (symbol)
            (numlist1 (default (1)))
            (symbol (default))
            (numlist1 (default (1))))

   (strings ("the SPTRF command"
             "the SPTRF command coputes the frequency response of"
             "a discrete time transfer function.")
            ("spectrum type switch"
             "The spectrum type switch should be either POW or AMP.")
            ("frequency response outfile"
             "The frequency response of the system in the infile.")
            ("column in outfile"
             "Column in which to put the frequency response.")
            ("" "")
            ("system infile"
             "A text file describing the system.")
            ("section" "")
            ("numerator polynomial type"
             "The numerator polynomial type can be either A, B, C or D.")
            ("numerator polynomial number" "")
            ("<"
             "Iran promotes hot stuff in Mecka.")
            ("denominator polynomial type"
             "The denominator polynomial type can be either A, B, C or D.")
            ("denominator polynomial number" "")
            ("file with frequency values" "")))

(stat
 (command stat
          (infile)
          (numlist1 (default (1)))
          (symbol (default)))

 (strings ("the STAT command"
           "The STAT command computes the sum, mean, variance,"
           "standard deviation, minimum and maximum for a time"
           "series file. Fancy, huh?")
          ("data infile"
           "The infile for which to compute the statistical properties.")
          ("column number"
           "The column number for the time series data. Defaults to (1).")
          ("extension tag"
           "If given, the name extension tag will cause global variables"
           "with the names SUM.EXT, STDEV.EXT, etc. to be set with the"
           "corresponding values.")))

(step
 (command step)

 (strings ("the STEP command"
           "The STEP command is a sub command of INSI. It generates a"
           "step signal.")))

(stop
 (command stop)

 (strings ("the STOP command"
           "The STOP command exits Idpac. Goodbye!")))

(switch
 (command switch
          (symbol)
          (symbol))
 (strings ("the SWITCH command"
           "The SWITCH command turns Idpac switches on/off.")
          ("symbol"
           "graph ...")
```

```
          ("symbol"
           "on off")))

 (trend
  (command trend
           (outfile)
           (numlist1 (default (1)))
           <
           (infile)
           (numlist1 (default (1)))
           (number)
           (number (default 1))
           (number (default ALL)))

  (strings ("the TREND command"
            "The TREND command is used to remove trends, i.e."
            "polynomial biases, in data files.")
           ("trend free outfile"
            "The outfile will contain the data with trends removed.")
           ("column in outfile"
            "This should be of the form (C), where C specifies the"
            "column number for the resulting data in the outfile. It"
            "defaults to (1).")
           ("<"
            "Was Reagan invited to the Shredding Party?")
           ("infile containing trends"
            "The infile should be a floating point data file, that may"
            "have trends in it.")
           ("indata column"
            "This should be of the form (C), where C specifies the"
            "column number for the data in the infile. It defaults to (1).")
           ("trend polynomial order"
            "The order of the trend polynomial to be fitted and removed.")
           ("starting point"
            "The first point to be corrected. Defaults to 1.")
           ("number of points"
            "The number of data points from the starting point to be"
            "affected. Defaults to all points in the file.")))

 (vecop
  (command vecop
           (outfile .d)
           (numlist1 (default (1)))
           <
           (infile .d)
           (numlist1 (default (1)))
           (symbol)
           (infile .d)
           (numlist1 (default (1))))

  (strings ("the VECOP command"
            "the VECOP command adds, subtracts, multiplies or divides"
            "two data vectors element by element.")
           ("outfile"
            "Where the result of the operation is placed.")
           ("column in outfile"
            "Default is (1).")
           ("<"
            "Nick Daniloff strikes again.")
           ("infile"
            "File to add, subtract, multiply or divide.")
           ("column in first infile"
            "Default is (1).")
           ("symbol"
            "should be +, -, * or /")
           ("infile"
            "File to add, subtract, multiply or divide.")
           ("column in first infile"
            "Default is (1).")))

 (write
  (command write
           (symbol (default)))

  (strings ("the WRITE command"
            "The WRITE command is used to print values of Idpac variables.")
           ("symbol"
```

```
            "The symbol you want to see tha value of. Defaults to all.")))
  (x
   (command x)

   (strings ("the X command"
             "The X command exits some subcommand levels. Viva Vieslander.")))

))
```

Listing of the file copy.l

```
; This function copies an object with a YAPS database. Tricky.
; This is a general problem with YAPS and object oriented programming,
; and copying is not encouraged. But sometimes it is necessary...

(defun copyyaps (old-db)
  (let (new)
    (setq new (make-instance 'script-flavor))
;   (>> new 'trace *yaps-trace*)
    (mapc #'(lambda (x)
              (<- new 'installp x))
            (mapcar #'(lambda (y)
                        (<< y 'name))
                    (<< old-db 'rules)))
    (>> new 'cycle (<< old-db 'cycle))
    (>> new 'strategy (<< old-db 'strategy))
    (>> new 'trace (<< old-db 'trace))
    (>> new 'pbreaks (<< old-db 'pbreaks))
    (mapc #'(lambda (x)
              (<- new 'fact x))
            (mapcar #'(lambda (y)
                        (<< y 'value))
                    (reverse (<< old-db 'facts))))
    (let ((old-conflict-set (<< old-db 'conflict-set))
          (new-conflict-set (<< new 'conflict-set)))
      (>> new 'conflict-set
          (change-conflict-set new-conflict-set old-conflict-set)))
    new))

; ------------------------------------------------------------------------

(defun change-conflict-set (new-list old-list)
  (cond ((null new-list) nil)
        ((member-conflict (car new-list) old-list)
         (cons (car new-list)
               (change-conflict-set (cdr new-list) old-list)))
        (t (change-conflict-set (cdr new-list) old-list))))

(defun member-conflict (binding bind-list)
  (cond ((null bind-list) nil)
        ((and (eq (<< binding 'back-link)
                  (<< (car bind-list) 'back-link))
              (equal (length (<< binding 'fact-list))
                     (length (<< (car bind-list) 'fact-list)))
              (equal-values (mapcar #'(lambda (x) (<< x 'value))
                                    (<< binding 'fact-list))
                            (mapcar #'(lambda (y) (<< y 'value))
                                    (<< (car bind-list) 'fact-list))))
         bind-list)
        (t (member-conflict binding (cdr bind-list)))))

(defun equal-values (list1 list2)
  (cond ((null list1) t)
        ((equal (car list1) (car list2))
         (equal-values (cdr list1) (cdr list2)))
        (t nil)))

; ------------------------------------------------------------------------

(def net-node-primary-remove-fact-method
  (lambda (self fact)
    ((lambda (g00120)
             (do (($$val)
                  (b (and (setq g00120
                                (symeval-in-instance self 'bind-nodes))
```

```
                    (car g00120))
               (and (setq g00120 (cdr g00120)) (car g00120))))
          ((null g00120) nil)
          (<- b 'remove-fact fact)))

     nil)
    (and (symeval-in-instance self 'path)
         ((lambda (val next)
                  (setq next
                        (and val
                             (cdr
                              (assq val
                                    (symeval-in-instance self
                                                         'assoc-branch)))))
                  (and next (<- next 'remove-fact fact))
                  (and (symeval-in-instance self 'else-branch)
                       (<- (symeval-in-instance self 'else-branch)
                           'remove-fact
                           fact)))

          (car
           (errset (apply (symeval-in-instance self 'path)
                          (rplaca ADD-CONS (<< fact 'value)))
                   nil))
          nil))))

; ------------------------------------------------------------------------

(def net-node-primary-add-fact-method
  (lambda (self fact)
    ((lambda (g00119)
             (do (($$val)
                  (b (and (setq g00119
                                (symeval-in-instance self 'bind-nodes))
                          (car g00119))
                     (and (setq g00119 (cdr g00119)) (car g00119))))
                 ((null g00119) nil)
                 (<- b 'add-fact fact)))

     nil)
    (and (symeval-in-instance self 'path)
         ((lambda (val next)
                  (setq next
                        (and val
                             (cdr
                              (assq val
                                    (symeval-in-instance self
                                                         'assoc-branch)))))
                  (and next (<- next 'add-fact fact))
                  (and (symeval-in-instance self 'else-branch)
                       (<- (symeval-in-instance self 'else-branch)
                           'add-fact
                           fact)))

          (car
           (errset (apply (symeval-in-instance self 'path)
                          (rplaca ADD-CONS (<< fact 'value)))
                   nil))
          nil))))
```

Listing of the file database.l

```
; This is the database, the central datastructure of the system.
; All data common to all modules (objects) is located here.

 (defflavor database-flavor
  (scripts
;
; Different superscripts
;
   actual-superscripts
   restart-superscript
   superscript-count
;
   script-macros
   intrac-variables       ; ((nplx. 100) .... (wmax. 10)) etc.
   command-history
```

```
    menu-list               ; defined in (ihs)
    command-grammar
    internal-command-names
    command-lists           ; for efficiency ... ((conv (c o n v)) ...)
                            ; should be implemented in another way
    other-dict-words
    allowed-commands        ; commands which are always allowed
    special-commands        ; commands which must call procedures in the
                            ; interface as well as in Idpac
;
;   File system
;
    default-file-spec
    file-system
    file-number
    tab-number
;
; Internal states
;
    now-ptime               ; to get info about the cpu-time of the lisp
    last-ptime
    now-cons                ; how many cons-cells used
    last-cons
    internal-states
   )
   (abbreviate-flavor))

; Exported methods ------------------------------------------------------------

; Match-actaul-superscripts distributes the command to all the
; different scripts in the actual superscript. The YAPS databases
; are NOT updated. (See match-actual-superscript-finally below.)
; Used when matching a command for the first time.

(defmethod (database-flavor :match-actual-superscripts) (command)
  (prog (temp list res scn)
    (setq list (<< database 'actual-superscripts))
    loop
    (cond
      ((null list)
        (return nil)))
    (setq temp (car list))
    (setq
      res
      (apply 'append
        (mapcar
         '(lambda (x)
            (let
              ((res
                (<- (<- (eval temp) ':script x) ':match-command command)))
              (cond
                (res (list (cons x res)))
                 (t nil))))
          (<< (eval temp) 'scripts))))
    (cond
      ((null res)
        (setq list (cdr list))
        (go loop)))

    (>> database 'actual-superscripts
      (cons temp (wegnehmen temp (<< database 'actual-superscripts))))
    (return res)))

; Match-restart-superscript distributes a command to all the
; scripts in the restart superscript.

(defmethod (database-flavor :match-restart-superscript) (command)
  (prog (temp res)
    (setq
      temp
      (<- (eval (<< database 'restart-superscript))
        ':copy-superscript 'temp))
    (mapc
      '(lambda (x) (<- (eval (concat 'temp- x)) ':initialize))
      (<< temp 'scripts))
    (setq
      res
```

```
      (apply 'append
        (mapcar
         '(lambda (x)
            (let
              ((res (<- (eval (concat 'temp- x))
                ':match-command command)))
              (cond
                (res (list (cons x res)))
                 (t nil))))
          (<< temp 'scripts))))
    (cond
      (res
       (>> database 'actual-superscripts
         (cons
           (concat 'actual-superscript- (<< database 'superscript-count))
           (<< database 'actual-superscripts)))
       (set
         (concat 'actual-superscript- (<< database 'superscript-count))
         (<- temp ':copy-superscript
           (concat 'actual-script- (<< database 'superscript-count))))
        (>> database 'superscript-count
          (add1 (<< database 'superscript-count)))))
    (<- self ':delete-superscript 'temp)
    (return res)))

; Used for the final matching. All YAPS-db's are updated.

(defmethod (database-flavor :match-actual-superscripts-finally) (command)
  (prog (temp list res scn)
    (setq list (<< database 'actual-superscripts))
    loop
    (cond
      ((null list)
        (return nil)))
    (setq temp (car list))
    (setq
      res
      (apply 'append
        (mapcar
         '(lambda (x)
            (let
              ((res
                (<- (<- (eval temp) ':script x) ':match-command-finally
                  command)))
              (cond
                (res (list (cons x res)))
                 (t nil))))
          (<< (eval temp) 'scripts))))
    (cond
      ((null res)
        (setq list (cdr list))
        (go loop)))

    (>> database 'actual-superscripts
      (cons temp (wegnehmen temp (<< database 'actual-superscripts))))
    (return res)))

; Distributes a command for final matching. YAPS-db's are updated.

(defmethod (database-flavor :match-restart-superscript-finally) (command)
  (prog (temp res)
    (setq
      temp
      (<- (eval (<< database 'restart-superscript))
        ':copy-superscript 'temp))
    (mapc
      '(lambda (x) (<- (eval (concat 'temp- x)) ':initialize))
      (<< temp 'scripts))
    (setq
      res
      (apply 'append
        (mapcar
         '(lambda (x)
            (let
              ((res (<- (eval (concat 'temp- x))
                ':match-command-finally command)))
              (cond
```

```
                (res (list (cons x res)))
                 (t nil))))
            (<< temp 'scripts))))
    (cond
      (res
       (>> database 'actual-superscripts
         (cons
           (concat 'actual-superscript- (<< database 'superscript-count))
           (<< database 'actual-superscripts)))
       (set
         (concat 'actual-superscript- (<< database 'superscript-count))
         (<- temp ':copy-superscript
           (concat 'actual-script- (<< database 'superscript-count))))
       (>> database 'superscript-count
         (add1 (<< database 'superscript-count)))))
    (<- self ':delete-superscript 'temp)
    (return res)))

; Distribute a message to all scripts, via the superscripts to
; start the expert system, if necessary.

(defmethod (database-flavor :perform-kscalls) ()
  (<- (eval (car (<< database 'actual-superscripts))) ':perform-kscalls))

; Distribute a message to all scripts, via the superscripts to
; carry out all (assign Y X) clauses

(defmethod (database-flavor :make-assigns) ()
  (<- (eval (car (<< database 'actual-superscripts))) ':make-assigns))

; Produces a dump of all scripts, with their current states, etc.

(defmethod (database-flavor :dump-scripts) ()
  (<- uinterf ':line 79)
  (<- uinterf ':writeln '(""))
  (<- uinterf ':writeln '("ACTUALS: "))
  (mapc
    '(lambda (x)
       (<- (eval x) ':dump-scripts))
    (<< database 'actual-superscripts))
  (<- uinterf ':writeln '("RESTART: "))
  (<- (eval (<< database 'restart-superscript)) ':dump-scripts))

; The input is a command (maybe only a part of a command name).
; Out comes the full command description of the command grammar.

(defmethod (database-flavor :get-command-specification) (com)
  (let ((cs (<- self ':get-abbreviated-word com command-lists)))
    (cond
      ((equal (car cs) 'fail) '|Erroneous command: |)
      ((and
        (not (atom cs))
        (equal (car cs) 'ambiguous))
        '|Ambiguous command: |)
      (t
        (cdadr (assq (cadr cs) command-grammar))))))

; Methods for getting and setting info for use by the menu.

(defmethod (database-flavor :get-lisp-system-trace) ()
  (equal t (cadr (assoc 'lisp-system-trace (<< database 'internal-states)))))

(defmethod (database-flavor :get-fruit) ()
  (cadr (assoc 'fruit (<< database 'internal-states))))

(defmethod (database-flavor :get-prompt) ()
  (cadr (assoc 'prompt (<< database 'internal-states))))

(defmethod (database-flavor :get-trace) ()
  (equal t (cadr (assoc 'trace-on (<< database 'internal-states)))))

(defmethod (database-flavor :get-pling) ()
  (equal t (cadr (assoc 'pling (<< database 'internal-states)))))

(defmethod (database-flavor :get-trace-scripts) ()
  (equal t (cadr (assoc 'trace-scripts-on (<< database 'internal-states)))))
```

```
(defmethod (database-flavor :get-user-state) ()
  (let
    ((us (cadr (assoc 'user-state (<< database 'internal-states)))))
    (cond
      ((equal us 'e) 'expert)
      ((equal us 'b) 'beginner))))

(defmethod (database-flavor :set-user-state) (state)
  (>> database 'internal-states
    (mapcar
      '(lambda (x)
         (cond
           ((equal (car x) 'user-state)
             (list 'user-state
               (cadr (assoc state '((expert e)
                                    (beginner b))))))
           ((and (equal (car x) 'expert) (equal state 'beginner))
             '(expert f))
           (t x)))
      (<< database 'internal-states))))

(defmethod (database-flavor :user-is-expert?) ()
  (equal (cadr (assoc 'expert (<< database 'internal-states))) 't))

(defmethod (database-flavor :get-yaps-debug) ()
  (equal t (cadr (assoc 'yaps-debug (<< database 'internal-states)))))

; Get-short-help-text gets a short help text from the database. This help
; text is the one normally used when referring to a command, file or parameter.

(defmethod (database-flavor :get-short-help-text) (command-name arg-number)
  (caar
    (nthcdr
      arg-number
      (cdaddr (assoc command-name (<< database 'command-grammar))))))

; Get-long-help-text gets a long help text from the database. This help text
; is supposed to be used when the user wants an explanation of the shorter
; help text. (He does this by typing a '?'.)

(defmethod (database-flavor :get-long-help-text) (command-name arg-number)
  (cdar
    (nthcdr
      arg-number
      (cdaddr (assoc command-name (<< database 'command-grammar))))))

(defmethod (database-flavor :get-clause-nr) (command-name arg-number)
  (car
    (nthcdr
      arg-number
      (cdadr (assoc command-name (<< database 'command-grammar))))))

; Miscellaneous methods for handling superscripts.

(defmethod (database-flavor :delete-superscript) (name)
  (cond
    ((eval name)
      (<- (eval name) ':delete)
      (set name nil))))

(defmethod (database-flavor :purge-superscripts) ()
  (setq
    actual-superscripts
    (mapzap
      '(lambda (x)
         (cond
           ((some 'cdr (<- (eval x) ':next-commands))
             x)
           (t
             nil)))
      actual-superscripts)))

; Methods for fetching the short and long help texts from
; the script descriptions.

(defmethod (database-flavor :get-script-name) (script-name)
  (caaddadr (assoc script-name (<< database 'scripts))))
```

```
(defmethod (database-flavor :get-script-desc) (script-name)
  (cdaddadr (assoc script-name (<< database 'scripts))))

(defmethod (database-flavor :update-command-history) (command)
  (setq command-history (append1 command-history command)))

; Methods for handling script-macros

(defmethod (database-flavor :get-scriptmacro) (name)
  (assoc name (<< database 'script-macros)))

; Returns the body of a scriptmacro

(defmethod (database-flavor :body) (name)
  (let ((x (<- database ':get-scriptmacro name)))
    (cond
      ((and
        (equal (caadr x) 'in)
        (equal (caaddr x) 'out)) (cdddr x))
      ((or
        (equal (caadr x) 'in)
        (equal (caadr x) 'out)) (cddr x)))))

; Returns the in-part/out-part of script macros.

(defmethod (database-flavor :ins-of-macro) (name)
  (cdr (assoc 'in (cdr (<- database ':get-scriptmacro name)))))

(defmethod (database-flavor :outs-of-macro) (name)
  (cdr (assoc 'out (cdr (<- database ':get-scriptmacro name)))))

; Methods for handling the internal commands.

(defmethod (database-flavor :internal-command) (command)
  (<- self (concat ':internal-command- (car command)) command))

; Special command stuff. Used by LET and FREE.

(defmethod (database-flavor :add-intrac-variable) (var value)
  (setq intrac-variables (cons (list var value) intrac-variables)))

(defmethod (database-flavor :delete-intrac-variable) (var)
  (setq
    intrac-variables
    (apply 'append
      (mapcar
        '(lambda (x)
           (cond ((equal var (car x)) nil) (t (list x))))
        intrac-variables))))

(defmethod (database-flavor :special-command) (command)
  (cond
    ((member command (<< database 'special-commands))
      (<- self (concat ':special-command- (car command)) command))))

; Initialization procedure

(defmethod (database-flavor :initialize) ()
  (setq last-ptime (ptime))
  (setq now-ptime last-ptime)
  (setq last-cons (purcopy (car (opval 'list)))) ; ?????
  (setq now-cons last-cons)
  (setq command-lists
    (mapcar
      'car
      command-grammar))
  (setq internal-command-names
    (mapcar 'car internal-commands))
  (setq file-number 0)
  (setq file-system (<- self ':file-symbol '/))
  (setq default-file-spec '/)
  (set
    (<- self ':file-symbol (<< self 'default-file-spec))
    (make-instance 'dir-file-flavor
      'name      (<< self 'default-file-spec)
      'number    file-number
```

```
      'file-type 'dir-file-flavor
      'file-list 'nil))

  (setq tab-number 15)) ; used when plotting the file tree

; Internal methods ------------------------------------------------------------

; Here may be found a plethora of methods that are only used by other
; methods in the database module. This is the place for subroutines proper.

(defmethod (database-flavor :ptime) ()
  (setq last-cons now-cons)
  (setq now-cons (purcopy (car (opval 'list)))) ; very very very strange indeed
  (setq last-ptime now-ptime)
  (setq now-ptime (ptime)))

(defmethod (database-flavor :get-script) (name)
  (eval
    (concat
      (<< (eval (car actual-superscripts)) 'script-concat-name)
      '-
      name)))

; Three procedures for handling of shortforms of commands.
; The commands are defined in the list command-lists like
; (setq command-lists '((conv (c o n v)) (bode (b o d e)) ...))
; These procedures are now replaced by abbreviate-flavor, but
; not everywhere. They are old and should not be used.

(defmethod (database-flavor :match-rec) (com)
  (let ((y (explode com)))
    (apply 'append
      (mapcar
        '(lambda (x)
           (<- self ':matchit (car x) (cadr x) y))
        command-lists))))

(defmethod (database-flavor :matchit) (complete-command pa li)
  (cond
    ((and (null pa) (null li))
      (list (list 'exact-match complete-command)))
    ((null li)
      (list (list 'partial-match complete-command)))
    ((equal (car li) (car pa))
      (<- self ':matchit complete-command (cdr pa) (cdr li)))
    (t
      nil)))

(defmethod (database-flavor :abbreviated-command) (com)
  (let ((x (<- self ':match-rec com)))
    (cond
      ((null x)
        "Invalid command")
      ((equal (length x) 1)
        (cadar x))
      ((assq 'exact-match x)
        (cadr (assq 'exact-match x)))
      (t
        (list "Ambiguous command" x)))))

; File handler ----------------------------------------------------------------

; This stuff handles the database of files. Called from the
; Query module, it keeps the files in a directed graph structure.
; There is some support for a complete directory structure, but
; this was skipped, and is not in use. Avoid it if you can!

(defmethod (database-flavor :fs) (file)
  (cond
    ((equal (atomcar file) '/)
      (concat "file-" file))
    (t
      (concat "file-"
        (cond
          ((equal default-file-spec '/) '/)
          (t (concat default-file-spec '/)))
        file))))
```

```
(defmethod (database-flavor :file-symbol) (f)
  (concat 'file- (<- self ':full-file-name f)))

(defmethod (database-flavor :expand) (l)
  (<- self ':dir-list-to-atom
    (<- self ':remove-dd
      (<- self ':list-of-dirs l))))

(defmethod (database-flavor :full-file-name) (file)
  (let ((x default-file-spec))
    (cond
      ((equal (atomcar file) '/)
        (<- self ':expand file))
      (t
        (<- self ':expand
          (concat
            (cond
              ((equal x '/) '/)
              (t (concat x '/)))
            file))))))

(defmethod (database-flavor :full-dir-name) (f-name)
  (do
    ((x (reverse (explode (<- self ':full-file-name f-name))) (cdr x)))
     ((equal (car x) '/)
       (<- self ':expand
         (implode (reverse (cond ((cdr x)) (t '(/))))))))
     (comment)))

(defmethod (database-flavor :dir-list-to-atom) (l)
  (cond
    ((null l) '/)
    (t
      (concatl (mapcar '(lambda (x) (concat '/ x)) l)))))

(defmethod (database-flavor :list-of-dirs) (ds)
  (do
    ((li (explode ds) (cdr li))
     (result nil)
     (tmp nil))
    ((null li)
      (cond
        ((null tmp) result)
        (t (append1 result (concatl tmp)))))
    (cond
      ((equal (car li) '/)
        (setq result (append result (cond (tmp (list (concatl tmp))))))
        (setq tmp nil))
      (t
        (setq tmp (append1 tmp (car li)))))
      (comment)))

(defmethod (database-flavor :remove-dd) (l)
  (do
    ((res l (rem-dd res)))
    ((not (member '.. res))
      res)
    (comment)))

(defmethod (database-flavor :rem-dd) (l)
  (cond
    ((null l)
      '())
    ((equal (car l) '..)
      (cdr l))
    ((and
       (cdr l)
       (equal (cadr l) '..))
      (cddr l))
    (t
      (cons (car l) (rem-dd (cdr l))))))

; ----------------------------------------------------------------------

(defmethod (database-flavor :new-file-parents-and-kids)
  (f-name par-list command)
```

```
  (<- self ':create-file 'data-file-flavor f-name)
  (<- (eval (<- self ':file-symbol f-name)) ':command command)
  (mapc
    '(lambda (x)
       (<-
         (eval (<- self ':file-symbol f-name))
        ':create-parent
         (<- self ':full-file-name x)))
  par-list)
  (mapc
    '(lambda (x)
       (<-
         (eval (<- self ':file-symbol x))
        ':create-child
         (<- self ':full-file-name f-name)))
  par-list))

(defmethod (database-flavor :write-children) (f-name)
  (cond
    ((<- self ':file-exists? f-name)
      (<-
        uinterf
        ':write-relatives
        (<- (eval (<- self ':file-symbol f-name)) ':children)
        0))
    (t
      (<- uinterf ':writeln (list "No such file:" f-name)))))

(defmethod (database-flavor :write-parents) (f-name)
  (cond
    ((<- self ':file-exists? f-name)
      (<-
        uinterf
        ':write-relatives
        (<- (eval (<- self ':file-symbol f-name)) ':parents)
        0))
    (t
      (<- uinterf ':writeln (list "No such file:" f-name)))))

(defmethod (database-flavor :dir) (p)
  (<-
    (eval (<- self ':file-symbol (cond (p) (t default-file-spec)))) ':dir))

; ----------------------------------------------------------------------------

(defmethod (database-flavor :create-directory) (dir-name)
  (let*
    ((fd (full-file-name dir-name)))
    (do
      ((dl (<- self ':list-of-dirs fd) (cdr dl))
       (ad nil (concat (cond (ad (concat ad '/)) (t '/)) (car dl))))
      ((null dl) (<- self ':create-file 'dir-file-flavor ad))
      (cond
        ((null ad))
        ((boundp (<- self ':file-symbol ad)))
        (t
          (<- self ':create-file 'dir-file-flavor ad)))
      (comment))))

(defmethod (database-flavor :create-file) (f-type f-name)
  (let*
    ((fn (<- self ':full-file-name f-name))
     (fd (<- self ':full-dir-name fn)))
    (cond
      ((boundp (<- self ':file-symbol fd))
        (cond
          ((boundp (<- self ':file-symbol fn))
            (<- self ':delete-file fn)))
        (setq
          created-files                                    ; GLOBAL variable
          (cons (<- self ':file-symbol fn) created-files))
        (set (<- self ':file-symbol fn)
          (make-instance
            f-type
            'name      fn
            'file-type f-type
            'number    (setq file-number (1+ file-number))))
```

```
        (<- (eval (<- self ':file-symbol fd)) ':insert-file fn))
       (t
         (<- uinterf ':writeln (list "Directory does not exist."))))))

(defmethod (database-flavor :delete-file) (f-name)
  (let*
    ((fn (<- self ':full-file-name f-name))
      (fd (<- self ':full-dir-name fn))
      (fs (<- self ':file-symbol fn)))
    (cond
      ((boundp fs)
        (let*
          ((new-name
             (concat (<< (eval fs) 'name) "_D" (<< (eval fs) 'number)))
           (fs-new-name (<- self ':file-symbol new-name)))
          (>> (eval fs) 'name new-name)
          (setq created-files (cons fs created-files))    ; GLOBAL variable
          (set fs-new-name (eval fs))
          (mapc
            '(lambda (x)
               (<- (eval (<- self ':file-symbol x)) ':delete-parent fn)
               (<- (eval (<- self ':file-symbol x)) ':create-parent new-name))
            (<- (eval fs-new-name) ':children?))
          (mapc
            '(lambda (x)
               (<- (eval (<- self ':file-symbol x)) ':delete-child fn)
               (<- (eval (<- self ':file-symbol x)) ':create-child new-name))
            (<- (eval fs-new-name) ':parents?))
          (<-
            (eval (<- self ':file-symbol fd))
            ':remove-file fn)
          (<-
            (eval (<- self ':file-symbol fd))
             ':insert-file new-name)
          (setq created-files (wegnehmen fn created-files))   ; GLOBAL variable
          (makunbound fn)
          t))
       (t
         (<- uinterf ':writeln
           (list "You can't delete a non-existing file, stupid."))
         nil))))

(defmethod (database-flavor :file-exists?) (f-name)
  (let*
    ((fn (<- self ':full-file-name f-name))
      (fd (<- self ':full-dir-name fn))
      (fs (<- self ':file-symbol fn)))
    (boundp fs)))

(defmethod (database-flavor :set-def) (dir-spec)
  (cond
    ((equal (atomcar dir-spec) '/)
      (setq default-file-spec (<- self ':expand dir-spec)))
    (t
      (setq default-file-spec
        (<- self ':expand (concat default-file-spec "/" dir-spec))))))

(defmethod (database-flavor :sh-def) ()
  (<- uinterf ':writeln (list default-file-spec)))

; Internal command methods ------------------------------------------------

; Each internal command has a method associated to it. These
; methods are found here. If a new internal command is added,
; this is the right place to put the corresponding code.

(defmethod (database-flavor :internal-command-menu) (command)
  (>> database 'internal-states
    (<- uinterf ':menu
      (<< database 'menu-list) (<< database 'internal-states)))
  (cond
    ((and
      (<- self ':user-is-expert?)
      (equal (<- self ':get-user-state) 'beginner))
     (<- self ':internal-command-think '(think)))))

(defmethod (database-flavor :internal-command-think) (command)
```

```
  (cond
    ((<- self ':user-is-expert?)
      (<- self ':set-user-state 'beginner)
      (<-
        (eval (car (<< self 'actual-superscripts))) ':fact '(question enable))
      (<- (eval (car (<< self 'actual-superscripts))) ':run))
    (t
      (<- uinterf ':writeln
        (list "THINK has no effect in" (<- self ':get-user-state) "mode."))))))

(defmethod (database-flavor :internal-command-lisp) (command)
  (setq user-top-level nil))

(defmethod (database-flavor :internal-command-children) (command)
  (<- self ':write-children
    (concat (cadaddr command) '|.| (caadadr command))))

(defmethod (database-flavor :internal-command-parents) (command)
  (<- self ':write-parents
    (concat (cadaddr command) '|.| (caadadr command))))

(defmethod (database-flavor :internal-command-dir) (command)
  (<- self ':dir '/))

(defmethod (database-flavor :internal-command-?) (command)
  (let*
    ((x (eval (car actual-superscripts)))
     (res (<- x ':get-yaps-info)))                          ; bug in flavors
      (<- uinterf ':write-yaps-info res)))

(defmethod (database-flavor :internal-command-??) (command)
  (let*
    ((x (eval (car actual-superscripts)))
     (res (<- x ':next-commands)))                          ; bug in flavors
      (<- uinterf ':write-next-commands res)))

(defmethod (database-flavor :internal-command-explain) (command)
  (<- dictionary ':explain (cadadr command)))

; Methods for writing the file system to disc and back again.
; Restore completely overwrites any intermediate stuff.

(defmethod (database-flavor :internal-command-dumpfiles) (command)
  (let
    ((port (fileopen (concat (cadadr command) '|.dmp|) "w")))
;   (pp (P port) created-files)
    (patom "(setq " port)               ;
    (terpr port)                        ;
    (patom "  created-files" port)      ;
    (terpr port)                        ;
    (patom "  (append" port)            ;
    (terpr port)                        ;
;   (patom "    created-files" port)    ;
    (patom "    nil          " port)    ;
    (terpr port)                        ;
    (patom created-files port)          ;
    (patom "))")                        ;
    (terpr port)                        ;
    (terpr port)
    (mapcar
      '(lambda (x)
         (patom
           "(>> database 'file-number (1+ (<< database 'file-number)))"
           port)
         (terpr port)
         (terpr port)
         (patom "(setq " port)
         (patom x port)
         (terpr port)
         (cond
           ((equal (<< (eval x) 'file-type) 'data-file-flavor)
             (patom "  (make-instance 'data-file-flavor" port)
             (terpr port)
             (patom "    'name      '" port)
               (patom (<< (eval x) 'name) port)
               (terpr port)
             (patom "    'number    '" port)
```

```
              (patom (<< (eval x) 'number) port)
              (terpr port)
            (patom "    'file-type '" port)
              (patom (<< (eval x) 'file-type) port)
              (terpr port)
            (patom "    'parents   '" port)
              (patom (<< (eval x) 'parents) port)
              (terpr port)
            (patom "    'children  '" port)
              (patom (<< (eval x) 'children) port)
              (terpr port)
            (patom "    'command   '" port)
              (patom (<< (eval x) 'command) port)
              (patom "))" port)
              (terpr port))
          ((equal (<< (eval x) 'file-type) 'dir-file-flavor)
            (patom "  (make-instance 'dir-file-flavor" port)
            (terpr port)
            (patom "    'name      '" port)
              (patom (<< (eval x) 'name) port)
              (terpr port)
            (patom "    'number    '" port)
              (patom (<< (eval x) 'number) port)
              (terpr port)
            (patom "    'file-type '" port)
              (patom (<< (eval x) 'file-type) port)
              (terpr port)
            (patom "    'file-list '" port)
              (patom (<< (eval x) 'file-list) port)
              (patom "))" port)
              (terpr port))))
    created-files)
  (resetio)))

(defmethod (database-flavor :internal-command-restorefiles) (command)
  (let ((tmp (concat (cadadr command) '|.dmp|)))
    (cond
      ((probef tmp)
        (load tmp))
      (t
        (<- uinterf ':writeln (list "Cannot find" tmp))))))

; Show a value of a menu parameter.

(defmethod (database-flavor :internal-command-show) (command)
  (let
    ((aw
      (<- self ':get-abbreviated-word
        (cadadr command)
        '(fruit scripts dump system history))))
  (cond
    ((equal (car aw) 'success)
      (cond
        ((equal (cadr aw) 'fruit)
          (<-
            uinterf
            ':writeln
            (list "The fruit is" (<- self ':get-fruit))))
        ((equal (cadr aw) 'scripts)
          (<- uinterf
            ':write-active-scripts
            (mapcar
              '(lambda (x)
                 (<- (eval x) ':next-commands))
              (<< database 'actual-superscripts))))
        ((equal (cadr aw) 'dump)
          (<- self ':dump-scripts))
        ((equal (cadr aw) 'history)
          (mapc
            '(lambda (nr x)
              (<- uinterf ':write '(,nr |.|))
              (<- uinterf ':patom x)
              (<- uinterf ':writeln '("")))
            (cdr (number-list (1+ (length command-history))))
            command-history))
        ((equal (cadr aw) 'system)
          (<- uinterf ':line 79) (terpr)
```

```
          (let
            ((pt (car now-ptime))
             (pto (car last-ptime))
             (gct (cadr now-ptime))
             (gcto (cadr last-ptime))
             (nc  now-cons)
             (lc  last-cons))
          (<- uinterf ':writeln
            `("Processor time: " ,(time-sym (- pt gct))
              "Delta time: "  ,(time-sym (- (- pt gct) (- pto gcto)))))
          (<- uinterf ':writeln
            `("Garbage collection time:" ,(time-sym gct)
              "Delta time: " ,(time-sym (- gct gcto))))
          (<- uinterf ':writeln (list "Cons: "  now-cons
                                      "Delta: " (- now-cons last-cons))))
          (<- uinterf ':writeln
            `("Hunks: " ,(apply '+
               (mapcar
                 'car
                 (mapcar
                   'opval
                   '(hunk0 hunk1 hunk2 hunk3 hunk4 hunk5 hunk6))))))
          (<- uinterf ':writeln `("Number of garbs: " ,$gccount$))
          (<- uinterf ':line 79) (terpr))))
    ((equal (car aw) 'ambiguous)
      (<- uinterf ':writeln `("Ambiguous parameter" ,(cdr aw))))
    ((equal (car aw) 'fail)
      (<- uinterf ':writeln '("No such parameter"))))))

; Special command methods ------------------------------------------------------

; Here is the proper place for methods needed by special commands.
; These are commands to Idpac, that demands a special meythod to
; be run. LET, e.g., affects the state of a variable in Idpac AND
; of a variable in (ihs).

(defmethod (database-flavor :special-command-let) (command)
  (<- self ':add-intrac-variable (cadadr command) (cadadddr command)))

(defmethod (database-flavor :special-comamnd-delet) (command)
  (<- self ':delete-file (cadadr command)))

(defmethod (database-flavor :special-command-free) (command)
  (<- uinterf ':writeln (list "Oh yes, I feel free!")))
```

Listing of the file dict.l

```
; This flavor implements the directory module. The directory is initialized
; with the method :initialize, where the instance variable is set to
; information from other parts of the system.
; To get help on a word, (mostly an explanation), send the message
; :explain with a word as argument. The dictionary gives multiple explanations
; on ambiguous words. The result is sent to the uinterf to be handled there.

; Words that should be put in the dictionary is entered in the file
; dictwords.l

(defflavor dictionary-flavor
  (dictionary)
  ())

; ------------------------------------------------------------------------------

; The method which should be used by other objects.

(defmethod (dictionary-flavor :explain) (word)
  (let ((lict (<- self ':find-word-list word)))
    (cond
      (lict
        (<- uinterf ':write-dictionary-list lict))
      (t
        (<- uinterf ':writeln (list "No help for" word))))))

; -- Internal methods ------------------------------------------------------

(defmethod (dictionary-flavor :match?) (pattern list)
  (cond
```

```
    ((null list) t)
    ((equal (car list) (car pattern))
      (<- self ':match? (cdr pattern) (cdr list)))
    (t nil)))

(defmethod (dictionary-flavor :match-word?) (word dict-entry)
  (cond
    ((<- self ':match? (cadr dict-entry) (explode word))
      (list (caddr dict-entry)))))

(defmethod (dictionary-flavor :find-word-list) (word)
  (apply 'append
    (mapcar
      '(lambda (x) (<- self ':match-word? word x))
      dictionary)))

; ---------------------------------------------------------------------------

; Routines for the initialization.


; The explanations of what the scripts do.

(defmethod (dictionary-flavor :scripts-dict-list) ()
  (mapcar
    '(lambda (x) (list (car x) (caddadr x)))
     (<< database 'scripts)))

; Explanations of commands.

(defmethod (dictionary-flavor :command-dict-list) ()
  (mapcar
    '(lambda (x) (list (car x) (cadaddr x)))
    (<< database 'command-grammar)))

; Explanation of other terminology.

(defmethod (dictionary-flavor :other-dict-words) ()
  (<< database 'other-dict-words))

(defmethod (dictionary-flavor :initialize) ()
  (setq dictionary
    (mapcar
      '(lambda (x)
         (list (car x) (explode (car x)) (cadr x)))
      (append
        (<- self ':scripts-dict-list)
        (<- self ':command-dict-list)
        (<- self ':other-dict-words)))))
```

Listing of the file dictwords.l

```
; This file contains explanations of word and terms. These explanations
; are shown with the 'explain' command.

(setq other-dict-words '(
  (internalcommands
    ,(cons
       "internal commands is obtained by asking about"
       (mapcar 'car internal-commands)))
  (fruit
    ("fruit" "The idpac fruit can be set to POTATIS."))
  (lags
    ("lags" "This term signifies a number of data points to be"
            "involved in a computation. It may be the size of"
            "the time window for a FFT or convolution computation."
            "Don't be afraid to experiment with a few different"
            "values if you don't know what number to give."))
  (larsson
     ("Larsson" "Larsson and Persson wrote the program."))
  (persson
     ("Persson" "Larsson and Persson wrote the program."))
  (outliers
    ("outliers" "An outlier is a data point, which is far out."
                "The reason for this may be some kind of measurement"
                "error. If you want to avoid outliers, try to use"
                "other parts of data. If that is not possible, you"
```

```
                "may replace an erratic data point with interpolation."
                "This is done with the PLMAG command."))
  (trends
    ("trends" "A trend is a finite order polynomial forming part of"
              "a data signal. The most usual kind of trend is one of"
              "zero order, i.e., a bias. Trends should always be"
              "removed before an estimation is started, or the results"
              "will be in error. A bias, e.g., will show up as an"
              "integrator. It is very difficult to remove higher order"
              "trends, so it is recommended that you try to use other"
              "data instead."))
  (help
    ("Ask for internal commands."))))
```

Listing of the file file.l

```
; This is the file system which is used in (ihs).
;
; There is support for a directory structure for files in a unix-like
; manner, but this is not used in the system.

(defflavor file-flavor
  (name number)
  ())

; ----------------------------------------------------------------------------

(defflavor data-file-flavor
  (file-type
   parents        ; a list of all files immediatelyneeded to create the file
   children       ; a list of all files immediately created from the file
   command)
  (file-flavor))

(defmethod (data-file-flavor :file-type?) ()
  file-type)

(defmethod (data-file-flavor :parents?) ()
  parents)

(defmethod (data-file-flavor :children?) ()
  children)

(defmethod (data-file-flavor :command?) ()
  command)

(defmethod (data-file-flavor :command) (com)
  (setq command com))

(defmethod (data-file-flavor :create-parent) (parent)
  (cond
    ((member parent parents)
      (<- uinterf ':writeln
        (list "CREATE-PARENT:" "PARENT" parent "EXISTS IN" name))
      nil)
    (t
      (setq parents (cons parent parents)))))

(defmethod (data-file-flavor :delete-parent) (parent)
  (cond
    ((member parent parents)
      (setq parents (wegnehmen parent parents)))
    (t
      (<- uinterf ':writeln
        (list "DELETE-PARENT:" "PARENT" parent "DOES NOT EXIST IN" name))
      nil)))

(defmethod (data-file-flavor :parents) ()
  (cond
    ((null parents)
      (list name))
    (t
      (cons
        name
        (list
          (mapcar
            '(lambda (x) (<- (eval (<- database ':file-symbol x)) ':parents))
```

```
            parents))))))

(defmethod (data-file-flavor :create-child) (child)
  (cond
    ((member child children)
      (<- uinterf ':writeln
        (list "CREATE-CHILD:" "CHILD" child "EXISTS IN" name))
      nil)
    (t
      (setq children (cons child children)))))

(defmethod (data-file-flavor :delete-child) (child)
  (cond
    ((member child children)
      (setq children (wegnehmen child children))
      t)
    (t
      (<- uinterf ':writeln
        (list "DELETE-CHILD:" "CHILD" child "DOES NOT EXIST IN" name))
      nil)))

(defmethod (data-file-flavor :children) ()
  (cond
    ((null children)
      (list name))
    (t
      (cons
        name
        (list
          (mapcar
            '(lambda (x) (<- (eval (<- database ':file-symbol x)) ':children))
            children))))))

; ---------------------------------------------------------------------------

(defflavor dir-file-flavor
  (file-type
   file-list)
  (file-flavor))

(defmethod (dir-file-flavor :file-type?) ()
  file-type)

(defmethod (dir-file-flavor :insert-file) (f-name)
  (setq file-list (cons f-name file-list)))

(defmethod (dir-file-flavor :remove-file) (f-name)
  (setq file-list (wegnehmen f-name file-list)))

(defmethod (dir-file-flavor :member-file) (f-name)
  (member f-name file-list))

(defmethod (dir-file-flavor :dir) ()
  (cond
    ((null file-list)
      (<- uinterf ':writeln (list "No files found.")))
    (t
      (mapc
       '(lambda (x y)
          (tab (* (mod y 4) (<< database 'tab-number)))
          (<- uinterf ':write (list x)))
        file-list
        (number-list (length file-list)))))
    (<- uinterf ':writeln '())
  t)
```

Listing of the file ihs.l

```
; The function (ihs) starts and initializes the entire system.

(defun ihs ()
    ;
    ; Various I-O functions are loaded into the system.
    ; (If the function ihs has not been run before.)
    ;
    (cond
      ((not (boundp 'loaded-ttio))
```

```
    (cfasl 'ttio.obj '_ttioinit    'ttioinit    "integer-function")
    (getaddress      '_inchar      'inchar      "integer-function")
    (getaddress      '_peekchar    'peekchar    "integer-function")
    (getaddress      '_ttreset     'ttreset     "integer-function")
    (cfasl 'ttio.o   '_outchar     'outchar     "integer-function")
    (getaddress      '_outstring   'outstring   "integer-function")
    (cfasl 'snd.o    '_initialize  'initialize  "integer-function")
    (getaddress      '_sendwait    'sendwait    "integer-function")
    (getaddress      '_sendnowait  'sendnowait  "integer-function")))

(resetio)

; Some local variables.

(setq float-format "%.2f")
(setq temporary-superscript ())
(setq all-rules ())
(setq loaded-ttio ())

; This is to get rid of some output from the expert system.

(init-junk-io)

(cond
  ((boundp 'created-files)
    (mapc '(lambda (x) (makunbound x)) created-files)))

(setq created-files '(file-/))
  ; file-/ is actually created in database :initialize ugly
  ;
  ; Define the database.
  ;

  ; Defining variables for the setup menu, is and ml,
  ; (= internal-states and menu-list).

(setq is '(
  (trace-on f)
  (fruit slime-mold)        ; Default for fruit is a kind of
  (prompt   "IHS> ")        ; slimy, cheese-like fungus.
  (pling f)
  (yaps-debug f)
  (trace-scripts-on f)
  (lisp-system-trace f)
  (expert t)
  (user-state e)))

(setq ml '(
  ("Tracing of the lisp system" lisp-system-trace t f)
  ("Internal tracing"           trace-on          t f)
  ("Tracing of rule system"     yaps-debug        t f)
  ("Tracing of scripts"         trace-scripts-on t f)
  ("Idpac fruit"                fruit             symbol)
  ("Prompt"                     prompt            symbol)
  ("Pling at prompt"            pling             t f)
  ("User is an expert"          expert            t f)
  ("User state"                 user-state        e b)))


; Define a database and initialize it.

(setq database
  (make-instance 'database-flavor
                 'scripts             scripts
                 'actual-superscripts nil
                 'restart-superscript nil
                 'superscript-count   1
                 'intrac-variables intrac-variables
                 'script-macros       script-macros
                 'internal-states     is
                 'menu-list           ml
                 'other-dict-words    other-dict-words
                 'allowed-commands    allowed-commands
                 'command-grammar
                   (append external-commands internal-commands)))
(<- database ':initialize)
  ;
```

```
    ; Create the initial superscript, which contains all scripts
    ; initially.
    ;
  (setq xss
    (make-instance 'superscr-flavor
                   'script-concat-name 'xs
                   'scripts (mapcar 'car (<< database 'scripts))))
    ;
    ; Create scripts.
    ;
  (mapc
    '(lambda (scr-desc)
       (cond
         ((not (is-this-a-script (cadddadr scr-desc)))
           (patom "Erroneous script: ")
;          (patom (cadddadr scr-desc))
           (terpr))
         (t
           (patom "Script ")
           (patom (car scr-desc))
           (patom " is OK.")
           (terpr)))
       (set
         (concat 'xs- (car scr-desc))
         (make-instance 'script-flavor
                        'full-name     (concat 'xs- (car scr-desc))
                        'script-name   (car scr-desc)
                        'command-sequence (list (cadddadr scr-desc))
                        'big-list         nil
                        'file-bindings    nil)))
     (<< database 'scripts))
    ;
    ; Create rule-bases.
    ; (suspendio) and (resumeio) is to get rid of a lot of
    ; unimportant output from the expert system.
    ;
  (suspendio)
  (mapc
    '(lambda (scr-desc)
       (let ((script (concat 'xs- (car scr-desc))) name vars)
         (mapc
           '(lambda (rule no)
              (<- (eval script) 'untrace)
;             (<- (eval script) 'trace)
              (<- (eval script) ':install-rule rule))
           (append global-rules (cdr (assoc (car scr-desc) rules)))
           (number-list (plus (length global-rules)
                              (apply 'max (mapcar 'length rules)))))))
     (<< database 'scripts))
  (resumeio)

  ;
  ; Initialize the instance variables actual-superscripts and
  ; restart-superscript.
  ;

  (setq actual-superscript-0 (<- xss ':copy-superscript 'actual-script-0))
  (>> database
    'actual-superscripts (list 'actual-superscript-0))
  (<- actual-superscript-0 ':initialize)

  (setq restart-superscript (<- xss ':copy-superscript 'restart-script))
  (>> database
    'restart-superscript  'restart-superscript)

    ;
    ; Define the parser.
    ;
  (setq parser
    (make-instance 'parser-flavor))
    ;
    ; Define the matcher.
    ;
  (setq matcher
    (make-instance 'matcher-flavor))
    ;
    ; Define the query-module.
```

```
    ;
    (setq query
      (make-instance 'query-flavor))
      ;
      ; Define the uinterface-module.
      ;
    (setq uinterf
      (make-instance 'uinterf-flavor))
      ;
      ; Define the iinterface-module (the interface to VMS and Idpac).
      ;
    (setq iinterf
      (make-instance 'iinterf-flavor))
      ;
      ; Create the dictionary and initialize it.
      ;
    (setq dictionary
      (make-instance 'dictionary-flavor))
    (<- dictionary ':initialize)
      ;
      ; Start running the system.
      ;
    (start-ihs)
    'Ready)
```

Listing of the file iinterf.l

```
; This module handles the communication with Idpac via
; C routines and VMS mailboxes. The C routines are
; sendwait and sendnowait.

(defflavor iinterf-flavor
  ())

(defmethod (iinterf-flavor :command-to-idpac) (command)
  (cond
    ((<- database ':get-trace)
      (<- uinterf ':patom (list "Iinterface: " command))
      (<- uinterf ':writeln `("Processor time: " ,(/ (car (ptime)) 60)))))
  (let
    ;
    ; Convert the list to a string.
    ;
    ((string (list-to-string command)))
      ;
      ; Send the string as a command to Idpac.
      ;
      (sendnowait string (difference (length (explode string)) 2))
      ;
      ; Send a dummy command and wait until it has been processed.
      ; Actually two dummy commands, because an empty line is interpreted
      ; as a comand in Idpac's editor.
      ;
      (sendnowait " " 1) ; Necessary in editing.
      (sendwait " " 1)
      ;
      ; These lines are used when idpac is not connected. Just prints
      ; the command on the screen.
      ;
;     (<- uinterf ':video-attribute 'reverse)
;     (<- uinterf ':writeln (list string))
;     (<- uinterf ':video-attribute 'off)
                                        ))
```

Listing of the file inc.l

```
(load 'inc1.l)
```

Listing of the file inc1.l

```
; Include file for the (ihs). Types a nice message and
; loads all files.

(defun arzenlisp ()
  (patom "Stand by for this...")
```

```
  (terpr)
  (patom (implode '(27)))
  (patom "[2J")                                          ; Erase screen
  (patom (implode '(27)))
  (patom "[9;1H")                                        ; Cursor to midscreen
  (patom (implode '(27)))
  (patom "#3               ArzenLisp               ")
  (terpr)
  (patom (implode '(27)))
  (patom "#4               ArzenLisp               ")
  (terpr)
  (patom (implode '(27)))
  (patom "[4B")
  (patom (implode '(27)))
  (patom "#6     Now available with source code!   ")
  (terpr)
  (patom (implode '(27)))
  (patom "[24;1H")
  (terpr))

(defun incload (file-list)
  (let ((nr 2))
    (mapc
      '(lambda (x)
         (load x)
         (patom (implode '(27)))
         (patom "[A")
         (patom (implode '(27)))
         (patom "[K")
         (patom "File number ")
         (patom nr)
         (patom ", ")
         (patom x)
         (patom " loaded.")
         (terpr)
         (setq nr (1+ nr)))
      file-list)))

(arzenlisp)
(patom "File number 1, inc1.l loaded.")
(terpr)
(incload '(
; perp.l
; perp1.l
  copy.l
  verifier.l
  util.l
  lowlevel.l
  visual.l
  vt100.l
  abbrev.l
  iinterf.l
  uinterf.l
  commands.l
  intracvar.l
  scripts.l
  scrmac.l
  rules.l
  dictwords.l
  script.l
  superscr.l
  file.l
  database.l
  parser.l
  matcher.l
  query.l
  dict.l
  ihs.l
       ))

;(ihs)
```

Listing of the file intracvar.l

```
; This file contains the definition of the original predeclared
; global Intrac variables.
```

```
(setq intrac-variables '(
  (nplx.   100)
  (pseps.  10.0E-7)
  (ymax.   0.0)
  (ifp.    1)
  (delta.  1.0)
  (reps.   10.0E-7)
  (aeps.   10.0E-19)
  (wmin.   10.0E-3)
  (ftest.  0)
  (ceps.   10.0E-5)
  (liml.   0)
  (itml.   20)
  (yc.     0.0)
  (ys.     0.0)
  (niter.  100)
  (print.  0)
  (amp.    1.0)
  (nu.     9)
  (ymin.   0.0)
  (scales. 1)
  (nof.    100)
  (wmax.   100.0)
  (tick.   1.0)
  (iniml.  0)
  (priml.  0)
  (xc.     0.0)
  (xs.     0.0)))
```

Listing of the file lowlevel.l

```
; This file contains the routines for basic IO and lexical analysis used
; in the uinterf-flavor, (and vt100-flavor). Here and there it is very
; Franz Lisp dependent.

(setq alphabet
          '(97 98 99 100 101 102 103 104 105 106 107 108 109 110
           111 112 113 114 115 116 117 118 119 120 121 122))

(setq ALPHABET
          '(65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
            85 86 87 88 89 90))

(setq QUICK-AND-DIRTY-LIST '(63))  ; to allow the command ??

(setq ALPHA (append ALPHABET alphabet QUICK-AND-DIRTY-LIST))

(setq NUMERALS '(48 49 50 51 52 53 54 55 56 57))

(setq delimiters '(32 9))

(setq LINE_DELIMITER '(10))

(setq END_OF_LINE_MARK '(*CR*))

(setq ALPHA_NUMERALS (append ALPHA NUMERALS))

(setq IDENTIFIER_CHARACTERS (append ALPHA_NUMERALS '(36 46 95)))

; Readloop reads one line from the keyboard and returns the lexical
; tokens as a Lisp list with atoms and numbers.

(def readloop
  (lambda ()
    (do ((result nil)) ((not (null result)) result)
      (do
        ((ch (tyipeek) (tyipeek)))
        ((equal ch (car LINE_DELIMITER))
          (tyi)
          (setq result (append result END_OF_LINE_MARK)))
        (cond
          ((memq ch ALPHA)
            (do
             ((loc '() (append1 loc (tyi)))
              (lch ch (tyipeek)))
             ((not (memq lch IDENTIFIER_CHARACTERS))
                 (setq result (append1 result (concatl (mapcar 'ascii loc)))))
```

```
          (comment EMPTY DO BODY)))
        ((or
           (memq ch NUMERALS)
           (memq ch '(43 45)))                ; + -
          (let
            ((tsign (cond ((= ch 43) (tyi) 1) ((= ch 45) (tyi) -1) (t 1)))
              (tnum))
            (cond
              ((memq (tyipeek) NUMERALS)
                (setq tnum (get-num))
                (cond
                  ((= (tyipeek) 46)             ; .
                    (tyi)
                    (setq tsign (times 1.0 tsign))
                    (cond
                      ((memq (tyipeek) NUMERALS)
                        (setq tnum (plus tnum (get-fract)))))))
                (cond
                  ((memq (tyipeek) '(69 101))    ; e E
                    (let ((x (tyi)))
                      (cond
                        ((memq (tyipeek) NUMERALS)
                          (setq result
                            (append1 result
                              (times tsign tnum (expt 10.0 (get-num))))))
                        ((memq (tyipeek) '(43 45)) ; + -
                          (let*
                            ((y (tyi))
                              (ttsign (cond ((= y 43) 1) (t -1))))
                            (cond
                              ((memq (tyipeek) NUMERALS)
                                (setq result
                                  (append1 result
                                    (times tnum tsign
                                      (expt 10.0 (times ttsign (get-num)))))))
                              (t
                                (setq result
                                  (append
                                    result
                                    (list
                                      (times tsign tnum)
                                      (ascii x) (ascii y))))))))
                        (t
                          (untyi x)
                          (setq result
                            (append result
                              (list (times tnum tsign) ))))))
                    (t
                      (setq result (append1 result (times tsign tnum))))))
                (t
                  (setq result (append1 result (ascii ch)))))))
          (t
            (let
              ((x (tyi)))
              (cond
                ((memq x delimiters))
                (t (setq result (append1 result (ascii x)))))))
          (comment last of great cond))))))

; Help routines for readloop.

(def get-num
  (lambda ()
    (do
      ((loc 0 (plus (times 10 loc) (difference (tyi) 48)))
        (lch (tyipeek) (tyipeek)))
      ((not (memq lch NUMERALS)) loc)
      (comment EMPTY DO BODY))))

(def get-fract
  (lambda ()
    (do
      ((scale 10.0 (times scale 10.0))
        (loc_1 0
          (plus loc_1
            (quotient (float (difference (tyi) 48)) scale)))
        (lch_1 (tyipeek) (tyipeek)))
```

```
      ((not (memq lch_1 NUMERALS)) loc_1)
      (comment EMPTY DO BODY))))

; Transforms an atom-list to a Lisp list with atoms and lists.
; Everything between |(| and |)| will be put on a list.

(def lispify
  (lambda (z)
    (do
      ((x z (cdr x))
        (stack ())
        (alist ()))
      ((null x)
        (cond
          ((and (null stack) (null x))
            alist)
          (t
            (list '*WARNING_UNMATCHING_PARENTHESES* z))))
      (cond
        ((equal (car x) '|(|)
          (setq stack (cons alist stack))
          (setq alist ()))
        ((equal (car x) '|)|)
          (setq alist (append1 (car stack) alist))
          (setq stack (cdr stack)))
        (t
          (setq alist (append1 alist (car x))))))))

; ------------------------------------------------------------------------------

(defun printitem (item)
  (cond
    ((numberp item)
      (outstring
        (substring (implode (explode item)) 1)
        (length (explode item))))
    ((stringp item)
      (outstring
        item
        (difference (length (explode item)) 2)))
    (t
      (outstring
        (substring item 1)
        (length (explode item))))))

(defun insymbol (x)
  (cond
    ((= x 127) 'DEL)
    ((= x 13)  'CR)
    ((= x 27) ;escape
      (let ((ch1 (peekchar)))
        (cond
          ((memq ch1 '(91)) ; [
            (let ((ch2 (peekchar)))
              (cond
                ((memq ch2 '(65)) (inchar) (inchar) 'UP)      ; A
                ((memq ch2 '(66)) (inchar) (inchar) 'DOW)     ; B
                ((memq ch2 '(67)) (inchar) (inchar) 'RIG)     ; C
                ((memq ch2 '(68)) (inchar) (inchar) 'LEF)))) ; D
          ((memq ch1 '(27))  ; escape escape = ^[
            (inchar)
            'ESC)
          (t
            (insymbol (inchar))))))
    ((< x 32)   ; Control-chars
      (concat (ascii 94) (ascii (+ x 64))))
    (t
      (ascii x))))

(defun insymb ()
  (insymbol (inchar)))
```

Listing of the file matcher.l

```
; This file contains the matcher-flavor, which makes the matcher.
; The matcher orchestrates, the actual matching of scripts,
; which is done in the scripts via the database. The matcher handles
```

```
; the matching strategy.

(defflavor matcher-flavor
  ())

; ------------------------------------------------------------------------

; The method for the first matching BEFORE the query module.

(defmethod (matcher-flavor :match-command) (command)
  (cond
    ((<- database ':get-trace)
      (<- uinterf ':patom (list "Matcher: " command))
      (<- uinterf ':writeln `("Processor time: " ,(/ (car (ptime)) 60)))))
  (let (x)
    (setq x (<- database ':match-actual-superscripts command))
    (cond
      ((null x) (setq x (<- database ':match-restart-superscript command))))
    (cond
      ((null x)
        (cond
          ((null (member (car command) (<< database 'allowed-commands)))
            (<- uinterf ':writeln (list "No match"))))
        (setq x (<- self ':build-no-match-command command))
        (setq x (<- self ':build-multi-command x))
        (<- query ':query-command x t)) ; no-match => t
      (t
        (setq x (<- self ':build-multi-command x))
        (<- query ':query-command x nil)))))

; The method for the second matching after the query module, used when all
; parameters in a command are known.

(defmethod (matcher-flavor :match-command-finally) (command)
  (cond
    ((<- database ':get-trace)
      (<- uinterf ':patom (list "Matcher finally: " command))))
  (let (x)
    (setq x (<- database ':match-actual-superscripts-finally command))
    (cond
      ((null x)
        (setq x (<- database ':match-restart-superscript-finally command))))
    (cond
      ((null x)
        (cond
          ((null (member (car command) (<< database 'allowed-commands)))
            ;  (<- uinterf ':writeln (list "No match"))
                                                    ))))))

; Internal methods ------------------------------------------------------

; The methods :build-multi-command, :put-together and, :move-in
; are used in building one command from several successfully matched commands
; which may come from different scripts.

(defmethod (matcher-flavor :build-multi-command) (multi-list)
  (<- self ':put-together
    (mapcar
      '(lambda (x)
         (<- self ':move-in
           (car x)
           (<- self ':put-together (cdr x))))
      multi-list)))

(defmethod (matcher-flavor :put-together) (command-list)
  (do
    ((c-l command-list (cdr c-l))
     (res nil))
    ((null c-l) res)
    (do
      ((h1 (car c-l) (cdr h1))
       (tmp res (cdr tmp))
       (resx nil))
      ((null h1) (setq res resx))
      (cond
        ((and
          (listp (car h1))
```

```
          (member (caar h1) '(infile outfile globfile parameter)))
            (setq
              resx
              (append1 resx (cons (caar h1) (cons (cadar h1) (cdar tmp))))))
         (t
           (setq resx (append1 resx (car h1))))))))

(defmethod (matcher-flavor :move-in) (scr-name command)
  (mapcar
    '(lambda (y)
       (cond
         ((and (listp y)
           (member (car y) '(infile outfile globfile parameter)))
           (list (car y) (cons scr-name (cdr y))))
         (t y)))
    command))

; When a command does not match any script.

(defmethod (matcher-flavor :build-no-match-command) (command)
  (list
    (list
     'no-script
      (mapcar
        '(lambda (clause)
           (cond
             ((atom clause) clause)
             ((member (car clause) '(infile outfile globfile parameter))
               (list
                 (car clause)
                 (cons 'no-name (cdr clause))))
             (t clause)))
        command))))
```

Listing of the file parser.l

```
; This module contains the code for the parser. Parse-command
; is called from the readloop. The command is parsed and
; replaced by the specification found in the command grammar.
; Then the matcher is called through :match-command.

(defflavor parser-flavor
  ())

(defmethod (parser-flavor :parse-command) (command)
  (cond
    ((<- database ':get-trace)
      (<- uinterf ':patom (list "Parser: " command))
      (<- uinterf ':writeln '("Processor time: " ,(/ (car (ptime)) 60)))))
  (let* ((cs (<- database ':get-command-specification (car command)))
         (com
           (cond
             ((and (listp cs) (equal (car cs) 'let))
               (append
                 (list 'let)
                 (cond ((cadr command) (list (cadr command))))
                 (<- self ':subst-intrac-variables (cddr command))))
             (t
               (cons
                 (car command)
                 (<- self ':subst-intrac-variables (cdr command)))))))
    (cond
      ((null command)                          ; if empty command then
        (<- iinterf ':command-to-idpac '(| |))) ; bypass the system
      ((atom cs)
        (<- uinterf ':writeln (list cs command)))
      (t
        (let ((tmp (<- self ':parse cs (cons (car cs) (cdr com)))))
          (cond
            ((equal (last tmp) '($error))
              (<- uinterf ':patom (list "Syntax error:" tmp)))
            ((member (car tmp) (<< database 'internal-command-names))
              (setq tmp (<- query ':ask-parameters tmp))
              (<- database ':internal-command tmp))
            (t
              (<- matcher ':match-command tmp)))))))))
```

```
; Internal methods ------------------------------------------------------------

; Intrac variables are substituted for their values before any
; matching takes place. This works in most cases.

(defmethod (parser-flavor :subst-intrac-variables) (command)
  (cond
    ((and
      (atom command)
      (assoc command (<< database 'intrac-variables)))
      (cadr (assoc command (<< database 'intrac-variables))))
    ((atom command)
      command)
    (t
      (mapcar
        '(lambda (x) (<- self ':subst-intrac-variables x))
        command))))

; Parse is the pattern matcher that performs the parsing.

(defmethod (parser-flavor :parse) (spec command)
  (cond
    ((and (null spec) (null command)) nil)
    ((null command)
      (<- self ':fill-in spec))
    ((<- self ':atomequal? spec command)
      (<- self ':atomcons spec command))
    ((<- self ':infile-spec? spec)
      (<- self ':infilecons spec command))
    ((<- self ':outfile-spec? spec)
      (<- self ':outfilecons spec command))
    ((<- self ':number-spec? spec)
      (<- self ':numbercons spec command))
    ((<- self ':numlist-spec? spec)
      (<- self ':numlistcons spec command))
    ((<- self ':numlist1-spec? spec)
      (<- self ':numlist1cons spec command))
    ((<- self ':symbol-spec? spec)
      (<- self ':symbolcons spec command))
    ((<- self ':symlist-spec? spec)
      (<- self ':symlistcons spec command))
    ((<- self ':symlist1-spec? spec)
      (<- self ':symlist1cons spec command))
    ((and
       spec
       (atom (car spec)))
      (cons (car spec) (<- self ':parse (cdr spec) command)))
    (t
      (<- self ':errorcons spec command))))

(defmethod (parser-flavor :fill-in) (spec)
  (cond
    ((null spec) '())
    ((atom (car spec))
      (cons (car spec) (<- self ':fill-in (cdr spec))))
    ((<- self ':infile-spec? spec)
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))
    ((<- self ':outfile-spec? spec)
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))
    ((<- self ':number-spec? spec)
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))
    ((<- self ':numlist-spec? spec)
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))
    ((<- self ':numlist1-spec? spec)
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))
    ((<- self ':symbol-spec? spec)
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))
    ((<- self ':symlist-spec? spec)
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))
    ((<- self ':symlist1-spec? spec)
```

```
      (cons (<- self ':default spec)
        (<- self ':fill-in (cdr spec))))))

(defmethod (parser-flavor :ready?) (spec command)
  (and (null spec) (null command)))

(defmethod (parser-flavor :atomequal?) (spec command)
  (and
    (atom (car spec))
    (atom (car command))
    (equal (car spec) (car command))))

; Some useful selectors. Used by the selectors in the next group.

(defmethod (parser-flavor :infile-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'infile)))

(defmethod (parser-flavor :outfile-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'outfile)))

(defmethod (parser-flavor :number-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'number)))

(defmethod (parser-flavor :numlist-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'numlist)))

(defmethod (parser-flavor :numlist1-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'numlist1)))

(defmethod (parser-flavor :symbol-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'symbol)))

(defmethod (parser-flavor :symlist-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'symlist)))

(defmethod (parser-flavor :symlist1-spec?) (spec)
  (and
    (listp (car spec))
    (equal (caar spec) 'symlist1)))

; More useful selectors. Use the selectors in the previuos group.
; Actually, these selectors operate on the complete command.

(defmethod (parser-flavor :file?) (command)
  (and
    (atom (car command))
    (not (numberp (car command)))
    (not (eq (car command) '<))))

(defmethod (parser-flavor :number?) (command)
  (numberp (car command)))

(defmethod (parser-flavor :numlist?) (command)
  (and
    (listp (car command))
    (>= (length (car command)) 1)
    (numberp (caar command))))

(defmethod (parser-flavor :numlist1?) (command)
  (and
    (listp (car command))
    (equal (length (car command)) 1)
    (numberp (caar command))))
```

```
(defmethod (parser-flavor :symbol?) (command)
  (and
    (atom (car command))
    (not (numberp (car command)))
    (not (equal (car command) '<))))

(defmethod (parser-flavor :symlist?) (command)
  (and
    (listp (car command))
    (> (length (car command)) 1)
    (<- self ':symbol? (car command))))

(defmethod (parser-flavor :symlist1?) (command)
  (and
    (listp (car command))
    (equal (length (car command)) 1)
    (<- self ':symbol? (car command))))

(defmethod (parser-flavor :default) (spec)
  (cond
    ((and
       (listp (car spec))
;      (cdar spec)             ; A little change by Perp
       (listp (cadar spec))
       (equal (caadar spec) 'default))
      (list (caar spec) (cadadar spec) 'idpac-default))
    (t (list (caar spec) '*unknown*))))

(defmethod (parser-flavor :atomcons) (spec command)
  (cons
    (car command)
    (<- self ':parse (cdr spec) (cdr command))))

(defmethod (parser-flavor :infilecons) (spec command)
  (cond
    ((<- self ':file? command)
      (cons
        (list 'infile (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))

(defmethod (parser-flavor :outfilecons) (spec command)
  (cond
    ((<- self ':file? command)
      (cons
        (list 'outfile (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))

(defmethod (parser-flavor :numbercons) (spec command)
  (cond
    ((<- self ':number? command)
      (cons
        (list 'number (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))

(defmethod (parser-flavor :numlistcons) (spec command)
  (cond
    ((<- self ':numlist? command)
      (cons
        (list 'numlist (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))
```

```
(defmethod (parser-flavor :numlist1cons) (spec command)
  (cond
    ((<- self ':numlist1? command)
      (cons
        (list 'numlist1 (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))

(defmethod (parser-flavor :symbolcons) (spec command)
  (cond
    ((<- self ':symbol? command)
      (cons
        (list 'symbol (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))

(defmethod (parser-flavor :symlistcons) (spec command)
  (cond
    ((<- self ':symlist? command)
      (cons
        (list 'symlist (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))

(defmethod (parser-flavor :symlist1cons) (spec command)
  (cond
    ((<- self ':symlist1? command)
      (cons
        (list 'symlist1 (car command))
        (<- self ':parse (cdr spec) (cdr command))))
    (t
      (cons
        (<- self ':default spec)
        (<- self ':parse (cdr spec) command)))))

(defmethod (parser-flavor :errorcons) (spec command)
  (list '$error))
```

Listing of the file query.l

```
; This is the Query module. It receives a command description
; from the matcher, fills in missing parameters by asking the
; user, send information to the file system and the YAPS
; databases, and sends a stripped down version od the command
; to the Idpac interface, for mailboxing to Idpac.

; Grammar for a file descriptor :
;
; (FILE-TYPE
;   (SCRIPT-NAME
;     (INTERNAL-NAME EXTERNAL-NAME [default])
;     [(INTERNAL-NAME EXTERNAL-NAME [default]) ...])
;   [(SCRIPT-NAME
;     (INTERNAL-NAME EXTERNAL-NAME [default])
;     [(INTERNAL-NAME EXTERNAL-NAME [default]) ...]) ...])
;
; FILE-TYPE     ::= infile | outfile | globfile
; SCRIPT-NAME   ::= <identifier>
; INTERNAL-NAME ::= <identifier>
; EXTERNAL-NAME ::= <identifier>

(defflavor query-flavor
  (no-file-match
   d-pattern
   (numlist '(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)))
  ())
```

```
; External methods ------------------------------------------------------------

; This is the method that is called from the matcher.

(defmethod (query-flavor :query-command) (command no-match)
  (cond
    ((<- database ':get-trace)
      (<- uinterf ':patom (list "Query: " command))
      (<- uinterf ':writeln '("Processor time: " ,(/ (car (ptime)) 60)))))
  (let (x)
    (setq x (<- self ':fix-or-ask-filenames command))
    (setq x (<- self ':ask-unknown-filenames x))
    (setq x (<- self ':ask-parameters x))
    (<- matcher
      ':match-command-finally (<- self ':rebuild-for-final-matching x))
    (cond
      ((<- self ':all-infiles-exists? x (car x) 0)
        (<- self ':update-filebindings x)
        (<- database ':make-assigns)
        (<- self ':update-filetree x)
        (<- database ':special-command x)
        (<- database ':purge-superscripts)
        (setq x (<- self ':rebuild-idpac-command x))
        (setq x (mapzap '(lambda (y) y) x))      ; to remove nil's in plot ...
        (<- database ':update-command-history x)
        (<- iinterf ':command-to-idpac x)
        (<- database ':perform-kscalls)))
    (cond
      ((<- database ':get-trace-scripts)
        (<- database ':dump-scripts)))))

; Internal methods ------------------------------------------------------------

; Masked list takes a mask, e.g. (t nil t t nil) and a list of filenames,
; e.g. (f1 f2 f3 f4 f5), and returns a new list of filenames with some of
; the filenames masked off, e.g. (f1 f3 f4).

(defmethod (query-flavor :masked-list) (list mask)
  (cond
    ((null list)
      nil)
    ((car mask)
      (cons (car list) (<- self ':masked-list (cdr list) (cdr mask))))
    (t
      (<- self ':masked-list (cdr list) (cdr mask)))))

; All-equal returns true if all atoms in the argument list are alike.

(defmethod (query-flavor :all-equal?) (atom-list)
  (cond
    ((equal (length atom-list) 1) t)
    (t
      (apply 'and
      (mapcar '(lambda (x) (equal (car atom-list) x)) (cdr atom-list))))))

; All-external-names takes a file descriptor and returns a list of all
; the external filenames in it.

(defmethod (query-flavor :all-external-names) (descr)
  (apply 'append
    (mapcar
      '(lambda (x) (mapcar 'cadr (cdr x)))
       (cdr descr))))

; Filenames-equal? takes a file descriptor and returns true if all the
; external names in it are alike.

(defmethod (query-flavor :filenames-equal?) (file-descriptor)
  (<- self ':all-equal? (<- self ':all-external-names file-descriptor)))

; Mask-file-descriptor takes a file descriptor and a mask pattern and
; applies the mask pattern to each script-part of the descriptor,
; removing obsolete external names. When ready, it returns an updated
; version of the file descriptor.

(defmethod (query-flavor :mask-file-descriptor) (descr pattern)
  (cond
```

```
    ((null descr) nil)
    ((member (car descr) '(infile outfile globfile parameter))
      (cons
        (car descr)
        (<- self ':mask-file-descriptor (cdr descr) (cdr pattern))))
    (t
      (let ((x (<- self ':masked-list (car descr) (car pattern))))
        (cond
          ((equal (length x) 1)
            (<- self ':mask-file-descriptor (cdr descr) (cdr pattern)))
          (t
            (cons
              x
              (<- self ':mask-file-descriptor (cdr descr) (cdr pattern)))))))))

; Find-pattern takes a command description and asks what external
; names should be used for the infiles in the command. In the case
; that all names are alike, there is, of course, no need to ask.
; It sets 'pattern' to the appropriate mask pattern.

(defmethod (query-flavor :find-pattern) (command)
  (cond
    ((null command) nil)
    ((atom (car command))
      (cons (car command) (<- self ':find-pattern (cdr command))))
    ((equal (caar command) 'infile)
      (cond
        ((<- self ':filenames-equal? (car command))
          (cons (car command) (<- self ':find-pattern (cdr command))))
        (t
           (setq d-pattern (<- self ':ask (car command)))
           (cons d-pattern (cdr command)))))
    (t
      (cons (car command) (<- self ':find-pattern (cdr command))))))

; Ask takes a file descriptor, presents the different alternatives for
; the external file name and asks the user for which one it should be.
; If the user conjures up a new one, an emergency exit is taken and
; 'no-file-match' is set to true.

(defmethod (query-flavor :ask) (pattern)
  (let*
    ((all (remove-multiple (<- self ':all-external-names pattern)))
     (y (<- uinterf ':prompt (append1 all "Which file") 'file)))
      (cond
        ((member y all)
          (cons
            (car pattern)                                  ; infile - outfile
            (mapcar
              '(lambda (x)
                 (cons (car x)                             ; script-name
                   (mapcar
                     '(lambda (xx)
                        (cond
                          ((equal (cadr xx) y)
                            (list (car xx) (cadr xx)))
                          (t nil)))
                      (cdr x))))
              (cdr pattern))))
        (t
          (setq no-file-match t)
          `(infile (no-script (no-name ,y)))))))

; Update-command takes a command and a mask pattern, removes obsolete
; external filenames from the infile and globfile descriptors, and
; returns the updated command.

(defmethod (query-flavor :update-command) (command pattern)
  (mapcar
    '(lambda (x)
       (cond ((atom x) x)
             ((member (car x) '(infile outfile globfile parameter))
                              ; To get a uniform treatment of
                              ; all interactions with scripts.
              (<- self ':mask-file-descriptor x pattern))
             (t x)))
  command))
```

```
; Ask-files-in-command takes a command, ask for the names of the
; files that are currently defaulted from non-matching scripts,
; and returns the updated command.

(defmethod (query-flavor :ask-files-in-command) (command)
  (mapcar
    '(lambda (x y)
       (cond ((atom x) x)
             ((member (car x) '(infile outfile globfile parameter))
                                ; Only infiles can be defaulted,
                                ; so this clause often ends up in (t...).
                                ; The purpose is to get (no-script ...).
              (cond
                ((equal (caddadadr x) 'default)
                 `(,(car x)
                    (no-script
                     (no-name ,(<- uinterf ':prompt-command command y nil)))))
                (t
                 `(,(car x) (no-script (no-name ,(cadadadr x)))))))
             (t x)))
  command
  numlist))

; Fix-or-ask-filenames takes a command, and iterates through the infile
; descriptors. For each unresolved name choice it asks the user and then
; goes through the command, removing several name alternatives. When
; ready, it returns the updated command.

(defmethod (query-flavor :fix-or-ask-filenames) (command)
  (setq no-file-match nil)
  (prog (new-command del-pat)
    (setq new-command command)
   label
    (setq d-pattern nil)
    (setq new-command (<- self ':find-pattern new-command))
    (cond
      ((null d-pattern)
        (return new-command)))
    (setq del-pat d-pattern)
    (cond
      (no-file-match (return (<- self ':ask-files-in-command new-command)))
      (t
        (setq new-command (<- self ':update-command new-command del-pat))))
    (go label)))

; Ask-unknown-filenames takes a command and asks the user for the external
; names currently represented with a '*', i.e. filenames that were not
; given in the command and could not be inferred from scripts. When ready
; it returns the updated command.

(defmethod (query-flavor :ask-unknown-filenames) (command)
  (do
    ((first-part nil) (last-part command (cdr last-part)))
    ((null last-part) first-part)
    (cond
      ((atom (car last-part))
        (setq first-part (append1 first-part (car last-part))))
      ((and (member (caar last-part) '(infile outfile globfile parameter))
            (equal
              (car (<- self ':all-external-names (car last-part)))
              '*unknown*))
        (setq first-part
          (append1 first-part
            (lsubst
              (list
                (<- uinterf ':prompt-command (append first-part last-part)
                  (length first-part) nil))
              '*unknown*
              (car last-part)))))
        (t
          (setq first-part (append1 first-part (car last-part)))))))

; Ask-parameters takes a command, asks for values of parameters
; currently given as '*' and returns the updated command.

(defmethod (query-flavor :ask-parameters) (command)
```

```
  (do
    ((first-part nil)
     (last-part command (cdr last-part)))
    ((null last-part) first-part)
    (cond
      ((atom (car last-part))
        (setq first-part (append1 first-part (car last-part))))
      ((equal (cadar last-part) '*unknown*)
        (setq first-part
         (append1
           first-part
           (list
             (caar last-part)
             (<- uinterf ':prompt-command (append first-part last-part)
               (length first-part) nil)))))
        (t
          (setq first-part (append1 first-part (car last-part)))))))

; Ask-defaulted-parameters takes a command and asks for the values of
; parameters that Idpac otherwise can default. Its questions gives the
; Idpac default, e.g. "Sampling interval? (Default delta.) > ". If the
; user hits CR only, that default is used. When ready, it returns the
; updated command. Ask-defaulted-parameters is only supposed to be used
; in an extremely helpful beginner mode. Currently it is not called
; from Query-command.

(defmethod (query-flavor :ask-defaulted-parameters) (command)
  (let (temp)
    (mapcar
     '(lambda (x y)
        (cond
          ((atom x) x)
          ((atom (cadr x)) x)
          ((equal (last x) '(idpac-default))
            (cond
              ((cadr x)
                (setq temp (<- uinterf ':prompt-command command y (cadr x))))
              (t
                (setq temp (<- uinterf ':prompt-command command y 'Nothing))))
            (cond
              ((equal temp "") x)
              (t
                (list (car x) temp))))
          (t
            x)))
      command
      numlist)))

; ------------------------------------------------------------------------------

(defmethod (query-flavor :all-infiles-exists?) (com com-name nr)
  (cond
    ((null com)
      t)
    ((atom (car com))
      (<- self ':all-infiles-exists? (cdr com) com-name (add1 nr)))
    ((equal (caar com) 'infile)
      (cond
        ((<- database ':file-exists?
          (concat
            (cadadadar com)
            (cond
              ((cadr (<- database ':get-clause-nr com-name nr)))
              (t '|.d|))))
          (<- self ':all-infiles-exists? (cdr com) com-name (add1 nr)))
        (t
          (<- uinterf ':writeln (list "No such file: " (cadadadar com)))
          (<- self ':all-infiles-exists? (cdr com) com-name (add1 nr))
          nil)))
    (t
      (<- self ':all-infiles-exists? (cdr com) com-name (add1 nr)))))

(defmethod (query-flavor :update-filebindings) (command)
  (mapc
    '(lambda (clause)
       (cond
         ((and
```

```
          (not (atom clause))
          (member (car clause) '(outfile globfile parameter)))
          (mapc
             '(lambda (script-desc)
                (cond
                  ((not (equal (car script-desc) 'no-script))
                    (mapc
                      '(lambda (int-ext-pair)
                        (<-
                          (<- database ':get-script (car script-desc))
                          ':new-external-name
                          (car int-ext-pair)
                          (cadr int-ext-pair)))
                      (cdr script-desc)))))
             (cdr clause)))))
    command))

(defmethod (query-flavor :get-outfile-name) (command)
  (cond
    ((null command) nil)
    ((and
      (not (atom (car command)))
      (equal (caar command) 'outfile))
      (cadadadar command))
    (t
      (<- self ':get-outfile-name (cdr command)))))

(defmethod (query-flavor :get-list-of-filenames) (type command)
  (apply 'append
    (mapcar
      '(lambda (x y)
        (cond
          ((and
            (not (atom x))
            (equal (car x) type))
            (list
              (concat
                (cadadadr x)
                (cond
                  ((cadr (<- database ':get-clause-nr (car command) y)))
                  (t '|.d|)))))
          (t nil)))
    command
    numlist)))

; Update-filetree takes a command, picks out the created files,
; (the outfiles and  globfiles), and the parent files, (the infiles),
; and calls the file system to update the file tree.
;
; "Yx, Byx! Kax, Brax!" (An old, Swedish proverb.) Certain commands
; should have more complicated effects.

(defmethod (query-flavor :update-filetree) (command)
  (let
    ((infiles (<- self ':get-list-of-filenames 'infile command))
     (outfiles (<- self ':get-list-of-filenames 'outfile command))
     (globfiles (<- self ':get-list-of-filenames 'globfile command)))
    (mapc
      '(lambda (gf)
         (cond
           ((null (<- database ':file-exists? gf))
             (<- database ':new-file-parents-and-kids
             gf
             '()
             (car command)))))
      globfiles)
    (mapc
      '(lambda (of)
         (<- database ':new-file-parents-and-kids
           of
           (append globfiles infiles)
           (car command)))
      outfiles)))

(defmethod (query-flavor :rebuild-idpac-command) (command)
  (apply 'append
    (mapcar
```

```
      '(lambda (x)
        (cond
          ((atom x) (list x))
          ((memq (car x) '(infile outfile globfile parameter))
            (list (cadadadr x)))
          ((equal (last x) '(idpac-default))
            '())
          (t
            (list (cadr x)))))
       command)))

(defmethod (query-flavor :rebuild-for-final-matching) (command)
  (let ((tmp) (spec))
    (setq tmp (<- self ':rebuild-idpac-command command))
    (setq spec (<- database ':get-command-specification (car tmp)))
    (<- parser ':parse spec tmp)))
```

Listing of the file rules.l

```
; This is the rule databases of the system.

(setq global-rules '(

  ((note double plot -y -z)
   -->
   (fact message ("Plot the files" -y "and" -z "with one PLOT command."
                  "GIVE THE COMMAND PLOT " -y "/" -z
                  ". There is not full support for the command PLOT,"
                  "so you will not be queried for all parameters.")))

  ((check other half -h1 -h2 -oh)
    test
    -oh
    (not (or (equal -h1 -oh) (equal -h2 -oh)))
    -->
   (fact message ("You should have plotted the two halves, wise guy!")))

  ((prepare-document)
    -->
    (fact document-text ()))

  ((document -x)
   (document-text -y)
    -->
    (setq -y (append1 -y -x))
    (fact document-text -y))

  ((write-document -filename)
   (document-text -text)
    -->
    (let
      ((tmp-prt (fileopen -filename "w")))
      (mapc
        '(lambda (x)
           (princ (list-to-string x) tmp-prt)
           (terpr tmp-prt))
        -text)
      (close tmp-prt)))

  ((patom -x)
   -->
   (patom -x) (terpr))

  ((message -x)
   -->
   (remove 1)
   (<- self ':save-yaps-info -x))

  ((note start plmag)
   -->
   (fact message ("If you want a strange block length, use BLOCK."
                  "In any case you must start with PLBEG.")))

  ((note kill plmag -file)
   -->
   (fact message ("You left the command PLMAG working on the file"
                  -file "with kill. You have not altered the file.")))
```

```
))

(setq rules '(

(ml

; ---ASK-------------------------------------------------------------------------

 ((question enable)
  (question a priori knowledge)
  -->
  (cond
    ((equal
      (<- uinterf ':prompt
        '("Do you have any a priori knowledge of the system") 'symbol) 'y)
     (fact a priori knowledge exists))))

  ((question enable)
   (a priori knowledge exists)
   -->
   (let ((lo) (hi))
     (setq lo
       (<- uinterf ':prompt
         '("What is your guess about the lower limit of model order") 'number))
     (setq hi
       (<- uinterf ':prompt
         '("What is your guess about the upper limit of model order") 'number))
     (fact ^(append1 '(guessed lower model order limit) lo))
     (fact ^(append1 '(guessed upper model order limit) hi))))

  ((question enable)
   (a priori knowledge exists)
   -->
   (cond
     ((equal
       (<- uinterf ':prompt
         '("Is the system identified in a feedback loop?") 'symbol) 'y)
       (fact identified in feedback))))

  ((question enable)
   (a priori knowledge exists)
   -->
   (cond
     ((equal
       (<- uinterf ':prompt
         '("Can the system be viewed as a linear system?") 'symbol) 'y)
       (fact system may be linear))))

  ((question enable)
   (a priori knowledge exists)
   -->
   (cond
     ((equal
       (<- uinterf ':prompt
         '("Is the purpose of the identification controller design")
         'symbol)
       'y)
       (fact identification for controller design))))

  ((question enable)
   (question results of coherence test -f1 -f2)
   -->
   (let ((x) (y))
     (setq x
       (<- uinterf ':prompt
           '("Lower limit of interval where the coherence is"
             "greater than ~0.7") 'number))
     (setq y
       (<- uinterf ':prompt
           '("Upper limit of interval where the coherence is"
             "greater than ~0.7") 'number))
     (fact fact coherence-limits -f1 -f2 ^x ^y)))

  ((question enable)
   (question results aspec insignal -f1)
   -->
   (let ((x) (y))
```

```
    (setq x
      (<- uinterf ':prompt
          '("Lower limit of interval where the spectral density is"
            "high") 'number))
    (setq y
      (<- uinterf ':prompt
          '("Upper limit of interval where the spectral density is"
            "high") 'number))
    (fact fact aspec-limits -f1 ^x ^y)))

((question enable)
 (question number of steps to slide -f1 -f2)
 -->
 (let ((steps))
   (setq steps
     (<- uinterf ':prompt
         '("What is the x-coordinate for the zero-crossing")
         'number))
   (fact fact slided -f1 -f2 ^steps)))

((~ (question enable))
 (question ml identification result -file)
 -->
 (let ((filename (concat -file ".t")) (lossfunc) (aic))
   (cond
     ((probef filename)
       (setq file (infile filename))
       (do
         ((expr (ratom file) (ratom file)))
         ((equal expr 'AIC)
           (setq aic (ratom file))
           (close file))
         (cond
           ((and
              (equal expr 'LOSS)
              (equal (ratom file) 'FUNCTION))
              (setq lossfunc (ratom file))))
         (do
           ()
           ((and (> (tyipeek file) 64) (< (tyipeek file) 91)))
           (tyi file)))
       (fact fact mlid-data -file ^aic ^lossfunc)))))

((question enable)
 (question ml identification result -file)
 -->
 (let ((filename (concat -file ".t")) (lossfunc) (aic))
   (cond
     ((probef filename)
       (setq file (infile filename))
       (do
         ((expr (ratom file) (ratom file)))
         ((equal expr 'AIC)
           (setq aic (ratom file))
           (close file))
         (cond
           ((and
              (equal expr 'LOSS)
              (equal (ratom file) 'FUNCTION))
              (setq lossfunc (ratom file))))
         (do
           ()
           ((and (> (tyipeek file) 64) (< (tyipeek file) 91)))
           (tyi file))))
     (t
       (setq aic (<- uinterf ':prompt '("What is the AIC") 'number))
       (setq lossfunc nil)))
  (fact fact mlid-data -file ^aic ^lossfunc)))

((question enable)
 (question loss function -syst)
 -->
 (let ((loss-func))
   (setq loss-func
     (<- uinterf ':prompt
        '("What is the value of the loss function") 'number))
   (fact fact xval-1-loss-func -syst ^loss-func)))
```

```
((question enable)
 (question length of -fil)
 -->
 (let ((x))
   (setq x
     (<- uinterf ':prompt
       '("What is the the length of the file" ,-fil) 'number))
   (fact fact length -fil ^x)))

((question enable)
 (question results of ccoff -fil)
 -->
 (let ((direct-term) (feed-back))
   (setq direct-term
     (cond
      ((equal (<- uinterf ':prompt
        '("Is there any correlation in zero") 'symbol) 'y)
        t)
       (t 'f)))
   (setq feed-back
     (cond
      ((equal (<- uinterf ':prompt
        '("Is there any correlation for negative values") 'symbol) 'y)
        t)
       (t 'f)))
   (fact fact feed-back -fil ^feed-back)
   (fact fact direct-term -fil ^direct-term)))

((question enable)
 (question loss for -resf -type file)
 -->
 (let ((loss))
   (setq loss
      (<- uinterf ':prompt '("What is the value of"
                             "the loss function for" ,-resf) 'number))
   (fact fact -type has loss -resf ^loss)))

((question enable)
 (question fixed aic -syst)
 -->
 (let ((aic))
   (setq aic
     (<- uinterf ':prompt
       '("What is the value of the AIC") 'number))
   (fact fact fixed-aic -syst ^aic)))

; ------------------------------------------------------------------------

((fact length -fil -len)
 (fact length -fil unknown)
 test
 (numberp -len)
 -->
 (remove 2))

((fact length -fil1 -len)
 (fact sampled -fil2 -fil1 -sr)
 test
 (numberp -len)
 -->
 (fact fact length -fil2 ^(fix (divide -len -sr))))

; ---NOTE-----------------------------------------------------------------------

((fact length -f1 -l1)
 (fact length -f2 -l2)
 test
 (and
   (not (equal -f1 -f2))
   (not (equal -l1 -l2)))
 -->
 (cond
   ((not (numberp -l1))
     (fact message ("The length of the file" -f1 "is unknown. This may"
                    "lead to trouble later on.")))
   ((and (numberp -l1) (numberp -l2))
```

```
    (fact message ("The files" -f1 "and" -f2 "are not equally long."
                   "You must CUT one or both before using an identification"
                   "algorithm.")))))

((note interpret stat)
 -->
 (fact message ("If maximum or minimum deviate very much from mean,"
                "this may indicate outliers.")))

((note pick rates must be equal -r1 -r2)
 test
 (not (equal -r1 -r2))
 -->
 (fact message ("You must pick the insignal and outsignal with the same"
                "frequency. This will not work. PICK once again.")))

((note select two interesting regions)
 -->
 (fact message ("I suggest that you try to select two different regions"
                "that are free of strange disturbances and other things"
                "that might not be part of the real system behavior."
                "This can only be done if the file is long enough, i.e.,"
                "if there is any noise, some one hundred points.")))

((note two interesting regions)
 -->
 (fact message ("Select the two regions. If you can't find any"
                "good choice, just take the first and second half of"
                "the data files. You must CUT out two regions if"
                "you want to do cross validation later.")))

((note cut insignal of first half -x)
 (fact length -x -len)
 -->
 (cond
   ((and (numberp -len) (> -len 50))
     (fact message ("Use CUT to cut out the first half of the input signal"
                    "in the file" -x ". Use first record = 1 and"
                    "number of records =" ^(fix (times -len 0.50)) ".")))
   ((and (numberp -len) (< -len 50))
     (fact message ("The file is probably too short to use for a"
                    "meaningful cross validation.")))
   ((equal -len 'unknown)
     (fact message ("Use CUT to cut out the first half of the input signal"
                    "in the file" -x "if the file is not too short.")))))

((note cut insignal of second half -x)
 (- cut-from 1 .-)
 -->
 (fact message ("Use CUT to cut out the second half of the input signal"
                "in the file" -x ".")))

((note cut outsignal of first half -x)
 -->
 (fact message ("Use CUT to cut out the first half of the output signal"
                "in the file" -x ".")))

((note cut outsignal of second half -x)
 (- cut-from 1 .-)
 -->
 (fact message ("Use CUT to cut out the second half of the output signal"
                "in the file" -x ".")))

((note xval-1-cut -file1)
 (fact cut-from 1 - -file1 -start -length)
 -->
 (fact message ("The first part of the file you cut out started at"
                -start "and had length" -length ".")))

((note earlier cut -file)
 (fact cut-from - -file -x -start -length)
 -->
 (fact message ("The last file you CUT was" -x ", the starting column"
                -start "and the record length" -length ".")))

((note earlier xval-1-cut -)
 (fact xval-1-cut - -file -lo -n)
```

```
 -->
 (remove 1)
 (fact message ("The last file you CUT was" -file ", the starting column"
                -lo "and the record length" -n "."))

((note earlier xval-1-cut -file)
 (fact cut-from 1 -file -x -start -length)
 (~(fact xval-1-cut - - - -))
 -->
 (remove 1)
 (fact message ("The last file you CUT was" -file ", the starting column"
                -start "and the record length" -length "."))

((note cut out second part)
 -->
 (fact message ("Cut out a part of the residuals file. Choose a part"
                "that was not used in the parameter estimation."))

((note remove trends from -x)
 -->
 (fact message ("Use the TREND command to remove any"
                "trends in" -x "."))

((note remove trends from second half -x)
 (fact cut-from 1 -x -x1 - -)
 (fact cut-from 1 -x2 -x1 - -)
 (fact trends-removed - -x2 -order)
 test
 (not (equal -x -x2))
 -->
 (remove 1)
 (fact message
  ("On the other half (" -x2 ") you removed trends of order" -order "."))

((note check no trends in -x)
 -->
 (fact message ("Make sure that there are no trends left in" -x "by PLOT.")))

((fact trends-removed -x trends-removed -y -order)
 test
 (> -order 0)
 -->
 (fact message ("You removed trends of an order higher than 0. This"
                "may suggest that the process was not in stationarity."
                "The model may not be valid for control purposes.")))

((note do coh)
 -->
 (fact message ("Check that it is possible to get reasonable results"
                "by making a coherence test. The coherence must be larger"
                "than ~ 0.7 if you are going to use the results in a"
                "parameter estimation and you use one input signal"
                "and expect a deterministic linear model. Use number of"
                "lags = 20 - 25% of the number of data points.")))

((note do aspec)
 -->
 (fact message ("Check that the input has excited the system sufficiently"
                "by examining the input spectrum with ASPEC.")))

((fact coherence-limits -f1 -f2 -locohlim -hicohlim)
 (fact aspec-limits -f1 -loasplim -hiasplim)
 -->
 (fact fact limits -f1 -f2 ~(cond ((< -locohlim -loasplim) -loasplim)
                                  (t -locohlim))
                           ~(cond ((> -hicohlim -hiasplim) -hiasplim)
                                  (t -hicohlim))))

((note do ccoff)
 -->
 (fact message ("Check for direct terms or time delays by making a"
                "cross correlation test with CCOFF on PREWHITEned"
                "inputs and outputs (use the macro PREWHITE). The cross"
                "correlation between input and output should pass"
                "through the origin.")))

((note slide correct number of steps -f1 -f2)
```

```
 (fact slided -f1 -f2 -steps)
 test
 (not (equal -steps 0))
 -->
 (fact message ("Slide the files" -f1 "and" -f2 "so that the correlation"
                "becomes visibly larger than zero in the point 1, i.e.,"
                "use SLID with" -steps "."))

((note start parameter estimation)
 -->
 (fact message ("Parameter estimation may be started. Use the MLID"
                "command to fit models of INCREASING order. Look at the"
                "parameters and the estimated uncertainties. A minimal"
                "AIC is a sign that the model order is high enough."))

((fact resfile -res -)
 (resid-file-list -lis)
 test
 (not (member -res -lis))
 -->
 (remove 2)
 (fact resid-file-list ^(append1 -lis -res)))

((fact identified-from -nr -sys - - -order)
 (system-list -nr -syslis)
 test
 (not (assoc -sys -syslis))
 -->
 (remove 2)
 (fact system-list -nr ^(append1 -syslis (list -sys -order))))

((resid-file-list -res)
 (fact res-file -f)
 test
 (not (member -f -res))
 -->
 (remove 2)
 (fact res-file ^(cons -f -res)))

((note analyze model -this)
 (fact mlid-data -file -aic1 -)
 (fact mlid-data -this -aic2 -)
 (fact identified-from 1 -file - - -order1)
 (fact identified-from 1 -this - - -order2)
 test
 (and
   (> -aic2 -aic1)
   (> -order2 -order1))
 -->
 (remove 1)
 (fact message ("The model order seems to be high enough, according to"
                "the AIC-test.")))

((note white residuals)
 -->
 (remove 1)
 (fact message ("If the order is high enough, the residuals will be white,"
                "i.e., their autocorrelation close to zero except at"
                "tau = zero. Note that small residuals due to quantization"
                "will not be white.")))

((note interpret resid plot)
 -->
 (remove 1)
 (fact message ("Large single residuals may be caused by measurement errors."
                "If there is any, remove them with PLMAG.")))

((note transferfunction B / A -syst)
 -->
 (fact message ("Use SPTRF ... <" -syst "B / A  to produce a frequency"
                "response file from" -syst ".")))

((fact frequency-response -file -syst)
 (bode-plot-list-ba -ba)
 test
 (not (member -file -ba))
 -->
```

```
 (remove 2)
 (fact bode-plot-list-ba ^(append1 -ba -file)))

((note interpret bode plot ba)
 (bode-plot-list-ba -ba)
 -->
 (remove 1)
 (fact message ("Plot the frequency responses for all systems in"
                "one diagram. Use the command BODE" ^(reverse -ba) ".")))

((note transferfunction C / A -syst)
 -->
 (fact message ("Use SPTRF ... <" -syst "C / A to produce a frequency"
                "response file from" -syst ", for the noise influence.")))

((fact noise-response -file -syst)
 (bode-plot-list-ca -ca)
 test
 (not (member -file -ca))
 -->
 (remove 2)
 (fact bode-plot-list-ca ^(append1 -ca -file)))

((note interpret bode plot ca)
 (bode-plot-list-ca -ca)
 -->
 (remove 1)
 (fact message ("Plot the frequency responses for the noise influence of"
                "all systems in one diagram. Use the command BODE"
                ^(reverse -ca) ".")))

((note compare with deter -simulated -real)
 -->
 (fact message ("The simulated signal," -simulated "should agree well with"
                "the real output from the system," -real ". If not, try"
                "another model.")))

((note coherence test -insi -outsi)
 (fact limits -insi -outsi -low -high)
 -->
 (fact message ("The 0.7 limits of the insignal and outsignal"
                "which you have identified from" "(" -insi -outsi ")"
                "are" -low "to" -high ". This is the interesting region in"
                "the Bode diagram.")))

((suggest cross validation or mlid)
 (~(- cut-from 1 .-))
  -->
 (remove 1)
 (fact message ("You may perform another estimation with a higher"
                "model order.")))

((suggest cross validation or mlid)
 (- cut-from 1 .-)
 -->
 (remove 1)
 (fact message ("You may perform another estimation with a higher"
                "model order, or you may start a cross validation"
                "with the part of the signal you have not used for"
                "estimation.")))

((note cross validation)
 (~ (fact cut before estimation))
 -->
 (fact message ("You can not cross validate because two data sets"
                "are not available.")))

((note cross validation)
 (fact cut before estimation)
 (system-list 1 -syslis)
 test
 (> (length -syslis) 1)
 -->
 (remove 1)
 (fact message ("The following system files are available for"
                "cross validation: "
                ^(mapcar 'car -syslis) ". The cross validation"
```

```
                "is started with the command TREND, where trends are"
                "removed from the entire files and loss functions are"
                "computed for the second half.")))

  ((note cross validation)
   -->
   (remove 1))

  ((note remove trends for xval-1 -fil)
   (fact cut-from 1 -x -fil - -)
   (fact trends-removed - -x -order)
   (system-list 1 -syslis)
   test
   (> (length -syslis) 1)
   -->
   (remove 1)
   (fact message ("Earlier you removed trends of order" -order "on a file"
                "cut from" -fil ". It is advisable that you do the same"
                "now.")))

  ((note remove trends for xval-1 -fil)
   (fact cut-from 1 - - - -)
   (system-list 1 -syslis)
   test
   (> (length -syslis) 1)
   -->
;  (remove 1)
   (fact message ("Remove the trends from the complete insignal"
                "and outsignal, for cross validation.")))

  ((note compute cross validation residuals)
   (xval-1-resid-file-list -res)
   (system-list 1 -sys)
   -->
   (let ((systems))
     (do
       ((resids -res (cdr resids))
        (syst   (mapcar 'car -sys)))
       ((null resids)
         (setq systems syst))
       (setq syst (wegnehmen (car resids) syst)))
   (remove 1)
   (cond
     (systems
       (fact message
         ("Compute residuals using the complete input and output"
          "signals and the"
          ^(cond ((> (length systems) 1) "systems") (t "system"))
          ^systems "."))))))

; Find minimal AIC

  ((fact mlid-data -syst -aic -)
   (~  (aic-minimum - -))
   -->
   (fact aic-minimum -syst -aic))

  ((aic-minimum -syst -aic)
   (fact identified-from 1 -syst - - -order)
   (~ (aic-minimum-set -))
   -->
   (fact aic-minimum-set ((-syst -order -aic))))

  ((aic-minimum-set -set)
   (aic-minimum -syst1 -aic1)
   (fact mlid-data -syst2 -aic2 -)
   (fact identified-from 1 -syst2 - - -order2)
   test
   (and
    (< -aic2 -aic1)
    (not (member (list -syst2 -order2 -aic2) -set)))
   -->
   (remove 1 2)
   (fact aic-minimum -syst2 -aic2)
   (let
    ((result nil))
    (do
```

```
      ((tmp -set (cdr tmp))
       (res ()))
      ((null tmp) (setq result res))
      (cond
        ((< (caddar tmp) (plus -aic2 (times 0.05 (abs -aic2))))
          (setq res (append1 res (car tmp))))))
    (setq result (cons (list -syst2 -order2 -aic2) result))
    (fact aic-minimum-set ^result)))

  ((aic-minimum-set -set)
   (aic-minimum -syst1 -aic1)
   (fact mlid-data -syst2 -aic2 -)
   (fact identified-from 1 -syst2 - - -order2)
   test
   (and
    (>= -aic2 -aic1)
    (not (member (list -syst2 -order2 -aic2) -set))
    (< -aic2 (plus -aic1 (times 0.05 (abs -aic1)))))
   -->
   (remove 1)
   (fact aic-minimum-set ^(append1 -set (list -syst2 -order2 -aic2))))

; Find minimal loss function of xval-1

 ((fact xval-1-resfile -resf -sys)
  (xval-1-resid-file-list -list)
  test
  (not (member -sys -list))
  -->
  (remove 2)
  (fact xval-1-resid-file-list ^(append1 -list -sys)))

 ((fact xval-1 has loss -resfc -loss)
  (fact xval-1-resfile -resf -syst)
  (fact xval-1-cut -resfc -resf - -)
  (^ (xval-1-minimum - -))
  -->
  (fact xval-1-minimum -syst -loss))

  ((xval-1-minimum -syst -loss)
   (fact identified-from 1 -syst - - -order)
   (^ (xval-1-minimum-set -))
   -->
   (fact xval-1-minimum-set ((-syst -order -loss))))

  ((xval-1-minimum-set -set)
   (xval-1-minimum -syst1 -loss1)
   (fact xval-1 has loss -resfc -loss2)
   (fact xval-1-resfile -resf -syst2)
   (fact xval-1-cut -resfc -resf - -)
   (fact identified-from 1 -syst2 - - -order2)
   test
   (and
    (< -loss2 -loss1)
    (not (member (list -syst2 -order2 -loss2) -set)))
   -->
   (remove 1 2)
   (fact xval-1-minimum -syst2 -loss2)
   (let
    ((result nil))
    (do
      ((tmp -set (cdr tmp))
       (res ()))
      ((null tmp) (setq result res))
      (cond
        ((< (caddar tmp) (plus -loss2 (times 0.05 (abs -loss2))))
          (setq res (append1 res (car tmp))))))
    (setq result (cons (list -syst2 -order2 -loss2) result))
    (fact xval-1-minimum-set ^result)))

  ((xval-1-minimum-set -set)
   (xval-1-minimum -syst1 -loss1)
   (fact xval-1 has loss -resfc -loss2)
   (fact xval-1-resfile -resf -syst2)
   (fact xval-1-cut -resfc -resf - -)
   (fact identified-from 1 -syst2 - - -order2)
   test
```

```
   (and
    (>= -loss2 -loss1)
    (not (member (list -syst2 -order2 -loss2) -set)))
    (< -loss2 (plus -loss1 (times 0.05 (abs -loss1))))
   -->
   (remove 1)
   (fact xval-1-minimum-set ^(append1 -set (list -syst2 -order2 -loss2))))

; Find minimal loss function of xval-2

 ((fact xval-2-resfile -resf -sys)
  (xval-2-resid-file-list -list)
  test
  (not (member -sys -list))
  -->
  (remove 2)
  (fact xval-2-resid-file-list ^(append1 -list -sys)))

 ((fact xval-2 has loss -resfc -loss)
  (fact xval-2-resfile -resf -syst)
  (fact xval-2-cut -resfc -resf - -)
  (~ (xval-2-minimum - -))
  -->
  (fact xval-2-minimum -syst -loss))

  ((xval-2-minimum -syst -loss)
   (fact identified-from 2 -syst - - -order)
   (~ (xval-2-minimum-set -))
   -->
   (fact xval-2-minimum-set ((-syst -order -loss))))

  ((xval-2-minimum-set -set)
   (xval-2-minimum -syst1 -loss1)
   (fact xval-2 has loss -resfc -loss2)
   (fact xval-2-resfile -resf -syst2)
   (fact xval-2-cut -resfc -resf - -)
   (fact identified-from 2 -syst2 - - -order2)
   test
   (and
    (< -loss2 -loss1)
    (not (member (list -syst2 -order2 -loss2) -set)))
   -->
   (remove 1 2)
   (fact xval-2-minimum -syst2 -loss2)
   (let
    ((result nil))
    (do
      ((tmp -set (cdr tmp))
       (res ()))
      ((null tmp) (setq result res))
      (cond
        ((< (caddar tmp) (plus -loss2 (times 0.05 (abs -loss2))))
          (setq res (append1 res (car tmp))))))
    (setq result (cons (list -syst2 -order2 -loss2) result))
    (fact xval-2-minimum-set ^result)))

  ((xval-2-minimum-set -set)
   (xval-2-minimum -syst1 -loss1)
   (fact xval-2 has loss -resfc -loss2)
   (fact xval-2-resfile -resf -syst2)
   (fact xval-2-cut -resfc -resf - -)
   (fact identified-from 2 -syst2 - - -order2)
   test
   (and
    (>= -loss2 -loss1)
    (not (member (list -syst2 -order2 -loss2) -set)))
    (< -loss2 (plus -loss1 (times 0.05 (abs -loss1))))
   -->
   (remove 1)
   (fact xval-2-minimum-set ^(append1 -set (list -syst2 -order2 -loss2))))

  ((note xval-2-cut -)
   (number of original -nr1)
   (number of xval-1   -nr2)
   test
   (not (equal -nr1 -nr2))
   -->
```

```
 (remove 1))

((note xval-2-cut -fil)
 (fact cut-from 2 - fil -start -no)
 -->
 (remove 1)
 (fact message ("The last cut in the second part of the cross validatiion"
                "was start at" -start "and number of records" -no ".")))

((note xval-2-cut -fil)
 (fact cut-from 1 - -fil -start -no)
 -->
 (fact message ("Start the second part of the cross validation by cutting"
                "out a part of the file" -fil "you have not identified on"
                "earlier. For the first estimation you used" -no "points"
                "starting at" -start ". The idea is to estimate models"
                "on two halves and to compute the loss functions of"
                "the residuals using the other halves.")))

((fact xval-1 has loss -resf -loss)              ; Trick rule
 -->
 (fact fact xval-1 has-loss -resf -loss))

((fact xval-2 has loss -resf -loss)              ; Trick rule
 -->
 (fact fact xval-2 has-loss -resf -loss))

((fact mlid-data -syst - -loss)                  ; Trick rule
 -->
 (fact fact original has-loss -syst -loss))

((fact -type has-loss -resf -loss)
 (~ (number of -type -))
 -->
 (remove 1)
 (fact number of -type 1))

((fact -type has-loss -resf -loss)
 (number of -type -nr)
 -->
 (remove 1)
 (remove 2)
 (fact number of -type ^(+ -nr 1)))

((number of original -nr1)
 (number of xval-1 -nr1)
 -->
 (fact fact xval-1 performed))

((fact xval-2 may be performed)
 (fact xval-1 performed)
 -->
 (fact message ("You should now compute models of the second half"
                "and compute loss functions using the first half.")))

((note earlier orders - -)
 -->
 (fact message ("Now run MLID on a part of the signal not earlier used.")))

((note earlier orders - -)
 (~ (order-list -))
 (system-list 1 -list)
 -->
 (fact order-list ^(mapcar 'cadr -list)))

((note possible earlier orders)
 (order-list -list)
 (note xval-2 model - -order)
 test
 (member -order -list)
 -->
 (remove 1 2)
 (fact order-list ^(wegnehmen -order -list)))

((note possible earlier orders)
 (order-list -list)
 test
```

```
  (not (null -list))
  -->
  (fact message ("Models of order" -list "should be handled.")))

 ((note cut for loss computations -file1 -file2)
  (fact cut-from 1 -file2 - -st -len)
  -->
  (fact message ("To eliminate initial transients"
                 "CUT out the last 80 percent of the file" -file1 ", i.e.,"
                 "starting column =" ^(fix (times 0.20 -len))
                 "and record length =" ^(fix (times 0.80 -len)) ".")))
;
; Decide which model to choose
;

 ((note interpret result table)
  (number of xval-2 -nr)
  (number of xval-1 -nr)
  (xval-2-minimum-set -s1)
  (xval-1-minimum-set -s2)
  -->
  (let
    ((xval-2-set (sort -s1 '(lambda (x y) (< (cadr x) (cadr y)))))
     (xval-1-set (sort -s2 '(lambda (x y) (< (cadr x) (cadr y))))))
    (do
      ((a xval-2-set)
       (x xval-1-set))
      ((cond
        ((or (null a) (null x))
          (fact message ("Since the two cross validations do not"
                         "coincide the results are not trustworthy."))
          t)
        ((equal (cadar a) (cadar x))
          (fact xval-1 and xval-2 coincide ^(caar a) ^(cadar a))
          t)))
      (cond
        ((> (cadar a) (cadar x)) (setq x (cdr x)))
        ((> (cadar x) (cadar a)) (setq a (cdr a)))))))

 ((xval-1 and xval-2 coincide -mod -ord)
  (aic-minimum-set -set)
  -->
  (let ((ss (sort -set '(lambda (x y) (< (cadr x) (cadr y)))))
        (res ()))
    (do
      ((aic-set ss (cdr aic-set)))
      ((or (null aic-set) (equal (cadar aic-set) -ord))
        (setq res (cadar aic-set))))
    (cond
      (res
        (fact fact best model -mod -ord)
        (fact message ("The model" -mod "is probably the best possible"
                       "since the loss functions and the AIC have"
                       "a common minimum. The order of this model is"
                       -ord ".")))
      (t
        (fact fact best model -mod -ord)
        (fact message ("Use the model" -mod ". It is probably the best"
                       "model although the AIC does not agree"
                       "with the loss functions."))))))

 ((note interpret result table)
  (number of xval-2 -nr1)
  (number of xval-1 -nr2)
  test
  (not (equal -nr1 -nr2))
  -->
  (remove 1))

 ((note create step signal for monte carlo -file)
  (cut before estimation)
  (fact length -file -len)
  (number of xval-2 -nr)
  (number of xval-1 -nr)
  test
  (numberp -len)
  -->
```

```
 (remove 1)
 (fact message ("Use INSI, STEP, and X to create a step insignal"
                "for use in a Monte Carlo simulation of step responses."
                "Use INSI to create a step input of length" -len "."))

((note create step signal for monte carlo)
 (number of original -nr)
 (~ (cut before estimation))
 -->
 (remove 1)
 (fact message ("Use INSI, STEP, and X to create a step insignal"
                "for use in a Monte Carlo simulation of step responses.")))

((note step response distribution of best model)
 -->
 (fact message ("There should not be too much deviation in the step"
                "responses. If there is, the model may not be useful."
                "However, if the signals are short, the static gain"
                "may have some variation.")))

((note transferfunction distribution of best model)
 (fact limits -f1 -f2 -lo -hi)
 -->
 (fact message ("If the transfer functions do not coincide in the"
                "interval" -lo "to" -hi "there may be something wrong.")))

((note transferfunction distribution of best model)
 (~(fact coherence-limits -f1 -f2 -lo -hi))
 -->
 (fact message ("If the transfer functions do not coincide in the"
                "medium frequencies there may be something wrong.")))

((note try to fix a parameter)
 -->
 (fact message ("If there are any parameters, whose values are as small"
                "as their variances, try to set them to zero, with the"
                "FIX subcommand in MLIDSC.")))

((note try to fix another parameter -syst1)
 (fact fixed-aic -syst1 -aic1)
 (fact best model -syst2 -)
 (fact mlid-data -syst2 -aic2 -)
 test
 (> -aic2 -aic1)
 -->
 (remove 1)
 (fact message ("It worked fine to fix one parameter. If there is another"
                "parameter which is as small as its variance, try to FIX"
                "it too.")))

((note try to fix another parameter -syst1)
 (fact fixed-aic -syst1 -aic1)
 (fact best model -syst2 -)
 (fact mlid-data -syst2 -aic2 -)
 test
 (< -aic2 -aic1)
 -->
 (remove 1)
 (fact message ("The last FIX didn't work. Do not use this model.")))

((note common sense validation)
 (fact best model -model -order)
 test
 (> -order 4)
 -->
 (fact message ("You have fitted a model with the order" -order "."
                "Maybe you should be suspicious about such a high"
                "model order.")))

((note check consistency)
 (fact best model - -order)
 (guessed lower model order limit -lowguess)
 test
 (< -order -lowguess)
 -->
 (fact message ("You have fitted a model with"
                "a lower order than your a priori guess.")))
```

```
((note check consistency)
 (fact best model - -order)
 (guessed upper model order limit -upguess)
 test
 (> -order -upguess)
 -->
 (fact message ("You have fitted a model with"
                "a higher order than your a priori guess.")))

((note cross validation)
 (- cut-from 1 .-)
 -->
 (fact message ("If you want to perform a cross validation you"
                "should pick out another interesting region of the"
                "signals and use it for a new fitting procedure.")))

((note choose model)
 (aic-minimum -syst -)
 (~(resvar-minimum - -))
 (~(xval-1-minimum - -))
 -->
 (fact message ("Probably the best model is that in the file" -syst
                ", according to the AIC.")))

((note xval-1 may be performed)
 (fact xval-1 performed)
 (~ (fact cut before estimation))
 -->
 (fact message ("You can not cross validate because two data sets"
                "are not available.")))

((note xval-1 may be performed)
 (fact xval-1 performed)
 (fact cut before estimation)
 (number of original -nr1)
 (number of xval-1 -nr1)
 -->
 (remove 1)
 (fact message ("The following system files are available for"
                "cross validation: " -syslis ". The cross validation"
                "is started with the command CUT, where parts of the"
                "signals not earlier estimated on are cut out.")))

((note trend second half as first half -fil)
 (fact trends-removed -res -fil -order)
 -->
 (fact message ("You removed trends of order" -order "from file" -fil ".")))

((note plot simulated and real outsignal -f1 -f1)
 -->
 (fact message ("Plot the simulated and real output signal in the"
                "same diagram. Give the command PLOT" -f1 "/" -f2 ".")))

; ---SUGGEST-------------------------------------------------------------------

((suggest take care of insignal)
 -->
 (fact message ("This script assumes that the input signal is"
                "available in ASCII format, use CONV"
                "to convert it to binary format."
                "After this, look at it with PLOT.")))


((suggest take care of outsignal)
 -->
 (fact message ("This script assumes that the output signal is"
                "available in ASCII format, use CONV"
                "to convert it to binary format."
                "After this, look at it with PLOT.")))

((suggest modify outliers in insignal)
 -->
 (fact message ("If there are any obviously crazy points in the insignal"
                "you should consider avoiding them or changing them with"
                "PLMAG. But beware! You might be making a fool"
                "of yourself.")))
```

```
  ((suggest modify outliers in outsignal)
   -->
   (fact message ("If there are any outliers in the outsignal, they must"
                  "be removed. The best way is if you can use parts of"
                  "the signals where there are no outliers. Otherwise use"
                  "PLMAG to change the values of the outlier points.")))

  ((suggest sample in and outsignals -in -out)
   -->
   (fact message ("When identifying it is not recommended that you sample the"
                  "system too fast. 3 - 5 samples per time constant is"
                  "a good choise. If necessary use the command PICK to"
                  "sample the signals" -in "and" -out "."
                  "If the input signal is a PRBS signal it may be dangerous"
                  "to sample it if it is not synchronized with the sampling"
                  "period.")))

   ((suggest cross validation or mlid)
    (- cut-from 1 .-)
    -->
    (fact message ("Either continue with the parameter estimation using"
                   "another model order or start the cross validation.")))

   ((suggest cross validation or mlid)
    (fact identified-from 1 -syst -insi -outsi -order)
    (fact mlid-data -syst -aic -loss)
    -->
    (fact message ("You have identified the model" -syst "with order" -order
                   "aic =" -aic "and loss function =" -loss ".")))

  ((suggest remove more trends -last)
   -->
   (fact message ("If there still are trends in the signal give the"
                  "TREND command with a higher order polynomial."
                  "The last trend polynomial you removed was of order"
                  -last ", a higher order may be appropriate.")))

; ---Other rules-----------------------------------------------------------

  ((fact length -file -len)
   (~ (nplx advice given))
   test
   (numberp -len)
   (not (equal -len 400))
   -->
   (fact nplx advice given)
   (fact message ("Set appropriate plot width with LET NPLX.=" -len ".")))

)))
```

Listing of the file script.l

```
; This file contains the script flavor. Everything that happen to
; the script, and notably the matching proper, is coded here.

(defflavor script-flavor
  (full-name                        ;
   script-name                      ;  BEWARE !
   command-sequence                 ;  THIS IS USED IN superscr-flavor
   big-list                         ;  for the method :copy-superscript.
   file-bindings                    ;
   kscalls                          ;
   assigns                          ;
   yaps-info)                       ;
  (yaps-database))

; Is called from the matcher via db. Command is the command list
; generated by the parser.

(defmethod (script-flavor :match-command) (command)
  (setq yaps-info nil)

  (let
    ((tmp))
    ;
    ; Returns a list of the matching scripts.
```

```
    ;
    (setq tmp
      (apply 'append
        (mapcar
          '(lambda (x) (<- self ':match-command-and-filenames x command))
           command-sequence)))
    ;
    ; Pairs internal and external filenames.
    ;
      (mapcar
        '(lambda (x)
          (<- self ':extract-filenames (car x) command))
        tmp)))


; Match-command-finally is called from Query to perform
; the final updating of the command sequences.

(defmethod (script-flavor :match-command-finally) (command)
  (setq yaps-info nil)

  (let
    ((tmp) (command-sequence-aux))
    ;
    ; Returns a list of the matching scripts.
    ;
    (setq command-sequence-aux
      (apply 'append
        (mapcar
          '(lambda (x) (<- self ':match-command-and-filenames x command))
           command-sequence)))
    ;
    ; Pairs internal and external filenames.
    ;
    (setq tmp
      (mapcar
        '(lambda (x)
           (<- self ':extract-filenames (car x) command))
         command-sequence-aux))
    ;
    ; Removes the first command in a script and returns the expanded script,
    ; thereby keeping the local database updated.
    ;
    (cond
      (tmp
        (setq command-sequence
          (apply 'append
            (mapcar
              '(lambda (x) (<- self ':update-command-sequence x))
               command-sequence-aux)))))
    tmp))

(defmethod (script-flavor :perform-kscalls) ()
  (setq yaps-info nil)
  (cond
    (kscalls
      (mapc
        '(lambda (fact)
           (let
             ((subst-fact (<- self ':substitute-parameters fact)))
             (<- self ':put-fact subst-fact)))
         (reverse kscalls))
      (cond
        ((<- database ':get-yaps-debug)
          (<- self 'allp)
          (<- self 'db)
          (<- self 'trace)
          (<- self ':run-yaps)
          (<- self 'untrace))
        (t
          (<- self ':run-yaps)))
      (setq kscalls nil))))

; Used for performing (assign ...). The grammar is (new old).

(defmethod (script-flavor :make-assigns) ()
  (cond
```

```
    (assigns
      (mapc
        '(lambda (x)
           (<- self ':new-external-name
             (car x)
             (<- self ':external-name (cadr x))))
        assigns)
      (setq assigns nil))))

(defmethod (script-flavor :save-yaps-info) (info)
  (setq yaps-info (append1 yaps-info info)))

; Given the script file variable "name", this method returns
; the actual value of this variable.

(defmethod (script-flavor :external-name) (name)
  (cadr (assoc name file-bindings)))

(defmethod (script-flavor :new-external-name) (intname extname)
  (setq file-bindings (cons (list intname extname) file-bindings)))

(defmethod (script-flavor :next-commands) ()
  (remove-multiple
    (mapcar 'cadar command-sequence)))

(defmethod (script-flavor :initialize) ()
  (setq
    command-sequence
    (apply 'append
      (mapcar
        '(lambda (x)
          (setq assigns nil)
          (setq big-list nil)
          (<- self ':expand-script x))
        command-sequence)))
  (<- self ':perform-kscalls))

(defmethod (script-flavor :dump-script) ()
  (<- uinterf ':writeln
   (list
     "Script name:"
     (<- database ':get-script-name (<< self 'script-name))))
  (<- uinterf ':writeln '("Command sequences:"))
  (<- uinterf ':line 18)
  (<- uinterf ':writeln '(""))
  (mapcar
    '(lambda (cs nr)
       (<- uinterf ':writeln `("Command sequence number "  ,nr))
       ($prpr (car cs))
       (<- uinterf ':writeln '(""))
       ($prpr (cadr cs))
       (<- uinterf ':writeln '(""))
       ($prpr (caddr cs))
       (<- uinterf ':writeln '(""))
       (<- uinterf ':writeln `(... + ,(length (cddr cs))))
       (<- uinterf ':line (cond ((< nr 100) 8) (t 9)))
       (<- uinterf ':writeln '("")))
    (<< self 'command-sequence)
    (number-list (length (<< self 'command-sequence))))
  (<- uinterf ':write (list "File bindings: "))
  (mapc
    '(lambda (x)
      (patom x)
      (patom " "))
    (<< self 'file-bindings))
  (<- uinterf ':writeln '(""))
  (<- uinterf ':writeln (list "KSCalls:" (<< self 'kscalls)))
  (<- uinterf ':writeln (list "YAPS info:" (<< self 'yaps-info)))
  (<- uinterf ':writeln (list "Database:"))
  (<- self 'db)
  (<- uinterf ':line 79)
  (<- uinterf ':writeln '("")))

; Internal methods ------------------------------------------------------------

; Matches the command name and file arguments.
```

```
(defmethod (script-flavor :match-command-and-filenames) (pattern command)
  (cond
    ((and
      (equal (cadar pattern) (car command))
      (<- self ':match-filenames (cddar pattern) (cdr command)))
     (list pattern))
    (t
      nil)))

; Pattern is a command-description ((outfile f1) (infile f2)).
; Command is a parsed command from the parser ((outfile N) < (infile P))
; with (command name) stripped off.

(defmethod (script-flavor :match-filenames) (pattern command)
  (cond
    ((null pattern)
      t)
    ((null command)
      nil)
    ((and
      (<- self ':infile? pattern)
      (<- self ':infile? command)
      (<- self ':external-name (cadar pattern)) ; The external name must
                                                ; exist!
      (or
        (equal '*unknown* (cadar command))
        (equal (<- self ':external-name (cadar pattern)) (cadar command)))
      (<- self ':match-filenames (cdr pattern) (cdr command))))
    ((and
      (<- self ':outfile? pattern)
      (<- self ':outfile? command)
      (<- self ':match-filenames (cdr pattern) (cdr command))))
    ((and
      (<- self ':globfile? pattern)
      (<- self ':infile? command)
      (<- self ':match-filenames (cdr pattern) (cdr command))))
    ((member (caar pattern) '(infile outfile globfile))
      (<- self ':match-filenames pattern (cdr command)))
    (t
      (<- self ':match-filenames (cdr pattern) command))))

; This method returns the command with both the internal and the actual
; external filenames inserted. (If possible else '*'.)

(defmethod (script-flavor :extract-filenames) (pattern command)
  (cond
    ((null command) nil)
    ((equal (car pattern) 'command)
      (cons (car command)
        (<- self ':extract-filenames (cddr pattern) (cdr command))))
    ((<- self ':outfile? command)
      (cons
        (list 'outfile (list (cadar pattern) (cadar command)))
        (<- self ':extract-filenames
          (cdr pattern)
          (cdr command))))
    ((and (<- self ':infile? command) (<- self ':infile? pattern))
      (cons
        (cond
          ((equal (cadar command) '*unknown*)
            (list 'infile
              (list
                (cadar pattern)
                (<- self ':external-name (cadar pattern))
                'default)))
          (t
            (list 'infile (list (cadar pattern) (cadar command)))))
        (<- self ':extract-filenames (cdr pattern) (cdr command))))
    ((and (<- self ':infile? command) (<- self ':globfile? pattern))
      (cons
        (list 'globfile (list (cadar pattern) (cadar command)))
        (<- self ':extract-filenames
          (cdr pattern)
          (cdr command))))
    ((and
      (not (atom (car command)))
      (member
```

```
      (caar command)
      '(number numlist numlist1 symbol symlist symlist1))
      (equal (caar pattern) (caar command)))
      (cons
        (list 'parameter (list (cadar pattern) (cadar command)))
        (<- self ':extract-filenames
          (cdr pattern)
          (cdr command))))
    (t
      (cons
        (car command)
        (<- self ':extract-filenames pattern (cdr command))))))

; Here is the basic matcher, that really performs the work.
; As in argument this method takes one script. It returns
; a list of scripts with a command as first item.

(defmethod (script-flavor :expand-script) (script)
  (cond
    ((null script)
      '(((empty-script))))
    ((member script big-list) '())
    (t
      (setq big-list (cons script big-list))
      (cond
        ((<- self ':empty-script? script)
          '(((empty-script))))
        ((<- self ':command? script)
          (list script))
        ((<- self ':assign? script)
          (setq assigns (append1 assigns (cdar script)))
          (<- self ':expand-script (cdr script)))
        ((<- self ':kscall? script)
          (setq kscalls (append kscalls (cdar script)))
          (<- self ':expand-script (cdr script)))
        ((<- self ':repeat? script)
          (<- self ':expand-script
            (append
              (cadar script)
              (list (list 'repeat* (cadar script)))
              (cdr script))))
        ((<- self ':repeat*? script)
           (append
             (<- self ':expand-script
               (append
                 (cadar script)
                 script))
             (<- self ':expand-script (cdr script))))
        ((<- self ':scriptmacro? script)
          (<- self ':expand-script
            (append
              (<- self
                ':expand-scriptmacro
                (cadar script)                  ; Name of script-macro.
                (assoc 'in (cddar script))      ; Actual in parameters.
                (assoc 'out (cddar script)))    ; Actual out parameters.
              (cdr script))))
        ((<- self ':or? script)
          (apply
            'append
            (mapcar
              '(lambda (x)
                 (<- self ':expand-script (append x (cdr script))))
              (cdar script))))
        ((<- self ':all? script)
          (apply
            'append
            (mapcar
              '(lambda (x)
                 (<- self ':expand-script
                   (append
                     x
                     (cond
                       ((null (cddar script)) nil)
                       (t (list (wegnehmen x (car script)))))
                     (cdr script))))
              (cdar script)))))))))
```

```
(defmethod (script-flavor :update-command-sequence) (script)
  (setq big-list nil)
  (setq assigns nil)
  (<- self ':expand-script (cdr script)))

(defmethod (script-flavor :substitute-parameters) (fact-list)
  (mapcar
    '(lambda (x)
       (cond
         ((atom x)
           x)
         ((and (listp x) (equal (car x) 'parameter)
           (<- self ':external-name (cadr x))))
         ((and (listp x) (equal (car x) 'parameter))
           (list 'UNBOUND-PARAMETER: (cadr x)))
         ((listp x)
           (<- self ':substitute-parameters x))))
   fact-list))

; ------------------------------------------------------------------------

; (<- obj ':merge (a b c) and (1 2 3)) => ((a 1) (b 2) (c 3))

(defmethod (script-flavor :merge) (l1 l2)
  (cond
    ((null l1)
      '())
    (t
      (cons
        (list (car l1) (car l2))
        (<- self ':merge (cdr l1) (cdr l2))))))

; Methods for handling scriptmacros. Assign in a scriptmacro is not
; handled. (This will come in a later version.)

(defmethod (script-flavor :expand-scriptmacro) (name actual-in actual-out)
  (cond
    ((and
      (equal
        (length (<- database ':ins-of-macro name))
        (length (cdr actual-in)))
      (equal
        (length (<- database ':outs-of-macro name))
        (length (cdr actual-out))))
      (<- self ':exchange
        (<- database ':body name)
        (<- self ':merge
          (append
            (<- database ':ins-of-macro name)
            (<- database ':outs-of-macro name))
          (append (cdr actual-in) (cdr actual-out)))))
    (t
      (break))))

; Exchange all (infile x) (outfile x) (globfile x)
; to           (infile y) (outfile y) (globfile y)
;
; where exc is ((x y) ...)

(defmethod (script-flavor :exchange) (pattern exc)
  (cond
    ((null pattern) '())
    ((atom pattern) pattern)
    ((member (car pattern)
      '(infile outfile globfile parameter
        symbol symlist symlist1
        number numlist numlist1))
      (list (car pattern) (cadr (assoc (cadr pattern) exc))))
    ((or
      (equal (car pattern) 'in)
      (equal (car pattern) 'out))
        (cons (car pattern)
          (mapcar '(lambda (x) (cadr (assoc x exc))) (cdr pattern))))
    (t
      (cons
        (<- self ':exchange (car pattern) exc)
```

```
        (<- self ':exchange (cdr pattern) exc)))))

(defmethod (script-flavor :delete) ()
  (comment))

; ------------------------------------------------------------------------

; The definitions of selector methods to work on scripts and commands.
; All are very similar in their structure.

; Gensel generates all the selectors. The code of this procedure is
; dedicated to Ulf Gennser.

(def gensel
  (macro (x)
    `(defmethod (script-flavor ,(concat ': (cadr x) '?)) (y)
        (and
          (listp (car y))
          (equal (caar y) (quote ,(cadr x)))))))

(def generate-selector-methods
  (lambda (list-of-atoms)
    (mapcar
      '(lambda (x) (eval (cons 'gensel (list x))))
      list-of-atoms)))

(generate-selector-methods
  '(infile
    assign
    outfile
    globfile
    repeat
    repeat*
    kscall
    command
    scriptmacro
    empty-script
    or
    all))

; ------------------------------------------------------------------------

; Methods to transform rules before they are put in the YAPS databases
; in the scripts. Not used.

(defmethod (script-flavor :transform-lhs) (rule script-name match-var)
  (cond
    ((null rule)
      (list (list 'script-to-run match-var script-name)))
    ((and
      (listp (car rule))
      (equal (caar rule) '^))
      (cons
        (list '^ (cons match-var (cadar rule)))
        (<- self ':transform-lhs (cdr rule) script-name match-var)))
    (t
      (cons
        (cons match-var (car rule))
        (<- self ':transform-lhs (cdr rule) script-name match-var)))))

(defmethod (script-flavor :transform-rhs) (rule script-name match-var)
  (cond
    ((null rule) ())
    ((and (listp (car rule)) (equal (caar rule) 'save-yaps-info))
      (cons
        `(<- (eval ,match-var) ':save-yaps-info ,(cadar rule))
        (<- self ':transform-rhs (cdr rule) script-name match-var)))
    ((and (listp (car rule)) (equal (caar rule) 'put-fact))
      (cons
        `(<- (eval ,match-var) ':put-fact ,(cadar rule))
        (<- self ':transform-rhs (cdr rule) script-name match-var)))
    (t
      (cons
        (car rule)
        (<- self ':transform-rhs (cdr rule) script-name match-var)))))

(defmethod (script-flavor :transform) (rule)
```

```
  (let ((unik (concat '- (newsym 'unique-))))
    (append
      (<- self ':transform-lhs (extract-lhs rule) script-name unik)
      (list '-->)
      (<- self ':transform-rhs (extract-rhs rule) script-name unik))))

(defmethod (script-flavor :put-fact) (fact)
  (<- self 'fact fact))

(defmethod (script-flavor :install-rule) (rule)
  (let ((rule-name (newsym (concat script-name '-rule-))))
    (setq all-rules (appendi all-rules rule-name))
    (<- self 'buildp rule-name rule)))

(defmethod (script-flavor :run-yaps) ()
  (<- self 'run))
```

Listing of the file scripts.l

```
(setq scripts '(

  (ml
    (script ml
      ("ML"
       "The ML (Maximum Likelihood) estimation script computes and"
       "verifies a model of a transfer function that could have"
       "produced the output (given) from the input (also given).")

      (
       (kscall (system-list 1 ()))
       (kscall (bode-plot-list-ba ()))
       (kscall (bode-plot-list-ca ()))
       (kscall (resid-file-list ()))
       (kscall (xval-1-resid-file-list ()))
       (kscall (xval-2-resid-file-list ()))
       (kscall (question a priori knowledge))
;
; Convert the input signal from ASCII to binary format, and examine it.
;
       (kscall (suggest take care of insignal))
       (command conv (outfile INSIGNAL-T) (globfile INSIGNALDATA))
       (or
        ((command stat (infile INSIGNAL-T))
         (kscall (note interpret stat))
         (kscall (question length of (parameter INSIGNAL-T))))
        ((kscall (fact length (parameter INSIGNAL-T) unknown))))
       (command plot (infile INSIGNAL-T))

       (kscall (suggest modify outliers in insignal))
       (or ((scriptmacro plmag-macro (in INSIGNAL-T))) ())
;
; Convert the output signal from ASCII to binary format, and examine it.
;
       (kscall (suggest take care of outsignal))
       (command conv (outfile OUTSIGNAL-T) (globfile OUTSIGNALDATA))
       (or
        ((command stat (infile OUTSIGNAL-T))
         (kscall (note interpret stat))
         (kscall (question length of (parameter OUTSIGNAL-T))))
        ((kscall (fact length (parameter OUTSIGNAL-T) unknown))))
       (command plot (infile OUTSIGNAL-T))

       (kscall (suggest modify outliers in outsignal))
       (or ((scriptmacro plmag-macro (in OUTSIGNAL-T))) ())
;
; Plot the input signal and output signal in the same plot.
;
       (kscall (note select two interesting regions))
       (scriptmacro double-plot-macro (in INSIGNAL-T OUTSIGNAL-T)(out JUNK OF))
;
; If necessary sample the given signals before identification.
;
       (kscall (suggest sample in and outsignals
                 (parameter INSIGNAL-T) (parameter OUTSIGNAL-T)))
       (or
        ((repeat
          ((command pick (outfile INSIGNAL-TP) (infile INSIGNAL-T)
```

```
                    (number SAMP-NR1))
          (kscall (fact sampled (parameter INSIGNAL-TP)
                    (parameter INSIGNAL-T) (parameter SAMP-NR1)))
          (command pick (outfile OUTSIGNAL-TP) (infile OUTSIGNAL-T)
                    (number SAMP-NR2))
          (kscall (fact sampled (parameter OUTSIGNAL-TP)
                    (parameter OUTSIGNAL-T) (parameter SAMP-NR2)))
          (kscall (note pick rates must be equal (parameter SAMP-NR1)
                                                 (parameter SAMP-NR2)))
          (or
           ()
           ((scriptmacro double-plot-macro
                 (in INSIGNAL-T OUTSIGNAL-T) (out JUNK OF)))))))
       ((assign INSIGNAL-TP INSIGNAL-T)
        (assign OUTSIGNAL-TP OUTSIGNAL-T)))
;
; Cut out one piece in the signals (if the series are long enough) for
; estimation.
;
      (kscall (note two interesting regions))
      (or
       ((kscall (note cut insignal of first half (parameter INSIGNAL-TP)))
        (command cut
         (outfile INSI-1-T) (infile INSIGNAL-TP) (number F1) (number L1))
        (kscall (fact cut before estimation))
        (kscall (fact cut-from 1 (parameter INSI-1-T) (parameter INSIGNAL-TP)
                    (parameter F1) (parameter L1)))
        (kscall (note cut outsignal of first half (parameter OUTSIGNAL-TP)))
        (kscall (note earlier cut (parameter INSI-1-T)))
        (command cut
         (outfile OUTSI-1-T) (infile OUTSIGNAL-TP) (number F2) (number L2))
        (kscall (fact cut-from 1 (parameter OUTSI-1-T)(parameter OUTSIGNAL-TP)
                    (parameter F2) (parameter L2))))
       ((assign INSI-1-T INSIGNAL-TP)
        (assign OUTSI-1-T OUTSIGNAL-TP)))
;
; Remove trends from the cutted signals.
;
      (kscall (note remove trends from (parameter INSI-1-T)))
      (repeat
       ((command trend (outfile INSI-1-OK) (infile INSI-1-T) (number TI-1))
        (kscall (fact trends-removed (parameter INSI-1-OK)
                    (parameter INSI-1-T) (parameter TI-1)))
        (kscall (note check no trends in (parameter INSI-1-OK)))
        (command plot (infile INSI-1-OK))
        (kscall (suggest remove more trends (parameter TI-1)))))

      (kscall (note remove trends from (parameter OUTSI-1-T)))
      (repeat
       ((command trend (outfile OUTSI-1-OK) (infile OUTSI-1-T) (number TI-2))
        (kscall (fact trends-removed (parameter OUTSI-1-OK)
                    (parameter OUTSI-1-T) (parameter TI-2)))
        (kscall (note check no trends in (parameter OUTSI-1-OK)))
        (command plot (infile OUTSI-1-OK))
        (kscall  (suggest remove more trends (parameter TI-2)))))
;
; Compute the coherence and autospectrum.
; Prewhite the output and input and try to detect time delays.
; Slide the signals if necessary.
;
      (or
       ((repeat
         ((or
          ((kscall (note do coh))
           (command coh (outfile COHF) (infile INSI-1-OK) (infile OUTSI-1-OK))
           (kscall (question results of coherence test (parameter INSI-1-OK)
                                                       (parameter OUTSI-1-OK)))
           (kscall (note do aspec))
           (command aspec (outfile ASP) (infile INSI-1-OK))
           (command bode (infile ASP))
           (kscall (question results aspec insignal (parameter INSI-1-OK))))
          ((kscall (note do ccoff))
           (command prewhite (outfile PREWI) (outfile PREWO)
                             (infile INSI-1-OK) (infile OUTSI-1-OK))
           (command ccoff (outfile CCOFF) (infile PREWI) (infile PREWO))
           (kscall (question number of steps to slide (parameter INSI-1-OK)
                                                      (parameter OUTSI-1-OK))))
```

```
        ((kscall (note slide correct number of steps (parameter INSI-1-OK)
                                                     (parameter OUTSI-1-OK)))
           (repeat
            ((command slid (outfile INSI-1) (outfile OUTSI-1)
                           (infile INSI-1-OK) (infile OUTSI-1-OK))
             (command prewhite (outfile PREWI) (outfile PREWO)
                               (infile INSI-1) (infile OUTSI-1))
             (command ccoff (outfile CCOFF) (infile PREWI) (infile PREWO))
             (kscall (question number of steps to slide (parameter INSI-1)
                                                        (parameter OUTSI-1)))
             (kscall (note slide correct number of steps (parameter INSI-1-OK)
                                                         (parameter OUTSI-1-OK))))))
       ((assign INSI-1 INSI-1-OK)
        (assign OUTSI-1 OUTSI-1-OK)))
;
; Estimate the parameters, look at the residuals, compute transfer function
; and plot it in a Bode diagram.
;
      (kscall (note start parameter estimation))
      (repeat
       ((command mlid
         (outfile SYST) (infile INSI-1) (infile OUTSI-1) (number ORDER))
        (kscall
         (fact identified-from 1 (parameter SYST) (parameter OUTSI-1)
               (parameter INSI-1) (parameter ORDER)))
        (kscall (question ml identification result (parameter SYST)))
        (kscall (note analyze model (parameter SYST)))

        (or
         ((command residu (outfile RES) (infile SYST) (infile INSI-1)
                                                      (infile OUTSI-1))
          (kscall (fact resfile (parameter RES) (parameter SYST)))
          (kscall (note white residuals))
          (or ((command page)) ())
          (command kill)
          (command plot (infile RES))
          (kscall (note interpret resid plot)))
         ())

        (or
         ((repeat
           ((or
             ((scriptmacro plmag-macro (in INSI-1)))
             ((scriptmacro plmag-macro (in OUTSI-1))))))
          ((kscall (note transferfunction B / A (parameter SYST)))
           (command sptrf (outfile TRF-1) (infile SYST))
           (kscall (fact frequency-response (parameter TRF-1)
                                            (parameter SYST)))
           (kscall (note interpret bode plot ba))
           (kscall (note coherence test (parameter INSI-1)
                                        (parameter OUTSI-1)))
           (command bode (infile TRF-1))

           (or
            ()
            ((kscall (note transferfunction C / A (parameter SYST)))
             (command sptrf (outfile TRF-2) (infile SYST))
             (kscall (fact noise-response (parameter TRF-2) (parameter SYST)))
             (kscall (note interpret bode plot ca))
             (kscall (note coherence test (parameter INSI-1)
                                          (parameter OUTSI-1)))
             (command bode (infile TRF-2)))))
          ())
         (kscall (suggest cross validation or mlid))))
;
; If the signals have been cutted, compute the residuals of the second
; half with a model estimated from the first half. Compute the loss
; function from the residuals of the second half.
;
      (or
       ((kscall (note cross validation))
        (kscall (note remove trends for cross validation))
        (repeat
         ((kscall (note remove trends for xval-1 (parameter INSIGNAL-TP)))
          (command trend (outfile INSIXV) (infile INSIGNAL-TP))
          (kscall (note check no trends in (parameter INSIXV)))
          (command plot (infile INSIXV))))
```

```
        (repeat
         ((kscall (note remove trends for xval-1 (parameter OUTSIGNAL-TP)))
          (command trend (outfile OUTSIXV) (infile OUTSIGNAL-TP))
          (kscall (note check no trends in (parameter OUTSIXV)))
          (command plot (infile OUTSIXV))))
        (repeat
         ((kscall (note compute cross validation residuals))
          (command residu (outfile XVRES) (globfile SYST) (infile INSIXV)
                                                          (infile OUTSIXV))
          (kscall (fact xval-1-resfile (parameter XVRES) (parameter SYST)))
          (or () ((command page)))
          (command kill)
          (kscall (note cut out second part))
          (kscall (note earlier xval-1-cut (parameter OUTSI-1-T)))
          (command cut (outfile XVCUT) (infile XVRES) (number LO) (number N))
          (kscall (fact xval-1-cut (parameter XVCUT) (parameter XVRES)
                                   (parameter LO) (parameter N)))
          (command loss (infile XVCUT))
          (kscall (question loss for (parameter XVCUT) xval-1 file))))
;
; Fit models from the second halves of the signals, and compute residuals
; with the first halves.
;
        (kscall (note xval-2 may be performed))
        (kscall (note xval-2-cut (parameter INSIGNAL-TP)))
        (command cut (outfile INSI-2-T) (infile INSIGNAL-TP)
                                   (number L-1) (number N-1))
        (kscall (fact cut-from 2 (parameter INSI-2-T) (parameter INSIGNAL-TP)
                                 (parameter L-1) (parameter N-1)))
        (kscall (note xval-2-cut (parameter INSIGNAL-TP)))
        (command cut (outfile OUTSI-2-T) (infile OUTSIGNAL-TP)
                                   (number L-2) (number N-2))
        (kscall (fact cut-from 2 (parameter OUTSI-2-T)(parameter OUTSIGNAL-TP)
                                 (parameter L-2) (parameter N-2)))
        (kscall (note trend second half as first half (parameter INSI-1-T)))
        (command trend (outfile INSI-2-OK) (infile INSI-2-T))
        (kscall (note trend second half as first half
                  (parameter OUTSI-1-T)))
        (command trend (outfile OUTSI-2-OK) (infile OUTSI-2-T))
        (assign INSI-2 INSI-2-OK)
        (assign OUTSI-2 OUTSI-2-OK)
        (kscall (note earlier orders (parameter OUTSI-1)(parameter INSI-1)))
        (repeat
         ((kscall (note possible earlier orders))
          (command mlid (outfile SYS) (infile INSI-2) (infile OUTSI-2)
                        (number ORDER))
          (kscall (note xval-2 model (parameter SYS) (parameter ORDER)))
          (kscall (fact identified-from 2 (parameter SYS) (parameter OUTSI-2)
                                          (parameter INSI-2) (parameter ORDER)))
          (command residu (outfile RES-2) (infile SYS) (infile INSI-1)
                          (infile OUTSI-1))
          (kscall (fact xval-2-resfile (parameter RES-2) (parameter SYS)))
          (or () ((command page)))
          (command kill)
          (kscall (note cut for loss computations (parameter RES-2)
                    (parameter INSI-1-T)))
          (command cut (outfile RC-2) (infile RES-2) (number LO)
                                        (number N))
          (kscall (fact xval-2-cut (parameter RC-2) (parameter RES-2)
                                   (parameter LO) (parameter N)))
          (command loss (infile RC-2))
          (kscall (question loss for (parameter RC-2) xval-2 file))))
         (kscall (note interpret result table)))
        ((assign INSIXV INSIGNAL-TP)
         (assign OUTSIXV OUTSIGNAL-TP)))
;
; If the signals have been cutted, compute models from the second half of the
; signals and compute residuals from the first half. Compute the loss function
; of the residuals.
;
       (or
        ((kscall (note create step signal for monte carlo
                         (parameter INSIGNAL-TP)))
         (command insi (outfile STP))
         (command step)
         (command x)
         (repeat
```

```
          ((command randstep (outfile STEPS) (globfile SYS) (infile STP))
           (kscall (note step response distribution of best model))
           (command randtf (outfile TFS) (infile SYS))
           (kscall (note transferfunction distribution of best model))
           (command deter (outfile DSIM) (infile SYS) (infile INSIGNAL-TP))
           (kscall (note plot simulated and real outsignal
                         (parameter DSIM) (parameter OUTSIGNAL-TP)))
           (or
            ((command plot (infile DSIM) (symbol S) (symbol OF)))
            ((command plot (infile OUTSIGNAL-TP) (symbol S) (symbol DF))))
           (kscall (note compare with deter (parameter DSIM)
                                            (parameter OUTSIGNAL-TP)))))))
;
; Try to fix any parameters, whose values are close to zero.
;
       (or
        ()
        ((kscall (note try to fix a parameter))
         (repeat
          ((command mlidsc (outfile FIXSYS) (infile INSI-1) (infile OUTSI-1))
           (repeat
             ((command fix)))
           (command resume)
           (kscall (question fixed aic (parameter FIXSYS)))
           (command residu (outfile FIXRES) (infile FIXSYS) (infile INSI-1)
                                                            (infile OUTSI-1))
           (kscall (note white residuals))
           (or () ((command page)))
           (command kill)
           (kscall (note try to fix another parameter (parameter FIXSYS)))))))

       (kscall (note check consistency))
       (kscall (note common sense validation))
       (kscall (note choose model))

       (command stop) )))))
```

Listing of the file scrmac.l

```
(setq script-macros '(

 (plmag-macro (in FILE)
  (command plmag (infile FILE))
  (kscall (note start plmag))
  (or () ((command block)))
  (command plbeg)
  (repeat*
   ((or ((command alter)) ((command page)) ((command plbeg)))))
  (or
   ((command x)
    (kscall (fact (parameter FILE) data-may-be-bad)))
   ((command kill)
    (kscall (note kill plmag (parameter FILE))))))

  (double-plot-macro (in F1 F2) (out S1 S2)
   (kscall (note double plot (parameter F1) (parameter F2)))
   (or
    ((command plot (infile F1) (symbol S1) (symbol S2)))
    ((command plot (infile F2) (symbol S1) (symbol S2))))
    (kscall (check other half (parameter F1) (parameter S2) (parameter F2))))

))
```

Listing of the file superscr.l

```
; This is the superscript flavor. Most of the methods are
; used for distributing and collecting information to and
; from the scripts contained in the superscript.

(defflavor superscr-flavor
  (scripts                    ; '(ml-1 ... ml-n corana)
   script-concat-name)
  ())

; Exported methods ------------------------------------------------------------
```

```
(defmethod (superscr-flavor :initialize) ()
  (mapc
    '(lambda (x) (<- (<- self ':script x) ':initialize))
    scripts))

; Run the YAPSes in each of the scripts.

(defmethod (superscr-flavor :run) ()
  (mapc
    '(lambda (x) (<- (<- self ':script x) ':run-yaps))
    scripts))

; Put facts in the YAPS db's.

(defmethod (superscr-flavor :fact) (fact)
  (mapc
    '(lambda (x) (<- (<- self ':script x) 'fact fact))
    scripts))

; Move facts from KSCalls to temporary lists.

(defmethod (superscr-flavor :perform-kscalls) ()
  (mapc
    '(lambda (x) (<- (<- self ':script x) ':perform-kscalls))
    scripts))

(defmethod (superscr-flavor :make-assigns) ()
  (mapc
    '(lambda (x) (<- (<- self ':script x) ':make-assigns))
    scripts))

(defmethod (superscr-flavor :match-command-finally) (com)
  (mapc
    '(lambda (x) (<- (<- self ':script x) ':match-command-finally com))
    scripts))

(defmethod (superscr-flavor :dump-scripts) ()
  (mapc
    '(lambda (x)
      (<- (<- self ':script x) ':dump-script))
    scripts))

(defmethod (superscr-flavor :delete) ()
  (mapc
    '(lambda (x)
       (let ((script-name (concat (<< self 'script-concat-name) '- x)))
         (<- (eval script-name) ':delete)
;        (makunbound script-name)
         (set script-name nil)))
    scripts))

(defmethod (superscr-flavor :copy-superscript) (new-script-concat-name)
  (mapc
    '(lambda (x)
      (let
        ((inst (eval (concat (<< self 'script-concat-name) '- x)))
         (new-full-name (concat new-script-concat-name '- x)))
        (suspendio)
        (set
          new-full-name
          (copyyaps inst))
        (resumeio)
        (>> (eval new-full-name) 'big-list ())
        (>> (eval new-full-name) 'script-name (<< inst 'script-name))
        (>> (eval new-full-name) 'command-sequence (<< inst 'command-sequence))
        (>> (eval new-full-name) 'file-bindings (<< inst 'file-bindings))
        (>> (eval new-full-name) 'kscalls (<< inst 'kscalls))
        (>> (eval new-full-name) 'yaps-info (<< inst 'yaps-info))
        (>> (eval new-full-name) 'full-name new-full-name)))
    scripts)
  (make-instance 'superscr-flavor
    'script-concat-name new-script-concat-name
    'scripts scripts)
)

(defmethod (superscr-flavor :get-yaps-info) ()
  (let ((y script-concat-name))                    ; Bug in flavors.
```

```
    (mapcar
      '(lambda (script)
         (cons script (<< (eval (concat y '- script)) 'yaps-info)))
       scripts)))

(defmethod (superscr-flavor :next-commands) ()
  (let ((y script-concat-name))                    ; Bug in flavors.
    (mapcar
      '(lambda (script)
         (cons script (<- (eval (concat y '- script)) ':next-commands)))
       scripts)))

(defmethod (superscr-flavor :script) (name)
  (eval (concat script-concat-name '- name)))
```

Listing of the file uinterf.l

```
; This file contans the definitions for the immediate user interface.
; The commands are read and handled here, and sent along to the actual
; interface. Visual and VT100's are currently supported.

(defflavor uinterf-flavor
  ()
  (vt100-interface-flavor))

; ---------------------------------------------------------------------------

(defmethod (uinterf-flavor :ihs-top-level) ()
  ;
  ; The Franz Lisp top level is replaced by this new top level.
  ;
  (setq user-top-level
    '(lambda ()
      (do
        ((input-command '($$$)))
        ((equal (car input-command) 'lisp)
          (setq user-top-level nil))       ; Return to original Franz Lisp.
        (<- database ':ptime)
        (cond                              ; If flag is set show system status.
          ((<- database ':get-lisp-system-trace)
            (<- database ':internal-command-show '(show (symbol system)))))
        (cond
          ((<- database ':get-pling)
            (<- uinterf ':bell)))
        (<- uinterf ':main-prompt)
        (setq input-command (readloop))
        ;
        ; Send the command to the parser for further processing.
        ;
        (<- parser
          ':parse-command (wegnehmen '*CR* (lispify input-command)))
        ;
        ; Different help printouts depending on the user-state.
        ;
        (cond
          ((equal (<- database ':get-user-state) 'beginner)
            (<- parser ':parse-command '(?))
            (<- parser ':parse-command '(??))))

        ))))
```

Listing of the file util.l

```
; This file contains functions common to the whole system.

; ATOMCAR ------------------------------------------------------------------

; Ford planned to develop a nuclear powered car in the 50's.
; This car could have been named the ``Atom Car''.

(defun atomcar (sym)
  (getchar sym 1))

; Own list manipulation functions ------------------------------------------

; Works like mapcar, but all nil's are removed.
```

```
; (mapzap '(lambda (x) x) '(1 2 nil 3 4 nil)) => (1 2 3 4)

(def mapzap
  (macro (arg)
   (let ((al (symbol-list (gensym) (length (cddr arg)))))
    `(mapcan
      '(lambda ,al
        (let ((tmp (funcall ,(cadr arg) ,@al)))
          (and tmp (list tmp))))
      ,@(cddr arg)))))

;
; The YAPS system has redefined the function remove!!! Unbelievable!!!
;

(defun wegnehmen (expr list)
  (cond
    ((null list) nil)
    ((equal expr (car list)) (wegnehmen expr (cdr list)))
    ('Wahrheit (cons (car list) (wegnehmen expr (cdr list))))))

(defun remove-multiple (e)
  (cond
    ((null e) nil)
    ((member (car e) (cdr e)) (remove-multiple (cdr e)))
    (t (cons (car e) (remove-multiple (cdr e))))))

; Number-list return a list of integers up to 'num',
; i.e., the result looks like (1 2 3 ... num).

(defun number-list (num)
  (cond
    ((< num 0)
     nil)
    (t
     (append1 (number-list (1- num)) num))))

; (symbol-list 'x 3) ==> (x1 x2 x3)

(defun symbol-list (sym number)
  (mapcar
    '(lambda (x) (concat sym x))
     (cdr (number-list number))))

(defun list-to-string (lst)
  (let*
    ((str-lst
      (mapcar
        '(lambda (x)
           (cond
             ((atom x)
              (concat x " "))
             ((listp x)
              (concat "(" (list-to-string x) ") "))
             (t
              " ")))
        lst))
     (atm (concatl str-lst))
     (str (substring atm 1)))
    str))

; IHS stuff ------------------------------------------------------------------

(defmacro pc (com)
  (let ((x (cond
             ((atom com) (list com))
             (t com))))
    `(<- parser (quote :parse-command) (quote ,x))))

; Starts the system from the function (ihs).

(defun start-ihs ()
  (initialize)
  (sendnowait " " 1)
  (<- uinterf ':ihs-top-level))

; This function restarts (ihs), if you leave it temporarily
```

```
; with the command 'lisp'. (ihs) is restarted to the same state
; as when you left it.

(defun igs ()
  (<- uinterf ':ihs-top-level)
  'Ready)

; Fix to redirect output from LISP ----------------------------------------------

; Franz Lisp tricks.

(defun init-junk-io ()
  (cond
    ((not (boundp 'junk-poport))
      (setq junk-poport (outfile '/dev/null)))))

(defun suspendio ()
  (let ((tmp))
    (setq tmp junk-poport)
    (setq junk-poport poport)
    (setq poport tmp)))

(defun resumeio ()
  (let ((tmp))
    (setq tmp junk-poport)
    (setq junk-poport poport)
    (setq poport tmp)))

; COMMON LISP functions by Arzen ------------------------------------------------

(defun some (func list)
  (cond
   ((null list)
     nil)
   ((funcall func (car list))
     list)
   (t
     (some func (cdr list)))))

(defun every (func list)
  (cond
    ((null list)
      t)
    ((funcall func (car list))
      (every func (cdr list)))
    (t
      nil)))

; Rule manipulation -------------------------------------------------------------

(defun extract-lhs (body)
  (cond
    ((equal (car body) '-->) ())
    (t (cons (car body) (extract-lhs (cdr body))))))

(defun extract-rhs (body)
  (cond
    ((equal (car body) '-->) (cdr body))
    (t (extract-rhs (cdr body)))))

(defun int-to-ms (int)
  (let*
    ((x1 (quotient int 60.0))
     (x2 (/ (fix x1) 60))
     (x3 (difference x1 (* x2 60))))
    (list x2 x3)))

(defun time-sym (int)
  (let
    ((x (int-to-ms int)))
    (implode
     (append
      (cond ((< (car x) 10) '(|0|)))
      (explode (car x))
      '(:)
      (cond ((< (cadr x) 10) '(|0|)))
      (explode (cadr x))))))
```

```
(defun flat (body)
  (apply 'append
    (mapcar
      '(lambda (x)
         (cond ((atom x) (list x))
               (t (flat x))))
       body)))

(defun collect-vars (body)
  (remove-multiple
    (mapzap
      '(lambda (x)
         (cond ((and (atom x) (equal (atomcar x) '-)) x)))
      (flat (extract-lhs body)))))

(defun put-fact (fact full-name)
  (<- *yaps-db* 'fact
    (cond
      ((atom fact) (list full-name fact))
      (t (cons full-name fact)))))
```

Listing of the file verifier.l

```
; The functions in this file implements a simple script verifier,
; in a recursive descent manner. It only tests the syntax of the script.
; The semantcs is not checked. E.g. the script verifier vill not
; complain if the user tries to use a symbol as infile in an
; infile clause if it has not been created as an outfile or a globfile
; before. Much more work is required do do this.
; Not foolproof, but it catches many of the errors.

(defun logic-patom (obj val)
  (cond
    ((equal (car obj) 'command)
      (patom obj)
      (patom " ")
      (terpr))
    ((equal (car obj) 'assign)
      (patom obj)
      (patom " ")
      (terpr))
    ((equal (car obj) 'kscall)
      (patom obj)
      (patom " ")
      (terpr))
    ((equal (car obj) 'scriptmacro)
      (patom obj)
      (patom " ")
      (terpr))
    ((equal (car obj) 'repeat)
      (patom "(repeat ... )")
      (terpr))
    ((equal (car obj) 'repeat*)
      (patom "(repeat* ... )")
      (terpr))
    ((equal (car obj) 'or)
      (patom "(or ... )")
      (terpr))
    ((equal (car obj) 'all)
      (patom "(all ... )")
      (terpr))
    ((listp (car obj))
      (patom "List ...")
      (terpr))
    (t
      (patom obj)
      (terpr)))
   val)

; This predicate returns nil if the script is found to be erroneous.
; The code is not too complicated.

(defun is-this-a-script (script)
  (or
    (is-this-an-empty-script script)
    (and
```

```
      (listp script)
      (is-this-a-script-clause (car script))
      (is-this-a-script (cdr script)))))

(defun is-this-a-script-clause (clause)
  (and
    (listp clause)
    (or
      (is-this-a-command clause)
      (is-this-a-kscall clause)
      (is-this-an-assign clause)
      (is-this-an-or clause)
      (is-this-a-repeat clause)
      (is-this-a-repeat* clause)
      (is-this-an-all clause)
      (is-this-a-scriptmacro clause)
      (logic-patom clause nil))))

; Here follows one predicate for each of the possible clauses in a script.

(defun is-this-a-command (clause)
  (cond
    ((and
      (equal (car clause) 'command)
      (atom (cadr clause)))
      (mapcar
        '(lambda (x)
          (cond
            ((and
              x
              (listp x)
              (not
                (memq
                 (car x)
                 '(infile outfile globfile number numlist numlist1
                 symbol symlist symlist1))))
              (patom "This may be a bad command: ")
              (patom clause)
              (terpr))))
         clause)
      t)))

(defun is-this-a-kscall (clause)
  (cond
    ((and
      (equal (car clause) 'kscall)
      (cdr clause))
      (mapcar
        '(lambda (x)
          (cond
            ((and x (listp x) (not (equal (car x) 'parameter)))
              (patom "This may be a bad kscall: ")
              (patom clause)
              (terpr))))
        (cadr clause))
      t)))

(defun is-this-an-assign (clause)
  (and
    (equal (car clause) 'assign)
    (atom (cadr clause))
    (atom (caddr clause))))

(defun is-this-an-or (clause)
  (and
    (equal (car clause) 'or)
    (apply 'and (mapcar 'is-this-a-script (cdr clause)))))

(defun is-this-a-repeat (clause)
  (and
    (equal (car clause) 'repeat)
    (is-this-a-script (cadr clause))
    (null (cddr clause))))

(defun is-this-a-repeat* (clause)
  (and
    (equal (car clause) 'repeat*)
```

```
    (is-this-a-script (cadr clause))
    (null (cddr clause))))

(defun is-this-an-all (clause)
  (and
    (equal (car clause) 'all)
    (apply 'and (mapcar 'is-this-a-script (cdr clause)))))

(defun is-this-a-scriptmacro (clause)
  (and
    (equal (car clause) 'scriptmacro)
    (atom (cadr clause))))

(defun is-this-an-empty-script (clause)
  (null clause))
```

Listing of the file visual.l

```
; This is the support for VISUAL (= Tektronix) terminals used in (ihs).
; Only a very limited subset of the screen capabilities are used.

(defflavor visual-interface-flavor
  ())

; Enter graphics mode.

(defmethod (visual-interface-flavor :enter-graphics) ()
  (outchar 27)
  (outchar 58)
  (outchar 31))

; Enter alphanumerics mode.

(defmethod (visual-interface-flavor :exit-graphics) ()
  (outchar 24))

(defmethod (visual-interface-flavor :home) ()
  (outchar 27)
  (outchar 58)
  (outchar 25))

; The rest of the functions are used to convert the VT100 flavor (temporarily)
; to handle VISUAL too. This is done by calling the function prepare-for-yaps.
; Quick and very dirty.

(defun prepare-for-yaps ()
  (outchar 31)
  (clear-rectangle 0 645 1024 135)
  (clear-rectangle 0 400  200 245)
  (<- xx ':home))

(defun visual-set-cursor (x y)
  (draw-rectangle x y 0 0 "@"))

(setq xx (make-instance 'visual-interface-flavor))

(defun help (itm) (<- xx itm))

(defun draw-rectangle (x1 y1 x2 y2 arg)
  (outchar 27)
  (printitem arg)
  (outchar 27)
  (printitem "/")
  (printitem x1)
  (printitem ";")
  (printitem y1)
  (printitem ";")
  (printitem x2)
  (printitem ";")
  (printitem y2)
  (printitem ";")
  (printitem "y"))

(defun clear-rectangle (x1 y1 x2 y2)
  (outchar 27)
  (printitem "/1d")
  (outchar 27)
```

```
  (printitem "@")
  (outchar 27)
  (printitem "/")
  (printitem x1)
  (printitem ";")
  (printitem y1)
  (printitem ";")
  (printitem x2)
  (printitem ";")
  (printitem y2)
  (printitem ";")
  (printitem "y")
  (outchar 27)
  (printitem "/0d"))

(defun eng () (help ':enter-graphics))
(defun exg () (help ':exit-graphics))
(defun cs () (clear-rectangle 0 640 1024 140))
```

Listing of the file vt100.l

```
; This file contains the support for IO on a VT100-terminal.
; It is used as a mixin in the uinterf-flavor. Among other
; things it contains functions for handling prompting and menus.

(defflavor vt100-interface-flavor
  ())

; ------------------------------------------------------------------------------

(defmethod (vt100-interface-flavor :patom) (expr)
  (patom expr)
  (terpr))

; A more general (sic!) version of patom. It will handle atoms,
; strings, and lists of print stuff in an appropriate way.

(defmethod (vt100-interface-flavor :patton) (expr)
  (cond
    ((atom expr)
      (patom expr))
    (t
      (<- self ':write expr))))

(defmethod (vt100-interface-flavor :write) (expr)
  (cond
    (expr
      (<- self ':patton (car expr))
      (mapc '(lambda (x) (patom " ") (<- self ':patton x)) (cdr expr)))))

(defmethod (vt100-interface-flavor :writeln) (expr)
  (cond
    (expr
      (<- self ':patton (car expr))
      (mapc '(lambda (x) (patom " ") (<- self ':patton x)) (cdr expr))))
  (terpr))

; Here follows quite a few specialized printing functions. The
; idea is that the internal representation always should remain
; the same, an only these procedure would need to be changed
; when a new terminal or graphics device should be used.

(defmethod (vt100-interface-flavor :write-yaps-info) (info-list)

                      ; This line should be present when using a VISUAL
  (prepare-for-yaps)  ; and should be removed if used on a VT100

  (mapc
    '(lambda (x)
       (cond
         ((cdr x)
           (patom (<- database ':get-script-name (car x))) (patom ": ")
           (<- self ':writeln (cdr x)))))
     info-list)
  t)

(defmethod (vt100-interface-flavor :write-script-and-commands) (com-list)
```

```
  (mapc
    '(lambda (x)
       (cond
         ((cdr x)
           (patom (<- database ':get-script-name (car x))) (patom ": ")
           (patom (cadr x))
           (mapc '(lambda (y) (patom ", ") (patom y)) (cddr x))
           (patom " "))))
     com-list)
  t)

(defmethod (vt100-interface-flavor :write-next-commands) (com-list)
  (<- self ':write-script-and-commands com-list)
  (terpr)
  t)

(defmethod (vt100-interface-flavor :write-active-scripts) (com-list)
  (<- self ':write-script-and-commands (car com-list))
  (cond
    ((cdr com-list)
      (patom "[ ")
      (mapc
        '(lambda (x) (<- self ':write-script-and-commands x))
        (cdr com-list))
      (patom "]")))
  (terpr))

(defmethod (vt100-interface-flavor :write-dictionary-list) (lict)
  (cond
    (lict
      (<- self ':line 80)
      (mapc
        '(lambda (x)
           (<- self ':writeln (list "Help for" (car x)))
           (<- self ':writeln (cdr x))
          (<- self ':line 80))
       lict))))

(defmethod (vt100-interface-flavor :print-partial-command) (com)
  (cond
    ((null com)
      (terpr))
    ((atom (car com))
      (patom (car com))
      (patom " ")
      (<- self ':print-partial-command (cdr com)))
    ((equal (caddar com) 'idpac-default)
      (<- self ':print-partial-command (cdr com)))
    ((and (atom (cadar com))
          (equal (cadar com) '*unknown*))
      (patom "... ")
      (<- self ':print-partial-command (cdr com)))
    ((atom (cadar com))
      (patom (cadar com))
      (patom " ")
      (<- self ':print-partial-command (cdr com)))
    ((equal (cadadadar com) '*unknown*)
      (patom "... ")
      (<- self ':print-partial-command (cdr com)))
    (t
      (patom (cadadadar com))
      (patom " ")
      (<- self ':print-partial-command (cdr com)))))

; Routine for prompting the user for a left out argument in a command.
; The command, the number of the argument to ask for, and a possible
; default-value are given. The function returns the left out argument.
; Used by the Query module.

(defmethod (vt100-interface-flavor :prompt-command) (command nr default)
  (prog (ans questno spec)
    (setq questno 1)
    (setq spec (list (<- database ':get-clause-nr (car command) nr)))
  label
    (<- self ':print-partial-command command)
    (patom (<- database ':get-short-help-text (car command) nr))
    (patom "?")
```

```
    (cond (default (patom " (Default ") (patom default) (patom ")")))
    (patom " > ")
    (setq ans (tyipeek))
    (cond
      ((and default (equal ans 10))
        (tyi)
        (setq ans (list default)))
      ((equal ans 10)
        (tyi)
        (go label))
      (t
        (setq ans (list (read)))))
    (cond
      ((and (equal ans '(?)) (equal questno 1))
        (mapc
          '(lambda (x) (patom x) (terpr))
           (<- database ':get-long-help-text (car command) nr))
        (setq questno 2))
      ((and (equal ans '(?)) (equal questno 2))
        (patom "Sorry, I have already given all help I can.")
        (terpr)
        (setq questno 3))
      ((and (equal ans '(?)) (> questno 2))
        (patom "RTDM")
        (terpr))
      ((and
        (<- parser ':infile-spec? spec)
        (<- parser ':file? ans))
       (return (car ans)))
      ((and
        (<- parser ':outfile-spec? spec)
        (<- parser ':file? ans))
       (return (car ans)))
      ((and
        (<- parser ':number-spec? spec)
        (<- parser ':number? ans))
       (return (car ans)))
      ((and
        (<- parser ':numlist-spec? spec)
        (<- parser ':numlist? ans))
       (return (car ans)))
      ((and
        (<- parser ':numlist1-spec? spec)
        (<- parser ':numlist1? ans))
       (return (car ans)))
      ((and
        (<- parser ':symbol-spec? spec)
        (<- parser ':symbol? ans))
       (return (car ans)))
      ((and
        (<- parser ':symlist-spec? spec)
        (<- parser ':symlist? ans))
       (return (car ans)))
      ((and
        (<- parser ':symlist1-spec? spec)
        (<- parser ':symlist1? ans))
       (return (car ans)))
      (t
        (patom "Erroneous input.") (terpr)))
    (go label)))

; Prompts the user for an object of type 'type'.

(defmethod (vt100-interface-flavor :prompt) (text-list type)
  (prog (ans)
    loop
    (<- self ':write (append1 text-list "? > "))
    (setq ans (tyipeek))
    (cond ((equal ans 10) (tyi) (go loop)))
    (setq ans (list (read)))
    (cond
      ((and
        (equal type 'file)
        (<- parser ':file? ans))
        (return (car ans)))
      ((and
        (equal type 'number)
```

```
        (<- parser ':number? ans))
        (return (car ans)))
      ((and
        (equal type 'numlist)
        (<- parser ':numlist? ans))
        (return (car ans)))
      ((and
        (equal type 'numlist1)
        (<- parser ':numlist1? ans))
        (return (car ans)))
      ((and
        (equal type 'symbol)
        (<- parser ':symbol? ans))
        (return (car ans)))
      ((and
        (equal type 'symlist)
        (<- parser ':symlist? ans))
        (return (car ans)))
      ((and
        (equal type 'symlist1)
        (<- parser ':symlist1? ans))
        (return (car ans)))
      (t
        (patom "%IHS-F-STUINP, stupid input, expecting a ") (patom type)
        (terpr)))
    (go loop)))

; The (ihs) system prompt, (default "IHS> ").

(defmethod (vt100-interface-flavor :main-prompt) ()
  (patom (<- database ':get-prompt)))

(defmethod (vt100-interface-flavor :write-relatives) (list generation)
   (tab (1+ (* (<< database 'tab-number) generation)))
   (patom (car list))
   (cond
     ((cdr list)
       (mapc
          '(lambda (x)
            (<- self ':write-relatives x (1+ generation)))
          (cadr list)))
     (t
       (terpr))))

; ----------------------------------------------------------------------------

; Routines to give support for menus.
;
; menu-list should be defined as:
;   ([(<text-string>  <state-identifier> value [value...])...)
;
; internal-state is defined as:
;   ([(state-identifier current-value)...])
;
; For example in (ihs):
;
;   (setq internal-state '(
;     (trace-on f)
;     (fruit slime-mold)
;     (prompt   "IHS> ")
;     (pling f)
;     (yaps-debug f))
;
;   (setq menu-list '(
;     ("Tracing of the lisp system" lisp-system-trace t f)
;     ("Internal tracing"           trace-on         t f)
;     ("Tracing of rule system"     yaps-debug       t f)
;     ("Tracing of scripts"         trace-scripts-on t f)
;     ("Idpac fruit"                fruit            symbol))
;
; The method :menu returns a new list of internal-states, (not necessarily
; in the same order).

(defmethod (vt100-interface-flavor :menu) (menu-list internal-state)
  (let
    ((menu-length (length menu-list))
      (tmp internal-state))
```

```
(<- self ':erase 'screen)
(<- self ':acursor  1 1)
(mapc
  '(lambda (x)
     (printitem (car x))
     (printitem " ")
     (printitem (cadr (assoc (cadr x) tmp)))
     (<- self ':crlf))
   menu-list)
(terpr)
(patom "Exit from menu by typing control-c")
(terpr)
(patom "Numbers and symbols are terminated by CR.")
(terpr)
(ttioinit)
(<- self ':acursor 1 (length (explode (caar menu-list))))
(prog (in-symbol row)
  (setq row 1)
  loop
  (setq in-symbol (insymb))
  (cond
    ((equal in-symbol '|^C|)
      (ttreset)
      (<- self ':acursor (+ 3 menu-length) 1)
      (<- self ':crlf)
      (return tmp)))
  (cond
    ((equal in-symbol 'UP)
      (cond
        ((= row 1) (setq row menu-length))
        (t (setq row (- row 1)))))
    ((member in-symbol '(DOW CR))
      (cond
        ((= row menu-length) (setq row 1))
        (t (setq row (+ row 1)))))
    ((and
      (equal (caddar (nthcdr (- row 1) menu-list)) 'number)
      (number in-symbol))
      (<- self ':erase 'toeoln)
      (printitem in-symbol)
      (let ((x-symbol (get-number in-symbol)))
        (setq
          tmp
          (update-assoc-list
            tmp
            (cadar (nthcdr (- row 1) menu-list))
            x-symbol))))
    ((and
      (equal (caddar (nthcdr (- row 1) menu-list)) 'symbol)
      (alfa in-symbol))
      (<- self ':erase 'toeoln)
      (printitem in-symbol)
;     (let ((x-symbol (get-alfa-num-symbol in-symbol)))
      (let ((x-symbol (get-symbol in-symbol)))
        (setq
          tmp
          (update-assoc-list
            tmp
            (cadar (nthcdr (- row 1) menu-list))
            x-symbol))))
    ((member in-symbol (cddar (nthcdr (- row 1) menu-list)))
      (setq
        tmp
        (update-assoc-list
          tmp
          (cadar (nthcdr (- row 1) menu-list))
          in-symbol))
      (printitem in-symbol)
      (<- self ':erase 'toeoln))
    ((equal (caddar (nthcdr (- row 1) menu-list)) 'number)
      (printitem "Expecting a number."))
    ((equal (caddar (nthcdr (- row 1) menu-list)) 'symbol)
      (printitem "Expecting a symbol."))
    (t
      (printitem "   Try ")
      (mapcar 'printitem (cddar (nthcdr (- row 1) menu-list)))))
  (<- self ':acursor
```

```
        row
        (length (explode (caar (nthcdr (- row 1) menu-list)))))
      (go loop))))

; ------------------------------------------------------------------------

; Bell.

(defmethod (vt100-interface-flavor :bell) ()
  (outchar 7))

; Routines for VT100 screen control.

(defmethod (vt100-interface-flavor :line) (length)
  (outchar 27)
  (outchar 40)
  (outchar 48)
  (do ((i 0 (1+ i))) ((equal i length)) (outchar 113))
  (outchar 27)
  (outchar 40)
  (outchar 66))

; Scrolling region.

(defmethod (vt100-interface-flavor :scrolling-region) (x y)
  (outchar 27)
  (outchar 91)
  (printitem x)
  (outchar 59)
  (printitem y)
  (outchar 114))

; Cursor movements.

(defmethod (vt100-interface-flavor :cursor) (name)
  (cond
    ((equal name 'up)
      (outchar 27) (printitem "[A"))
    ((equal name 'down)
      (outchar 27) (printitem "[B"))
    ((equal name 'right)
      (outchar 27) (printitem "[C"))
    ((equal name 'left)
      (outchar 27) (printitem "[D"))))

; Cursor addressing.

(defmethod (vt100-interface-flavor :acursor) (row col)
  (outchar 27)
  (outchar 91)
  (printitem row)
  (outchar 59)
  (printitem col)
  (outchar 102))

(defmethod (vt100-interface-flavor :cursor-index) ()
  (outchar 27) (printitem "D"))

(defmethod (vt100-interface-flavor :cursor-reverse-index) ()
  (outchar 27) (printitem "M"))

(defmethod (vt100-interface-flavor :cursor-next-line) ()
  (outchar 27) (printitem "E"))

(defmethod (vt100-interface-flavor :save-cursor) ()
  (outchar 27) (printitem "7"))

(defmethod (vt100-interface-flavor restore-cursor) ()
  (outchar 27) (printitem "8"))

; Erase commands.

(defmethod (vt100-interface-flavor :erase) (name)
  (cond
    ((equal name 'toeoln)
      (outchar 27) (printitem "[K"))
    ((equal name 'fromboln)
```

```
      (outchar 27) (printitem "[1K"))
    ((equal name 'line)
      (outchar 27) (printitem "[2K"))
    ((equal name 'toeoscr)
      (outchar 27) (printitem "[J"))
    ((equal name 'fromboscr)
      (outchar 27) (printitem "[1J"))
    ((equal name 'screen)
      (outchar 27) (printitem "[2J"))))

; Video attribute commands.

(defmethod (vt100-interface-flavor :video-attribute) (name)
  (cond
    ((equal name 'off)
      (outchar 27) (printitem "[m"))
    ((equal name 'bold)
      (outchar 27) (printitem "[1m"))
    ((equal name 'blank)
      (outchar 27) (printitem "[2m"))
    ((equal name 'underline)
      (outchar 27) (printitem "[4m"))
    ((equal name 'blink)
      (outchar 27) (printitem "[5m"))
    ((equal name 'reverse)
      (outchar 27) (printitem "[7m"))))

; Miscellaneous.

(defmethod (vt100-interface-flavor :crlf) ()
  (outchar 13) (outchar 10))

; ----------------------------------------------------------------------------

; Functions for the menu system. Mainly pretty low-level IO.

(defun get-symbol (start-symbol)
  (prog (res inch)
    (setq res (list start-symbol))
    loop
    (setq inch (inchar))
    (cond ((equal inch 13) (return (implode res))))
    (cond
      ((or
        (in-interval inch 32 126))
        (outchar inch)
        (setq res (append1 res (ascii inch)))))
    (go loop)))

(defun get-alfa-num-symbol (start-symbol)
  (prog (res inch)
    (setq res (list start-symbol))
    loop
    (setq inch (inchar))
    (cond ((equal inch 13) (return (implode res))))
    (cond
      ((or
        (in-interval inch 48 57)
        (in-interval inch 65 90)
        (in-interval inch 97 122))
        (outchar inch)
        (setq res (append1 res (ascii inch)))))
    (go loop)))

(defun get-number (start-symbol)
  (prog (res inch)
    (setq res (- (car (aexploden start-symbol)) 48))
    loop
    (setq inch (inchar))
    (cond ((equal inch 13) (return res)))
    (cond
      ((in-interval inch 48 57)
        (outchar inch)
        (setq res (+ (* res 10) (- inch 48)))))
    (go loop)))

(defun in-interval (num lolim hilim)
```

```
  (and (> num (1- lolim)) (< num (1+ hilim))))

(defun alfa (symb)
  (let ((x (aexploden symb)))
    (cond
      ((and
        (equal (length x) 1)
        (or
          (in-interval (car x) 62 62)          ; >
          (in-interval (car x) 65 90)          ; A ... Z
          (in-interval (car x) 97 122)))))))   ; a ... z

(defun number (symb)
  (let ((x (aexploden symb)))
    (cond
      ((and
        (equal (length x) 1)
        (in-interval (car x) 48 57))))))

; Given an association-list, an atom, and a value, a new list is returned
; with the old value of the atom substituted by the new value.

(defun update-assoc-list (assoc-list var val)
  (cond
    ((null assoc-list) nil)
    ((equal (caar assoc-list) var)
      (cons (list var val) (cdr assoc-list)))
    (t (cons (car assoc-list) (update-assoc-list (cdr assoc-list) var val)))))
```

The Pascal Code

Listing of the file ttio.pas

```
module ttio(input,output);

const
  bufsiz = 128;

type
  string  = packed array [1..64] of char;
  $byte   = [byte] 0..255;
  $word   = [word] 0..65535;
  $quad   = [quad,unsafe]
            record
              l0 : unsigned;
              l1 : integer;
            end;
  terminftype = packed record
                  typ    : $byte;
                  cla    : $byte;
                  bufsiz : $word;
                  i1     : integer;
                  i2     : integer;
                end;
  buffertype  = record
                  inx   : integer;
                  outx  : integer;
                  peekx : integer;
                  nchar : integer;
                  buff  : packed array [1..bufsiz] of char;
                  char0 : char;
                end;
  itemtype = packed record
               buflen : $word; itemcode : $word;
               bufadr : integer;
               lenadr : integer;
             end;
  itemlist = packed record
               item1 : itemtype;
               term  : integer;
             end;
  iosbtype = packed record
               condval : $word;
```

```
              trfcnt  : $word;
              devinf  : integer;
            end;
  exitblocktype = packed record
                    forwlink   : integer;
                    exitaddr   : integer;
                    mbz        : integer;
                    condvaladr : integer;
                  end;

%include 'USE:[PERP.REAL.VAX]MBXINC.PAS/NOLIST'

;
; This is the declarations of VAX/VMS system services and some constants,
; can also be found in SYS$LIBRARY:STARLET.PAS
;

var
  ttchan   : [volatile] $word;
  ch       : [volatile] char;
  buffer   : [volatile] buffertype;
  newterm  : [volatile] terminftype;
  oldterm  : [volatile] terminftype;
  item     : [volatile] itemlist;
  devnam   : [volatile] string;
  devlen   : [volatile] integer;
  iosb     : [volatile] iosbtype;
  status   : [volatile] integer;
  exblk    : [volatile] exitblocktype;
  exitstat : [volatile] integer;
{------------------------------------------------------------------------------}
[asynchronous] function inclusive_or(i1,i2 : integer): integer;

type
  conv = record
           case integer of
             1 : (i : integer);
             2 : (b : packed array [0..31] of boolean);
         end;
var
  tmp1,tmp2,out : conv;
  i : integer;

begin
  tmp1.i := i1;
  tmp2.i := i2;
  for i := 0 to 31 do
    out.b[i] := tmp1.b[i] or tmp2.b[i];
  inclusive_or := out.i;
end;
{------------------------------------------------------------------------------}
[asynchronous, unbound] procedure ast0; forward;
{------------------------------------------------------------------------------}
[asynchronous, unbound] procedure enableread;
begin
  sys$qio(chan   := ttchan,
          func   := IO$_READVBLK,
          iosb   := iosb,
          astadr := ast0,
          p1     := buffer.char0,
          p2     := 1);
end;
{------------------------------------------------------------------------------}
{[asynchronous, unbound]} procedure ast0;
begin
  if (iosb.condval = SS$_ABORT) or (iosb.condval = SS$_CANCEL) then
  begin end
  else
  begin
    if iosb.condval <> SS$_NORMAL then
      buffer.char0 := chr(255);
    with buffer do
    begin
      buff[inx] := char0;
      if inx = bufsiz then
        inx := 1
      else
```

```
        inx := inx + 1;
      nchar := nchar + 1;
      if nchar = bufsiz then
        sys$setast(0);
    end;
    sys$setef(1);
    enableread;
  end;
end;
{-------------------------------------------------------------------------------}
[global] procedure ttreset;
begin
  sys$canexh;
  sys$cancel(chan := ttchan);
  sys$setast(0);
  sys$qiow(chan := ttchan,
           func := IO$_SETMODE,
           p1   := oldterm,
           p2   := 12);
  sys$dassgn(chan := ttchan);
end;
{-------------------------------------------------------------------------------}
[global] function inchar : char;
begin
  sys$waitfr(efn := 1);
  sys$setast(0);
  with buffer do
  begin
    inchar := buff[outx];
    if outx = bufsiz then
      outx := 1
    else
      outx := outx + 1;
    peekx := outx;
    nchar := nchar - 1;
    if nchar = 0 then
      sys$clref(efn := 1);
  end;
  sys$setast(1);
end;
{-------------------------------------------------------------------------------}
[global] function peekchar : char;
begin
  with buffer do
  begin
    if nchar = 0 then
      peekchar := chr(128)
    else
    begin
      peekchar := buff[peekx];
      if peekx = bufsiz then
        peekx := 1
      else
        peekx := peekx + 1;
    end;
  end;
end;
{-------------------------------------------------------------------------------
[global] procedure outchar(ch : char);
begin
  sys$qiow(chan := ttchan,
           efn  := 2,
           func := IO$_WRITEVBLK,
           p1   := ch,
           p2   := 1);
end;
 ------------------------------------------------------------------------------
[global] procedure outstring(var st : char;
                             nr : integer);
begin
  sys$qiow(chan := ttchan,
           efn  := 2,
           func := IO$_WRITEVBLK,
           p1   := st,
           p2   := nr);
end;
 ------------------------------------------------------------------------------}
```

```
[global] procedure ttioinit;

var
  tmp : integer;

begin
  with buffer do
  begin
    inx   := 1;
    outx  := 1;
    peekx := 1;
    nchar := 0;
  end;

  item.item1.buflen   := 64;
  item.item1.itemcode := LNM$_STRING;
  item.item1.bufadr   := iaddress(devnam);
  item.item1.lenadr   := iaddress(devlen);
  item.term           := 0;

  status := sys$trnlnm(tabnam := 'LNM$PROCESS_TABLE',
                       lognam := 'TT',
                       itmlst := item);

  sys$assign(devnam := devnam,
             chan   := ttchan);

  for tmp := devlen + 1 to 64 do
    devnam[tmp] := chr(0);

  sys$qiow(chan := ttchan,
           func := IO$_SENSEMODE,
           p1 := oldterm,
           p2 := 12);
  newterm := oldterm;

  tmp := newterm.i1;
  tmp := inclusive_or(tmp, TT$M_MBXDSABL);
  tmp := inclusive_or(tmp, TT$M_NOECHO);
  tmp := inclusive_or(tmp, TT$M_TTSYNC);
  newterm.i1 := tmp;

  tmp := newterm.i2;
  tmp := inclusive_or(tmp, TT2$M_PASTHRU);
  tmp := inclusive_or(tmp, TT2$M_BRDCSTMBX);
  newterm.i2 := tmp;

  sys$qiow(chan := ttchan,
           func := IO$_SETMODE,
           p1   := newterm,
           p2   := 12);

  exblk.forwlink := 0;
  exblk.exitaddr := iaddress(ttreset);
  exblk.mbz := 1;
  exblk.condvaladr := iaddress(exitstat);
  sys$dclexh(desblk := exblk);

  sys$clref(1);
  enableread;
  sys$setast(1);
end;
{------------------------------------------------------------------------------}
[global] procedure showchar(ch : char);
begin
  if ch in [' '..'~'] then
    write(ch)
  else if ch = chr(127) then
  begin
    write('D');
    write('E');
    write('L');
  end
  else
  begin
    write('^');
    write(chr(ord(ch)+64));
```

```
  end;
end;
{-------------------------------------------------------------------------------}
[global] procedure dumpbuffer;

var
  i : integer;

begin
  with buffer do
  begin
    write(' inx   '); write(inx:2);
    write(' outx  '); write(outx:2);
    write(' peekx '); write(peekx:2);
    write(' nchar '); write(nchar:2);
    write(' char0 '); showchar(char0);
    writeln;
    writeln('-');
    for i := 1 to bufsiz do
      showchar(buff[i]);
    writeln;
    writeln('-');
    writeln;
  end;
end;

end.
```

The C Code

Listing of the file snd.c

```
/* These routines are used for the mailbox communication between the Lisp    */
/* and Idpac. Sendwait sends a character string to a mailbox and waits       */
/* until the text has been read. Sendnowait also sends a character string    */
/* to a mailbox, but immediately continues execution. Initialize creates     */
/* the mailbox. PERP is the login name of one of the authors.                */

#include <stdio.h>

struct stdescr {
  int i;
  char *c;
};

char   txtnam[9]   = "PERP_BOX";
struct stdescr txtbox;
int    txtchan;

char   message [80];
int    msglen;
int    io_writevblk = 48;

/*-------------------------------------------------------------------------*/

initialize ()
{
  txtbox.c = txtnam;
  txtbox.i = 8;
  sys$crembx (0, &txtchan,  0, 0, 0, 0, &txtbox);
}

/*-------------------------------------------------------------------------*/

sendwait (message, msglen)
char *message;
int *msglen;
{
  sys$qiow (0, txtchan, io_writevblk, 0, 0, 0, message, *msglen, 0, 0, 0, 0);
}

/*-------------------------------------------------------------------------*/
```

```
sendnowait (message, msglen)
char *message;
int *msglen;
{
  sys$qio (0, txtchan, io_writevblk, 0, 0, 0, message, *msglen, 0, 0, 0, 0);
}
```

Listing of the file ttio.c

```
/* These two procedures are used by Franz Lisp to write a character resp.    */
/* a character string to the standard output.                                */

#include <sys/ioctl.h>
#include <stdio.h>

/*---------------------------------------------------------------------------*/

outchar (ch)
char *ch;
{
  write (1, ch, 1);
}

/*---------------------------------------------------------------------------*/

outstring (str, len)
char *str;
int *len;
{
  write (1, str, *len);
}
```

The DCL Code

Listing of the file idpac.com

```
$ SPAWN/NOWAIT @PACSTART IDPAC REG:[REGLER.EXE] IDPAC
```

Listing of the file pacstart.com

```
$ old_verify='f$verify("NO")'
$ if "''trace_on'" then set verify
$ on control_y then goto interrupt
$ on error then goto interrupt
$
$ ! Common start-up procedure for the program packages:
$
$ ! p1  - Program package name, e.g. IDPAC
$ ! p2  - .exe file area
$ ! p3  - Name of standard data area:
$ !       textliblev2 = reg:[regler.machelp.'p3'] (default: p1)
$ ! p4  - Program version (default: highest)
$
$ ! a/ Enable ctrl/y catching
$ ! b/ Make the terminal react to form feeds
$ ! c/ Rename the print file to 'p1'.log
$ ! d/ Type package specific as well as general notice file
$ ! e/ Run the package
$ ! f/ Remind the user that a print file is available, if any
$
$ ! Tomas Schonthal 1981-07-16
$
$ if "''p3'".eqs."" then p3:="''p1'"
$ if "''p2'".eqs."REG:[REGLER.EXE]" then type REG:[REGLER.MACHELP.'p3']'p3'.TXT
$ p4=p4-";"
$ where_am_I=f$parse("[]")-":"-".;"
$ job_mode=f$mode()
$ type REG:[REGLER.MACHELP.PROGPACS]PROGPACS.TXT
$ if "''job_mode'".nes."BATCH" then set terminal/form
$ assign/user_mode 'p1'.log for106
$ assign/user_mode 'p1'.spy for007
```

```
$ if "''f$logical("TEXTLIBLEV1")'".eqs."" then assign/user_mode [] textliblev1
$ assign/user_mode REG:[REGLER.MACHELP.'p3'] textliblev2
$ assign/user_mode REG:[REGLER.MACHELP.PROGPACS] textliblev3
$ assign/user_mode perp_box sys$input  ! **** CHANGE HERE ****
$ assign/user_mode sys$output dis_1
$ assign/user_mode REG:[REGLER.SETUP]pacsid.key pac_sid
$ start_time=f$cvtime(f$time())
$ image=f$search(p2+p1+".exe;"+p4)
$ run/nodebug 'image'
$ pac_status=$status
$ if pac_status then goto log_check
$
$ interrupt: pac_status=$status
$ set message/notext/noidentification/nofacility/noseverity
$ delete hcfil.*;*, *.at;*, *.dt;*, *.pt;*, *.tt;*
$ set message/text/identification/facility/severity
$
$ log_check: open/read/error=spy_check 1 'p1'.log
$ close 1
$ if f$cvtime(f$file_attributes("''p1'.log","rdt")).lts.start_time then -
  goto spy_check
$
$ write sys$output -
  "To print & delete your log, type: PRINT/HEAD/DELETE ''p1'.LOG"
$
$ spy_check: open/read/error=done 1 'p1'.spy
$ close 1
$ if f$cvtime(f$file_attributes("''p1'.spy","rdt")).lts.start_time then -
  goto done
$
$ open/append 1 'p1'.spy
$ write 1 ""
$ write 1 " "" ----------------------------------------------------------"
$ write 1 " "" Cpu:            ''f$getsyi("cpu")'"
$ write 1 " "" Directory:      ''f$logical("sys$disk")''f$directory()'"
$ write 1 " "" Date:           ''f$time()'"
$ write 1 " "" Image:          ''image'"
$ write 1 " "" Mode:           ''job_mode'"
$ write 1 " "" PID:            ''f$getjpi("","pid")'"
$ write 1 " "" Priority:       ''f$getjpi("","pri")'"
$ write 1 " "" Process:        ''f$process()'"
$ write 1 " "" SID:            ''f$fao("!XL",f$getsyi("sid"))'"
$ write 1 " "" Site:           ''f$logical("sys$announce")'"
$ write 1 " "" Pac_status:     ''f$message(pac_status)'"
$ write 1 " "" Terminal:       ''f$logical("sys$output")'"
$ write 1 " "" Terminal type:  ''f$logical("terminal$type")'"
$ write 1 " "" Textliblev1:    ''f$logical("textliblev1")'"
$ write 1 " "" UIC:            ''f$getjpi("","uic")'"
$ write 1 " "" Username:       ''f$getjpi("","username")'"
$ write 1 " "" VMS             ''f$getsyi("version")'"
$ write 1 " "" ----------------------------------------------------------"
$ close 1
$ write sys$output -
  "Keyboard input captured in file:                    ''p1'.SPY"
$
$ done: set noverify
$ if old_verify then set verify
```

The Idpac Macro Code

Listing of the file ccoff.t

```
MACRO ccoff cc < data1 data2  nlag

LET ns = nlag * 2
LET nn = ns + 1
LET nq = nlag + 1

INSI zqp nn
  LET amp. = 2
  LET ifp. = nq
  ZERO
  STEP
```

```
  LET amp. = 1
  LET ifp. = 1
  RAMP
X

SCLOP zqp (2) < zqp (2) - 1.0
SCLOP zqp (4) < zqp (1) + 1.0
SCLOP zqp (3) < zqp (3) - nlag

CCOF cc < data1 data2 nlag
PLOT (nm) (nn) zqp (3) < cc zqp (1 4) (hp) zqp (2)

DELET zqp

END
```

Listing of the file coh.t

```
MACRO coh cohf  < insi outsi nlag

ASPEC xqpiff   < insi  nlag
ASPEC xqpuff   < outsi nlag
CSPEC xqpaff   < insi  outsi nlag
MOVE  plink    < xqpaff
SCLOP plink(3) < plink(3) * -1.0
FROP  plink    < xqpaff   * plink
FROP  plank    < xqpiff   * xqpuff
FROP  cohf     < plink    / plank
MOVE  cohf(3)  < cohf(2)

BODE (p) cohf

DELET xqpiff
DELET xqpaff
DELET xqpuff
DELET plink
DELET plank

END
```

Listing of the file loss.t

```
MACRO loss file

FREE rows.ext
LET  rows.ext = 0
FREE var.ext
LET  var.ext = 0.0

FHEAD file /ext
 X
TURN TEXT OFF
STAT file ext
TURN TEXT ON

LET loss = 0.0
LET loss = var.ext * rows.ext

WRITE 'The value of the loss function for ' file  ' is ' loss

END
```

Listing of the file mlid.t

```
MACRO mlid syst  < insi outsi no

FTEST (t) syst
IF ftest. EQ 0 GOTO lab1
DELET (t) syst
LABEL lab1

FTEST xqpiff
IF ftest. EQ 0 GOTO lab2
DELET xqpiff
LABEL lab2
```

```
MOVE  xqpiff (1) < insi
MOVE  xqpiff (2) < outsi

"LET INIML. = 1
ML (SC) syst     < xqpiff no ext
  SAVE COMAT
X
"LET INIML. = 0

END
```

Listing of the file mlidsc.t

```
MACRO mlid syst  < insi outsi no

FTEST (t) syst
IF ftest. EQ 0 GOTO lab1
DELET (t) syst
LABEL lab1

FTEST xqpiff
IF ftest. EQ 0 GOTO lab2
DELET xqpiff
LABEL lab2

MOVE  xqpiff (1) < insi
MOVE  xqpiff (2) < outsi

ML (SC) syst     < xqpiff no ext
  SUSPEND
  SAVE COMAT
X

END
```

Listing of the file prewhite.t

```
MACRO PREWHITE PREWI PREWO < INPUT OUTPUT ; N
"
"MACRO FOR PREWHITENING OF INPUT AND OUTPUT SIGNALS BEFORE
"CROSSCORRELATION ETC.
"
"INPUT  NAME OF FILE WHERE THE INPUT SIGNAL IS
"OUTPUT NAME OF FILE WHERE THE OUTPUT SIGNAL IS
"N      ORDER OF AUTOREGRESSIVE FILTER USED FOR THE PRE-
"       WHITENING. THIS PARAMETER IS OPTIONAL. DEFAULT 10
"PREWI
"PREWO  NAME OF FILES WHERE THE PREWHITENED SIGNALS ARE PLACED.
"       INPUT SIGNAL IN PREWI, OUTPUT SIGNAL IN PREWO
"
DEFAULT N=10
"
TREND INPUTT<INPUT 0
TREND OUTPUTT<OUTPUT 0
"
LET STDEV.ZQPREW=1.
"
"TURN TEXT OFF
"TURN GRAPH OFF
"
STRUC ZQSTRF
NA MAX N
NU MAX 0
X
SQR ZQRMAT<INPUTT ZQSTRF
LS ZQLSMOD<ZQSTRF
"
RESID PREWI<ZQLSMOD INPUTT
KILL
"
STAT PREWI ZQPREW
SCLOP PREWI<PREWI/STDEV.ZQPREW
"
RESID PREWO<ZQLSMOD OUTPUTT
KILL
"
```

```
SCLOP PREWO<PREWO/STDEV.ZQPREW
"
DELET (T) ZQLSMOD INPUTT OUTPUTT
DELET (T) ZQSTRF
"
"TURN TEXT ON
"TURN GRAPH ON
"
END
```

Listing of the file randstep.t

```
MACRO RANDSTEP Y<MOD U NL ; NPLX
"
"MACRO FOR MONTE CARLO SIMULATION OF TIME RESPONSES IN ORDER TO
"TEST THE UNCERTAINTY OF AN ESTIMATED MODEL.
"
"MOD   SYSTEM FILE WHERE THE ESTIMATED MODEL PARAMETERS AND
"      THEIR ESTIMATED COVARIANCE MATRIX ARE
"U     FILE WITH INPUT SIGNAL FOR THE SIMULATIONS, E.G. A STEP SIGNAL.
"      NOTICE THAT THE SAMPLING INTERVAL OF THIS FILE MUST COINCIDE
"      WITH THE SAMPLING INTERVAL IN THE SYSTEM FILE MOD.
"NL    THE NUMBER OF SIMULATED TIME RESPONSES REQUESTED
"NPLX  OPTIONAL ARGUMENT (DEFAULT VALUE = THE GLOBAL VARIABLE
"      NPLX.) DETERMINING NUMBER OF POINTS PLOTTED PER PAGE,
"      NORMALLY EQUAL TO THE NUMBER OF POINTS IN THE DATA FILE U
"Y     DATA FILE CONTAINING THE SIMULATED TIME RESPONSES AS
"      COLUMNS
"
DEFAULT NPLX=NPLX.
"
FOR I=1 TO NL
RANPA P<MOD
DETER Y(I)<P U
DELET (T) P
NEXT I
"
PLOT (NPLX) Y
"
END
```

Listing of the file randtf.t

```
MACRO randtf fil < spc n

FTEST (t) zzypt
IF ftest. EQ 0 GOTO lab1
DELET (t) zzypt
LABEL lab1

FOR i = 1 TO n
  RANPA zzypt < spc
  SPTRF fil(i) < zzypt b / a
  DELET (t) zzypt
NEXT i

BODE fil

END
```

Listing of the file residu.t

```
MACRO residu res < syst insi outsi
FTEST xqpiff
IF ftest. EQ 0 GOTO lab
DELET xqpiff
LABEL lab
MOVE xqpiff (1) < insi
MOVE xqpiff (2) < outsi
RESID res < syst xqpiff
END
```

Listing of the file slid.t

```
MACRO slid o1 o2 < i1 i2 nr
```

```
MOVE zzzz(1) < i1
MOVE zzzz(2) < i2
SLIDE zzzzz < zzzz nr 0
MOVE o1 < zzzzz(1)
MOVE o2 < zzzzz(2)

DELET zzzz
DELET zzzzz

END
```