LUND UNIVERSITY

An Expert System Interface for Idpac

Larsson, Jan Eric; Persson, Per

1987

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

Link to publication

Citation for published version (APA):
Larsson, J. E., & Persson, P. (1987). *An Expert System Interface for Idpac*. [Licentiate Thesis, Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
2

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

CODEN: LUTFD2/(TFRT-3184)/1-65/(1987)

An Expert System Interface For Idpac

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
Licentiate Thesis

Date of issue
January 1987

Document Number
CODEN: LUTFD2/(TFRT-3184)/1-65/(1987)

Author(s)
Jan Eric Larsson
Per Persson

Supervisor
Karl Johan Åström

Sponsoring organisation

Title and subtitle
An Expert System Interface for Idpac

Abstract

This thesis describes an expert system interface for system identification, using the interactive identification program Idpac. The interface works as an intelligent help system, using the command spy strategy. It contains a multitude of help system ideas. The concept of scripts is introduced as a data structure used to describe the procedural part of the knowledge in the interface. Production rules are used to represent diagnostic knowledge. The implementation of the system is described. A small knowledge base of scripts and rules has been developed and an example run is shown. Finally, there is an outline of further developments.

Key words
Expert Systems, Help Systems, Man-Machine Interfaces, Scripts, System Identification

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
65

Recipient's notes

Security classification

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

An Expert System Interface for Idpac

The Authors.

An Expert System Interface for Idpac

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

© 1987 by Jan Eric Larsson and Per Persson. All rights reserved.

Published in Lund 1987.

Printed in Sweden.

Contents

Preface

The idea of the command spy was born in 1983. The command spy was a sneaky little program that hid behind the terminal and eavesdropped on you. The smart little guy figured out what you were trying to do even before you knew it yourself. Every time you goofed up and deleted your most precious file or something like that, the command spy secretly laughed at you. But, alas, he was too shy to ever say anything.

That's why we decided to give the command spy a voice and turn him into an intelligent help system. Actually, we did not plan to build a working system, but the famous composer Franz Lisp let us use one of his cars, and off we went. We had some trouble with a couple of hoodlums named Flavors and YAPS, but we managed to escape in the end, and the command spy was finally implemented.

Of course we tested the command spy ourselves. This is what it told us. "How about a vacation? The next sensible command is `logout`."

This is a jointly authored thesis. Both of us have written and rewritten substantial parts of every chapter and thus we are both responsible for every part whether produced in compromise or in total agreement.

We would like to thank our supervisor Karl Johan Åström who suggested the problem and proposed using scripts. We are grateful to our colleagues Sven Erik Mattsson, Lars Nielsen, and Karl-Erik Årzén for useful discussions.

The project is part of the Computer-Aided Control Engineering Project at the Department of Automatic Control, Lund Institute of Technology, supported by STU, the National Swedish Board for Technical Development, under contract No. 85–3042.

J.E.L.

P.P.

1

Introduction

Die Welt ist alles was der Fall ist.*

This thesis describes a help system based on expert system techniques. It works as an interface to Idpac, an interactive program for system identification. The main features of the help system are that it is non-invasive, that it keeps track of what the user has done and that it has procedural and diagnostic knowledge about system identification. This is made possible by using the expert system as an interface to Idpac and enables it to work as an intelligent help system. This chapter formulates the problems and outlines a solution.

Presentation of the Problem

Knowledge based computer programs, expert systems, are rapidly being developed and will probably be quite common in the near future. When the techniques used in these programs grow more reliable and become better known, it will be obvious for a CAD program to use them. Several problems must be solved in order to incorporate expert knowledge in a program package.

- An inexperienced user often has a general idea of what he wants to do, but does not know exactly how to do it. An intelligent help system should be able to guide such a user from general ideas to specific commands. This is not taken care of by an ordinary "list all available commands" help system. Therefore a *goal related* help facility should be available.
- There will inevitably be times when users do things that the help system does not understand. This might e.g., be an experienced user taking short cuts or doing completely new things. At these times the help system will not be able to work properly. Also, sometimes a user does not wish to have any help. In such cases it should not be forced

* The quotations in the chapter heads are the seven main statements of Ludwig Wittgenstein's Tractatus Logico-Philosophicus.

upon him. For these reasons the help system should be totally non-invasive, i.e., only come into action on the user's request. As long as one does not issue any help command, one should not notice that the help system is there. One solution to this is the *command spy* concept.

- Idpac uses a flexible and powerful command dialog. This way of communication should be kept when the expert system is added, instead of using a question and answer dialog. To do this a flexible and easy-to-use front-end must be developed and interfaced to an expert system framework. The front end should also be very close to the Idpac command language.
- A large part of the knowledge of an expert interface for running Idpac concerns sequences. The user's basic actions will be to type sequences of commands. On a more abstract level, the user will solve sequences of problems, where each problem will demand several commands to be performed. Sequences may be represented using production rules to implement a state machine, but this will become cumbersome when the number of states grow. Therefore we have introduced the concept of *scripts*, as a data structure for describing sequences.
- The help system needs not only procedural knowledge about Idpac, but also knowledge of system identification in general. This is required for it to be able to diagnose problems, estimate the validity of results and propose further tests. This function can be taken care of by an ordinary production rule system.
- Part of the practical difficulties of running Idpac is keeping track of details, such as remembering file names, what operations have been performed on what data, values of parameters, and so on. The help system should aid the user in this and keep the important detail information in a database. Data might be retrieved both through special commands to the interface and by providing intelligent defaults for command arguments.
- One important task for the help system is to transfer knowledge from the knowledge database to the user. Several facilities for this must be provided. The production system should give help to interpret results and suggest what to do next. The system should prompt for parameters in commands and give thorough explanations of them in the process. There should also be an on-line dictionary to explain the vocabulary of system identification.

An intelligent help system has been designed according to these design goals. Its different parts are a command parser with its command grammar, a command matcher with its script database, containing both script data and production rules, a query module, a file system, an on-line dictionary, and interfaces to the user and Idpac.

Rationale for the Project

In order for a solution involving expert system techniques to be applicable there is a number of requirements that should be fulfilled, Brownston *et al* [1985] or Stefik *et al* [1982 a]. Before we go on to present the solution given in the thesis, let us look through the checklist in order to verify that the project is a reasonable undertaking.

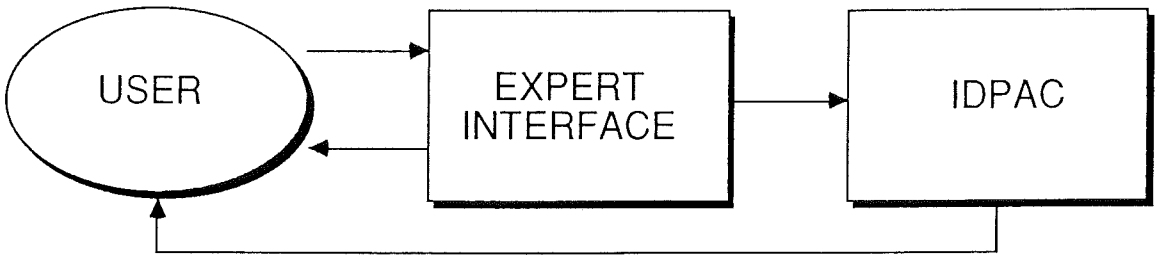
- The problem area should be well known, and experts must be available. Process identification is a well known problem. Several people at our department have a thorough experience of process identification and in using Idpac.
- The problem area should be easy to isolate from the rest of the world. System identification is a well defined problem. There is a limited amount of input types, methods to use and results one may want to reach.
- A problem to be solved with expert system techniques must not be to simple. The problem of system identification is far from trivial. Many projects involving expert systems have addressed very simple tasks, and it is important to move up over the "embarassment level."
- A project must be reasonably limited if a solution is to be at all possible. We believe that this thesis shows that the project indeed is reasonable, but there is no way of knowing this until the knowledge database is completely developed. But we feel sure that it will be possible to find natural limits for the project within the area of system identification.

The program Idpac is in itself well fitted to be used with another program. Its command language contains no complex features and allows no compounding of commands. This means that the parsing problem is not very difficult. Idpac has a limited number of commands and is only used for a few methods of system identification. Therefore it quite naturally gives a limited context for the expertise.

The conclusion of this is that an expert system solution is possible.

The Knowledge Based Command Spy

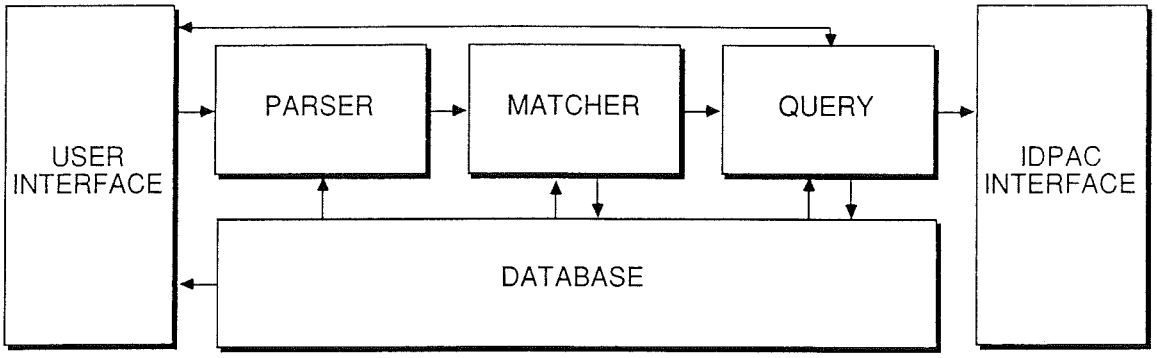
To sum up the problems stated in the previous sections, let us formulate them in one simple question. How does one combine a CAD program with an expert system while keeping the good features of both? The solution proposed in this thesis is the *command spy* concept.



The expert system used as an interface to Idpac.

The expert system is used as an interface to the CAD program. It is either placed before the command decoder of the CAD program or built into the outermost level of it. In our implementation we choosed to have it run separately from Idpac. In this way a command will be parsed both in the interface and in Idpac, but the alternative, i.e., to start digging

in the code of Idpac, a 37 000 line Fortran program, seemed to be much worse. The system contains the following features.



Layout of the system.

The expert system interface reads the commands and passes them on to Idpac, keeping track of what is happening, *without* asking any questions to the user. In fact, this whole process goes on in secret, and the user does not even have to know that the help system is there.

The expert interface will only give help if the user asks for it. It may start a short question and answer dialog with the user, but only if he initiates it. There is one single moment when the expert interface comes into action on its own. This is when the user types a command that does not match any of the scripts. The command spy will try to get back into tracing the user following a few different strategies, but if these strategies fail the help system prints the message "No match," indicating that it can no longer give any help. The user is still free to go on, without the help system. The whole idea of non-invasiveness is very important for the design of an intelligent help system.

The command spy uses *scripts* in order to understand what the user is doing. Scripts represent command sequences and events associated with them and each script describes one of the methods that Idpac is normally used for. In this way the interface is able to use procedural knowledge, reasoning about methods for achieving desired results, i.e., work as a goal related help system. By matching a set of scripts against the actual command history, the help system tries to guess what the user wants to do. From this it can give advice on how to continue towards the assumed goal. The guessing without asking any questions is part of the non-invasiveness philosophy. If the user wants to follow a script from the beginning, help is available that tells him what command to start with. Because of this the guessing strategy works very well, and there is no reason to tell the system explicitly what the user wants to do.

The expert interface keeps track of a user state. This can currently be either "expert" or "beginner," but there could be several states if needed. The system starts in expert mode, and stays in it until the user requests beginner mode. In the expert mode the system works as described above, and is not allowed to take any actions on its own. This means that a lot of diagnostic reasoning cannot be done, since the system lacks information. If the user needs more active help, he may enter the beginner mode, which is usually done by the command `think`. Help texts for every step taken are given automatically, and the rule system is allowed to actively ask questions. This usually means that the expert interface starts a short question and answer dialog in order to catch up with what has happened so far during the session. After that, the user is in a dialog with the help system, with guidance for every step, a few

questions asked and some diagnoses given every now and then. The user always has the opportunity of setting the user state as he wishes.

The command spy is transparent in the sense that it always lets the user fall back on plain Idpac. This is important for the non-invasiveness. The intelligent help system contains some features that make it easier to use Idpac. This includes a query mode for reading parameters, Rimvall and Bomholt [1985], short forms of commands, a file system database, and an on-line dictionary. These facilities work independently of the script matching and the user never loose them because of doing things that the command spy does not understand. The production rules can also be used to implement automatic documentation, by writing script based information commands to a text file.

Literature and Other Projects

Other works have been done in this and related areas. System identification is described in Cox [1958], Eykhoff [1974, 1981], Fedorov [1972], Ljung and Söderström [1985], Åström and Eykhoff [1971] and Åström [1980]. Idpac is built with the interaction module Intrac, a framework that provides an interactive environment for numerical Fortran routines. Idpac and Intrac were developed at the Department of Automatic Control, Lund Institute of Technology, Wieslander [1979 a, b, c, 1980] and Åström [1983 a, b, 1985]. More about Idpac can be found in Gustavsson and Nilsson [1979] and Gustavsson [1979].

For readings about expert systems in general see Brownston *et al* [1985], Harmon and King [1985], Hayes-Roth, Waterman and Lenat [1983], Stefik *et al* [1982] and Waterman [1986]. For interesting projects in expert systems, see Allen [1983], Forgy [1981], Nii and Aiello [1979] and van Melle *et al* [1981]. For projects in design of control systems see James *et al* [1985], Taylor *et al* [1984 a, b] and Birdwell *et al* [1984, 1985, 1986]. There are also more closely related works, e.g., Gale and Pregibon, [1982, 1983], describing an expert interface for a statistical program package, smaller than, but similar to Idpac, and Weiss *et al* [1982] and Welin [1986], giving general overviews on expert interfaces.

Two more recent projects containing lots of ideas close to the project described in this thesis are the Unix Consultant and the Knowledge Based Emacs. The Unix Consultant is an intelligent help system for Unix. It reads questions in natural language and uses scripts to tell the user how to perform different tasks, Wilensky *et al* [1986]. Knowledge Based Emacs is an AI system built into an editor. This allows the user to work on a higher level of programming than the ordinary Lisp or Ada level, Waters [1985 a, b]. The project described in this thesis started with a master thesis by one of the authors, Larsson [1984], Larsson and Åström [1985 a]. A first system was developed during 1985. Readings on that system is found in Larsson and Persson [1986 b]. The system described in this thesis is further documented and the source code is given in Larsson and Persson [1987 b]. The full knowledge database is described in Larsson and Persson [1987 c].

A Guide for the Reader

The first chapter of this thesis presents the problem of designing a knowledge-based help system for Idpac and gives an outline of a solution. In chapter 2 the concept of *scripts* and their use is described more closely. Chapter 3 is the most comprehensive chapter. It describes the implementation of the help system in detail. Chapters 2 and 3 may be skipped during

a first reading. Chapter 4 contains an example knowledge database and describes a test run using it. The reader mainly interested in system identification and the one who wants to see an example of what the system can do should move directly to chapter 4 and later return to chapter 2 and 3. Chapter 5 discusses possible extensions and details that was never implemented.

2

Scripts

Was der Fall ist, die Tatsache, ist das Bestehen von Sachverhalten.

A large part of the knowledge needed in an expert interface for Idpac is concerned with sequences. An Idpac session consists in the user solving a sequence of problems, each one demanding a sequence of commands to be performed. The concept of *scripts* is used to represent sequences. Scripts was originally used in natural language understanding, Schank and Abelson [1977], Schank and Riesbeck [1981]. Scripts can be implemented in several ways, notably with production rules or outright as Lisp lists. The latter solution was used in this project and is described in detail in this chapter.

The Need for Representing Sequences

Running Idpac takes a lot of procedural knowledge. In a typical Idpac session the user solves one sub-problem after the other. He might e.g., start by reading a data file and plot it to get a feel for what has happened during the experiment that generated the signal. After this the user probably tries to find out statistical properties of the data and pick out a few interesting parts of the data series. He uses these parts to perform maximum likelihood estimations of systems of increasing order. When he has reached what he thinks is a good model, he usually performs a few validation tests, in order to verify the model obtained.

Each of these sub-problems is solved with a sequence of commands. The data validation would e.g., involve the `plot` and `stat` commands to find out which parts of the data series are interesting, the `cut` command to pick those parts out and, finally, the `trend` command to remove biases from the chosen data sets.

If the expert interface is to be able to give help on the methods used in system identification, it is obvious that it must have a flexible and powerful way of representing sequences. There is of course also a need for making the data structure easy to use during implementation of the knowledge database. One would very much want to have a possibility to structure the sequences hierarchically so that an implementor could type in sequences of sub-problem

solvers, i.e., a subroutine or "macro" facility.

Representing Sequences with Production Rules

One obvious solution is to use production rules, representing the sequences as state transitions. This is a very simple solution, utilizing the rule system that certainly will be present anyway. A typical rule might look like this.

```
(rule state-transition-1
    (state is wait for plot or trend)
    (command is plot)
 -->
    (remove 1)
    (fact state is wait for trend)
    (fact perform actions for plot))
```

This rule would take care of a plot command, if the system is in a certain state (`wait for plot or trend`), changing the state of the system, updating the fact database and triggering the actions that should be taken. Implementing the sequences in this way means that one must have one rule for every possible transition. For all reasonable sizes of the knowledge database this would mean hundreds if not thousands of rules, with a lot of intricate interrelations. With this approach there is no obvious way to structure such a rulebase, and also no easy way to implement a subroutine facility which automatically takes care of the passing of parameters.

A Data Structure for Sequences

An alternative to using production rules is to define a new data structure and develop methods for working on it. This is the solution chosen in our project. The data structure is called script. This concept is thoroughly described in Schank and Riesbeck [1981], where it is used in natural language analysis. The idea is simple and attractive. Schank's program uses a script that describes the different steps involved in visiting a restaurant, and matches it to an actual conversation. In this way it can tell who is saying what and what has happened in the meantime. E.g., the line "Could I have the bill, please?" would only be uttered by a customer, and it indicates that the dinner has probably already been eaten. The scripts could also tell you that the sentence is actually not a question, but a request for paying.

Natural language analysis is a complicated problem. An Idpac session may also be described in a language, by stating a sequence of commands. We feel that the concept of scripts is very well suited for the much simpler problems encountered in an expert interface. So the choice was made to represent the command sequences as scripts. The implementation is straight-forward and uses Lisp lists with different kinds of clauses. Let us look at an example of a script.

```
((command conv (outfile DATA-T) (infile WORK))
 (command plot (infile DATA-T))
 (or
  ((command trend (outfile DATA) (infile DATA-T)))
  ((command cut (outfile DATA) (infile DATA-T))))
```

```
(command plot (infile DATA)))
```

Examples of the **command** and **or** clauses are seen. The script says that first one should use **conv** to convert the data from ASCII to binary representation, then look at it with **plot**. Then there is a choice between doing **trend** or **cut**, and after that one should plot the data once again. The **infile** and **outfile** sub-clauses are used to give the data files names that are internal to the script. Thus the script says that after the **conv** command the resulting file, and not an other one, should be plotted.

The Script Language

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>[<identifier>...])
MACROOUTCLAUSE      ::= (out <identifier>[<identifier>...])
```

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

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 automatically created. 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 cut (outfile CUT-FILE) (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 of the 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 practise 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.

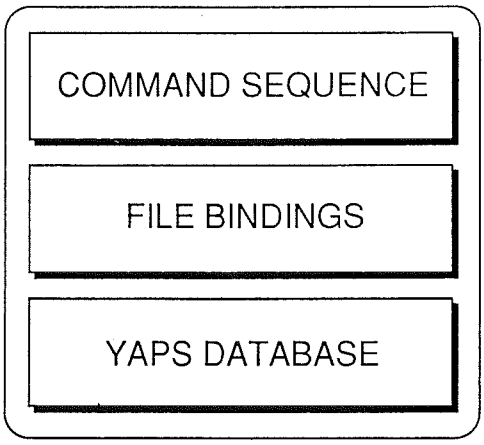
There are, of course, several other control constructs that could have been implemented. One would be a repetition with a countdown, i.e., with a specification of the maximum number of iterations allowed. Another would be the **some** construct, matching one or more commands. This construct is easily expressed as **(repeat ((or ...)))** though. So far it has proven very easy to implement new script constructs. The only difficulty seems to be to decide which ones that are really needed. Only a substantial experience of programming with scripts will make it possible to decide this.

One of our findings is that it is inconvenient to demand that scripts can be parsed with a one command lookahead only. Many scripts and script-macros start with the same commands. For this reason the current implementation of the script matcher does not demand the scripts to be parsed with a one-command lookahead. A more general parsing scheme is easily implemented with pattern matching techniques. This does have a drawback though, it is inefficient. In a later implementation, a design alternative may be to use existing software for parsing, e.g., programs like LEX, Lesk [1975], YACC, Johnson [1975], or other compiler-compilers, Wirth [1976], Aho, Sehti, and Ullman [1986]. These programs are usually not able to parse anything but one-step lookahead languages. In this case one would have to put limits on the script programmer to write the correct kind of scripts. As these limits would probably feel somewhat artificial, this talks for the solution presented here.

The Script Matcher

The syntax of the script language mirrors that the implementation was done in Lisp. We will now give a closer description of the script matching device.

The script matcher works by incrementally matching the incoming commands to the database of scripts. In this process the scripts are internally changed. In order to explain what happens, we will first have to look at a script as it is represented in the database.



The data structure of a script.

The three main parts of a script object are a command sequence, a file bindings list, and a YAPS database. The command sequence is a Lisp list containing the script language data structure. The file bindings list is an association list that keeps track of what internal names are associated with what external filenames and parameter values. The YAPS database, contains rules and facts for the production system included in the script object.

In the process of matching, the script matcher updates the command sequences, file bindings and YAPS databases. If the first command of the command sequence does not match the incoming one, the script does not match, and is put away for the time being. Otherwise, the matching command is removed from the sequence, the file bindings are updated, facts are fed to the production system and the command sequence is expanded until the next command is found. The details of this are found in chapter 3.

The earlier version of the expert interface, Larsson and Persson [1986 b], saved the actual command history and performed a complete matching of all previous commands over and over again. The new script matcher works incrementally and is very fast. This is important, since the command spy must be able to do its work within fractions of a second, if the user is not to be disturbed.

In case that several scripts match simultaneously, each one is updated separately. All the internal data are kept completely separate and no mixing problems appear until it is time for the results to be presented on the screen. For this reason, all help is tagged with the name of the script that generated it. After a few commands have been given, all but one script will usually have been matched off. If the user wants to have the help of one script only, the **`set script`** command tells the script matcher to do away with all other scripts.

When a script no longer matches, it is put in a suspended state. If later no scripts match a command, the suspended scripts are tried again too see if they match. This takes care of the case were the user follows a script for a while, then goes off to do something else, and finally returns to the previous task. It also makes it possible to let any number of different or alike scripts to be performed in parallel by the user. Taken to the extreme, this may be very confusing, but still, the system can handle it.

There is also a number of commands that are always allowed. This may for example be to plot a file or look at the values of Idpac parameters. If a command does not match any script, a list is checked to see if it is a command that is always allowed. If so, it is simply skipped in the script matching. A further development would be to associate one list of allowed and one of forbidden commands with each script. But at this time there is only one global list of commands that are always allowed.

Currently there is no way to allow that the user leaves out a command from a script. This must be taken care of when writing the scripts. Several strategies that would allow a "looser" matching could be devised, but since we do not know any good strategies, this has been pushed into the future. Loose or macro pattern matching is discussed in Bobrow and Winograd [1977], Hayes-Roth [1978] and Charniak *et al* [1980]. Such matching strategies would most likely not allow an incremental scheme, and would thus not be very efficient. It is always possible to take care of this kind of things, however, by writing the scripts so that they will accept commands in many different orders. In general, facilities lacking in the script matcher can usually be compensated for by writing more complicated scripts, and facilities lacking in the script language can be taken care of by the production rule system.

3

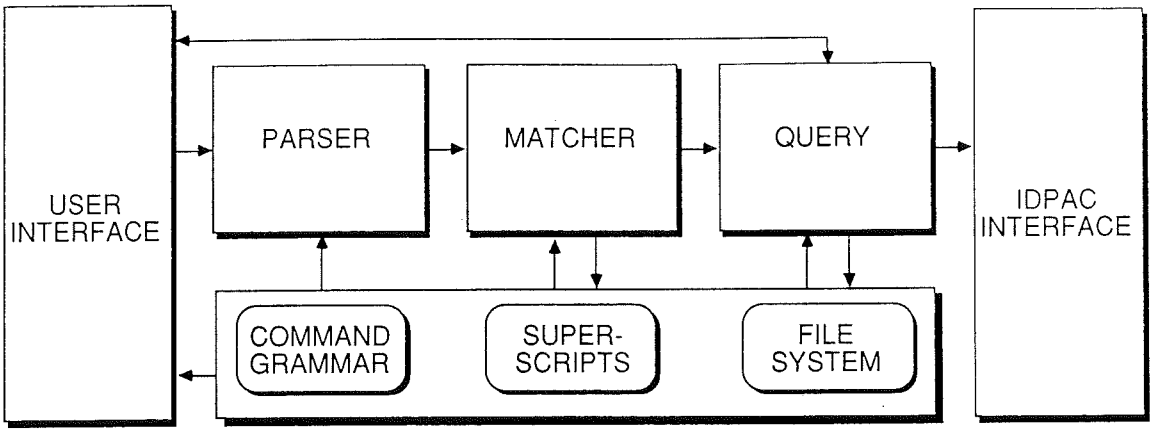
Implementation of the System

Das logische Bild der Tatsachen ist der Gedanke.

In order to explore what an expert system based help system really can do, there is no other way than to design one and try it by test runs. Therefore the implementation is crucial to the project and has formed a major part of it. An intelligent help system has been designed and implemented. It contains all the central parts, as the 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 the query module, a file system and interfaces to the user and Idpac. There are some limitations though. The interface presently 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. Also, the knowledge database is far from complete. Apart from this, everything described in this chapter has, unless explicitly stated, been implemented and tested.

General Layout

The expert interface is made up from several independent 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 and 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.

Each script object contains a YAPS database. When a 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 works 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 e.g., 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 [1984], and YAPS, Allen [1983]. It consists of about 5000 lines of code and runs under VMS on a VAX 11/780. The system is further documented and the source code is given in Larsson and Persson [1987 b].

The rest of this chapter will describe how a command is treated by the help system and give a thorough description of the different program parts.

The Transformations of a Command

One of the general design philosophies of the project has been to view the different actions taken with a command as translations between different representations. A command passing from the user through the parser, matcher and query to Idpac will go through a number of different transformations. The transformations are all done with pattern matching Lisp functions. We will now show an example of this process.

The Idpac command **conv** has the following grammar specification, taken from the User's Guide, Wieslander [1980].

```
conv dname < fnam [(C1...)] ncolx [tsamp]
```

The **dname** is the resulting binary data file, **fnam** the ASCII input file to be converted, **C1...** a column number specification for **fnam**, **ncolx** the number of columns and **tsamp** the sample interval of the output file.

Suppose that two scripts match the command sequence so far. One of them is a script for maximum likelihood estimation and its name is **ml-script**. The other performs correlation analysis and is named **corana-script**. Each script has a file bindings list that looks like this.

```
(file-bindings (file-2 WRK) (file-1 WRK))
```

The file bindings list maps internal names to external. The internal names are names of files and variables in the scripts. The external names are the corresponding names typed by the user. In the maximum likelihood script, the names **file-1** and **file-2** signifies the real world file **WRK**.

The maximum likelihood script has two alternative branches. This may have resulted from an **repeat**, **and**, or **or** construct. Thus there are two internal names (**file-1** and **file-2**) associated with the external name **WRK**.

The user types the the very cryptic command **co**, which the user interface turns into the Lisp expression

```
(co)
```

The parser recognizes the short form for **conv** and produces the following, somewhat longer, command description. This is done with the help of a command grammar, where all the clauses of a command are specified.

```
(conv
 (outfile *unknown*)
```

```
<
(infile *unknown*)
(numlist ALL idpac-default)
(number *unknown*)
(number delta idpac-default))
```

The parser guarantees grammatical correctness for its results. The ***unknown*** signifies an unknown value and the **idpac-default** tells the system that this value came from the parser's automatic defaulting. All clauses ending with **idpac-default** will be removed before the command is read by Idpac. This simplifies the interface considerably. In this case the interface does not have to supply a list of all columns in a file if the whole file is to be converted. Idpac's normal defaulting mechanism takes care of this. Thus, the column description may simply be represented with **ALL**. The command description is passed to the script matcher, and after going through it, the command looks like this.

```
(conv
 (outfile (ml-script (file-3 *unknown*) (file-4 *unknown*))
          (corana-script (file-5 *unknown*)))
 <
 (infile (ml-script (file-2 WRK) (file-1 WRK))
         (corana-script (file-1 WRK)))
 (numlist ALL idpac-default)
 (number *unknown*)
 (number delta idpac-default))
```

The **infile** and **outfile** clauses have been changed. They now contain one part for each script. Information about what internal name corresponds to what external name are found in these parts. The maximum likelihood script has two alternatives, so there are two different name mappings. The infile's external name could be defaulted as it was found in the file bindings lists. There is no information about the external name of the outfile, though, so it remains ***unknown***.

The query module now tries get rid of any remaining unknowns and ambiguities by asking. The second last parameter is the column number for the resulting data file. It cannot be defaulted by Idpac or by the scripts, so it must be asked for. After the user has been asked for the unknown parameters the command description turns into the following.

```
(conv
 (outfile (ml-script (file-3 DATA) (file-4 DATA))
          (corana-script (file-5 DATA)))
 <
 (infile (ml-script (file-2 WRK) (file-1 WRK))
         (corana-script (file-1 WRK)))
 (numlist ALL idpac-default)
 (number 1)
 (number delta idpac-default))
```

After this everything is known, and the script objects may be updated. This will happen in a second pass through the matcher. The first pass through the matcher does not change any data in the scripts. The second pass is needed because the user may specify a file during the query that is not consistent with the scripts. In this case it is wrong to update them. The file system includes the newly created outfile in the directed graph structure, and the command is put through a last pass, that turns it into this.

```
(conv DATA < WRK 1)
```

In the Idpac interface this is turned into a character string and sent to Idpac.

The technique of using small pattern matching functions that translate Lisp expressions from one representation to another is simple but very powerful. It is used throughout the system. Output is sent to the user interface in Lisp list form and the actual printout is confined to that module. The production rules are entered without names and fed through a filter in order to give them the appropriate names, etc. The strength of this technique is that the transformations are easily and compactly defined in a grammar-like style.

The Command Parser

The first thing that happens with a command sent to the expert system interface is that it is parsed for grammatical correctness. The parser does the same job as does the front end of Idpac. The reason for not using the Idpac parsing was to avoid any "close encounters" with the Fortran code of Idpac.

The parser actually does more than Idpac's command decoder. It accepts short forms of the commands, i.e., it allows one to abbreviate the command name as long as it does not become ambiguous.

The parser works with a command grammar, defined in a "lispified" version of Extended Bacchus-Naur Form. In this way it is a trivial matter to define a new command. The code uses a pattern matcher to do the job. It could have been done with transition network techniques, Winston and Horn [1981], or with a parser-generator such as YACC, but the current implementation is so simple and straight-forward that it is doubtful whether anything more complicated is justified.

One of the ideas behind the expert system interface is that the user should be able to leave out arguments in the commands, to be filled in by the system. In order to do this the parser must be able to accept incomplete commands. The matcher and query modules will take care of the arguments later on.

The parser gets a list of tokens from the lexical analyser. These tokens are atoms or Lisp lists. It compares the command tokens with the grammar specification of the command. If the parser encounters a token in the command whose type does not match the type of the clause in the corresponding position in the grammar, the parser has two alternatives. Either a token has been left out, which is marked as `*unknown*`, or there is a default clause in the grammar, in which case the value of the default clause will be used. The parser signals an error if any symbols remain in the command when the specification is empty.

The result of the parsing is the grammar specification of the command in question, with values associated to all items in the specification, either the actual values typed by the user, `idpac-default`, or `*unknown*`.

Earlier the Idpac User's Guide definition of the command `conv` was shown. The Lisp grammar of the command `conv`, used in the expert interface is

```
(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.")
             (""
              "No comment.")
             ("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.")))
```

The first part of the definition is used by the parser. The `outfile` and `infile` clauses contains information about the type of file. This may be either text file (`.t`) or data file (`.d`). The second part consists of short and long help texts used by the Query module. All commands and macros are treated equally and new ones are easily introduced. All that has to be done is to enter a description such as the one given above.

Let us look at an example of how a command is treated. If the user types the command

```
conv 1
```

The resulting parsed command will be the list

```
(conv
 (outfile *unknown*)
 <
 (infile *unknown*)
 (numlist ALL idpac-default)
 (number 1)
 (number DELTA idpac-default))
```

If the user types the command

```
conv < myfile 1
```

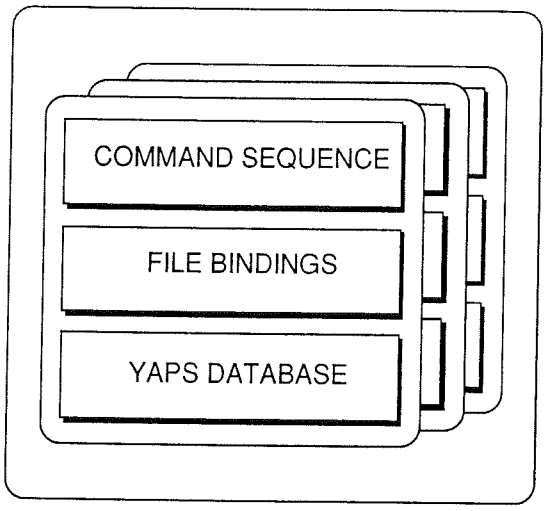
The result will be

```
(conv
 (outfile *unknown*)
 <
 (infile myfile)
```

```
(numlist ALL idpac-default)
(number 1)
(number DELTA idpac-default))
```

The Script Matcher

This module is the very heart of the expert interface. It works with the script data structure defined in chapter 2.



The structure of a script object.

A script object is composed of three major parts. The command-sequence is a Lisp list containing lists of the actual commands and control directives, against which the incoming commands are matched. The sub-lists are written in the script language presented in chapter 2. The file bindings is an association list containing pairs of internal and external names. The internal names are the names of data files and parameters used internally in the script and the external names are the corresponding actual data files and parameter values. The third part is the YAPS database which contains facts and production rules. The **kscall** clause is used to send a fact from the script object to the YAPS database. There is a mechanism for transferring parameter values from the command, through the matching over to the fact database. Script objects are implemented as instances of a script flavor, and they inherit a YAPS database. In this way each script object has its own fact database and its own rules, i.e., several production rule systems may run independently of each other.

The command-sequence instance variable in a script object contains a number of scripts, expressed in the script language. When matching a command, the first thing to be checked is whether the command name is equal to the name of the first command in one or more of the scripts. The parameters are then matched.

A parameter of a command in the script declared as **globfile**, **number**, **numlist**, **numlist1**, **symbol**, **symlist**, or **symlist1** matches any value in the actual command, provided that the argument is of the indicated type. A **number** is a real or an integer number, a

numlist is a list of numbers, a **numlist1** is a **numlist** of length one, a **symbol** is string of alphanumeric characters beginning with an alphabetic character, a **symlist** is a list of symbols, and a **symlist1** is a **symlist** of length one. A file declared as **globfile** must be associated with an already existing file and the name of that file supplied by the user. A **globfile** only matches a parameter declared as **infile** in the actual command. The **number**, **numlist**, **numlist1**, **symbol**, **symlist**, and **symlist1** clauses pick up values from a command to the script. These clauses may be left out when defining a script.

A parameter declared as **outfile** matches against any value of the corresponding **outfile** in the command. This is used when the command creates a new file. If the user does not supply values for files or parameters, the internal names will be bound to ***unknown*** and the query module will ask the user for the values later.

A parameter declared as **infile** matches if it is found in the file bindings list and the external name is equal to the internal name or ***unknown***. Using an internal name as a parameter in an **infile** clause requires that it has previously occurred in an **outfile** or **globfile** clause. The **infile** declaration is used to get the intelligent default of the system. The system assumes that the internal name in an **infile** clause already exists in the script.

The matching of a command is performed in two passes. In the first pass the script is matched against the command and the result of this matching is a new version of the command containing information from which unknown names can be deduced from the script. If the first matching fails, nothing happens to the script object. It will keep on waiting for a new command to match. After the first pass through the matcher the command is passed to the query module where the user will be asked questions to resolve all ambiguities. When the command is complete, i.e., no ***unknown*** symbols are present in the command and all internal names are bound to one external name only, the script will be matched a second time, this time against a complete command. The second matching changes the data in a script object. It starts by doing the same things as the first matching. Then all scripts on the command-sequence list which do not match will be removed. After that the scripts left will be expanded until their first element once again is a command. During the expansion all **kscalls** encountered will be put on the kscall list to be handled after the command have been sent to Idpac. Note that these are the **kscalls** that come after the command that was matched.

An Example of Script Expansion

The expansion of a script is best clearified by an example. Suppose the command-sequence of a script object is

```
(
 ((command x1)
  (kscall (A test call))
  (repeat
   ((or
     ((command x2)
      (command x3))
     ((command x4)))
    (command x5))))
)
```

Let us assume that this list of one script has just succeeded in the first and second matching and should now be expanded. First the matching command `x1` will be removed and the fact `(A test call)` will be put on the kscall list. This leaves

```
(
 ((repeat
   ((or
     ((command x2)
      (command x3))
     ((command x4)))
    (command x5))))
)
```

The `repeat` means that the script inside it must be used at least once. The command sequence will now be expanded to

```
(
 ((or
   ((command x2)
    (command x3))
   ((command x4)))
  (command x5)
  (repeat*
   ((or
     ((command x2)
      (command x3))
     ((command x4)))
    (command x5))))
)
```

The command sequence is finally transformed into two sequences, which makes it possible to remove the `or` clause.

```
(
 ((command x2)
  (command x3)
  (command x5)
  (repeat*
   ((or
     ((command x2)
      (command x3))
     ((command x4)))
    (command x5))))

 ((command x4)
  (command x5)
  (repeat*
   ((or
     ((command x2)
      (command x3))
     ((command x4)))
    (command x5))))
```

```
)
```

The original command-sequence contained one script. After the matching and expansion it contains two scripts. These scripts match the commands **x2** or **x4**.

Expansion of the Script Clauses

When a **scriptmacro** is encountered during the expansion, all occurences of the formal **in** and **out** parameters in the script-macro will be substituted by the actual variables and the script-macro placed first in the script. After this the expansion continues.

An **all** clause is transformed into a list of scripts, each beginning with a parameter of the **all** clause followed by a new **all** clause. Consider the following example, where **(A)**, **(B)** and **(C)** are scripts.

```
(
 (all (A) (B) (C))
)
```

This is transformed to

```
(
 (A
  (all (B) (C)))

 (B
  (all (A) (C)))

 (C
  (all (A) (B)))
)
```

The scripts **(A)**, **(B)** and **(C)** will be further expanded. With the same assumptions **repeat*** is expanded according the following example.

```
(
 (repeat* (A))
 B
)
```

The expansion will give

```
(
 (A
  (repeat* (A))
  B)

 (B)
)
```

When an **assign** clause is encountered the arguments of the clause are put in a pair on a temporary list. Before the command is read by Idpac the new-names of the pairs on the temporary list are associated with the external names of the the old names, and pairs of internal names and external names are put first in the file bindings list.

A problem may arise when we have a command sequence list like

```
(
 ((repeat* ((repeat* ((command x)))))
  (command y))
)
```

The first expansion, according to the expansion rules of **repeat***, gives.

```
(
 ((repeat* ((command x)))
  (repeat* ((repeat* ((command x)))))
  (command y))

 ((command y))
)
```

The next expansion would give

```
(
 ((command x)
  (repeat* ((command x)))
  (repeat* ((repeat* ((command x)))))
  (command y))

 ((repeat* ((repeat* (command x))))
  (command y))

 ((command y))
)
```

The second of these lists looks exactly like the original script, and should not be in the list. To handle this case, the script expanding function keeps track of which scripts have been expanded, and will not put a script in the command sequence list if it has been expanded before. In this case the result of the expansion of the original script will be

```
(
 ((command x)
  (repeat* ((command x)))
  (repeat* ((repeat* ((command x)))))
  (command y))

 ((command y))
)
```

Programming languages are often designed so that it should be possible to parse a program with only one symbol lookahead. Writing scripts for Idpac is to specify a language for solving problems with Idpac. Demanding that the script language must be parsed using only a one symbol look ahead would complicate things for the knowledge engineer. The incremental algorithm used puts no such restrictions on the scripts, but of course it may prove to be inefficient.

YAPS

Each script object contains a YAPS database, where rules and facts are stored. The production system shell YAPS, Allen [1983], is used to store facts about the current session. This enables the expert interface to give diagnostics after certain commands, etc.

Facts in YAPS may be arbitrarily nested Lisp lists. An atom starting with a '-' is a variable and matches anything in an actual fact. A typical YAPS rule may look like this.

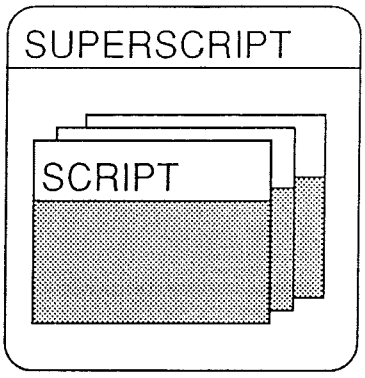
```
(defp Beer
 (bar open)
 (beer price is -cost)
 (funds contain -funds)
 (~ (too drunk))
test
 (>= -funds -cost)
-->
 (remove 3)
 (fact has a beer)
 (fact funds contain ^(- -funds -cost)))
```

If the facts `(bar open)`, `(beer price is 1)`, and `(funds contain 3)` but not `(too drunk)` are present in the fact database, the rule may be triggered. The variables `-cost` and `-funds` are associated with the integer values in the facts and the condition after `test` is found to be true. The third fact, `(funds contain 3)`, is removed from and the facts `(has a beer)` and `(funds contain 2)` are added to the database.

YAPS is a standard expert system shell. Its rule notation is simple and easy to use. Examples of rules are found in chapter 4. A great advantage with YAPS is that it may be run by Lisp function calls and not only from a special top-level. This makes it very easy to integrate in a Lisp-based system.

Superscripts

All script objects, containing scripts able to match a command, are initially stored in a superscript. This is essentially a list of script objects.



The structure of a superscript.

When the system is initialized an initial superscript containing all scripts that the knowledge engineer has defined is created. This initial superscript is copied and stored in a list in the database. It will be used when matching commands. The main use of superscripts is to allow for several parallel tasks.

There may be several superscripts in the list of superscripts in use. The first is the currently active and the one that the system first tries to match the command with. If a command does not match the current superscript, the system tries to match with the suspended superscripts. These are found in the tail of the list of superscripts in use. If one of the suspended superscripts match, it is moved up to the front and used as the new current superscript. The old current superscript gets suspended. If there is no suspended superscript or none of the available ones match, the system tries with the original superscript. This takes care of a fresh start. If the original superscript matches, it is used as the new current superscript and the old one will be suspended. If it does not, the system gives the message "No match" and remains in its present state. Some commands are allowed anywhere in a script. This is checked immediately before giving the "No match" message.

This strategy enables the system to handle any number of sessions in parallel. It will switch to the appropriate context whenever a command or filename does not match the current session. If a command matches neither the initial superscript nor any suspended one, the system waits and tries to get back in action after every new command. This takes care of the case where the user leaves the scripts and gives a few other commands before getting back. The command given will always be read and processed by Idpac.

The Query Module

One task of the query module is to resolve the ambiguities that may arise when a command is matched against one or more scripts and the user has left out arguments in it. The other is to fill in values of arguments that the user left out when entering the command. It is of course very important that the query module works even when a command does not match any of the scripts.

The ambiguities can be due to the fact that several scripts match because they all start with the same command name, or that one script has branched into several scripts which all start with the same command name but have different infiles as default.

The query module first looks at all infile descriptors in the command. If all external names in the descriptor are equal nothing happens, but if there are different external names in a descriptor the user will be asked for the file he wants the system to use. If the user chooses one of the possible default files the query system will go through the command and remove all impossible alternatives in the infile, outfile, globfile and parameter descriptors. If not, the command does not match any script. Then the query system will look for another ambiguous infile descriptor. The system iterates through the command until none are left.

The query module then goes through the entire command looking for the symbol `*unknown*` in all outfile, globfile, and parameter descriptors. This means that the parameter has been left out and cannot be defaulted from the scripts. The user will be asked for the file or parameter. After this the command is complete.

At any time when the user is asked for a parameter he may ask for help on that particular argument by typing a '`?`'. This will give him a short help text for the argument. The text

may include hints on naming conventions, values of numerical arguments and so on.

Next, when the command is fully specified, the query module sends the command to the matcher for the second matching. If the command still matches, the query module updates the scripts, leaving them ready for a new command.

The query module is the last module the command passes through before it is sent to Idpac via the lowlevel routines in the Idpac interface. Therefore the query module also does the final bookkeeping. The file bindings in all matching scripts will be updated to the new values, all assignments specified in `assign` clauses will be performed, and the internal file database and the command history in the database will be updated. In some cases the command is a "special" command. This means that it changes states both in Idpac and in the interface. Examples of this are `let`, which changes an Intrac variable, and `delet`, which deletes a file. In these cases special functions performing the appropriate actions will be called. These functions are located in the data base. The function associated with the `let` command updates a list of Intrac variables, and the function associated with the `delet` command marks the file as deleted. Finally, the command will be sent to Idpac via the Idpac interface module. After the command has been sent to Idpac the expert systems of all active scripts will be started. This must not happen before the command has reached Idpac and been processed, because the expert system may ask for information not available until the command has been executed.

The Database

The database contains all essential data in the system and functions to operate on it. The superscripts are placed in a list in the database. The database provides methods for matching a command against the scripts in a superscript. The script macros are placed in the data base together with functions to operate on them. When needed, the scripts make calls to the data base to get the code of the script-macros for the expansion of the scripts.

The command grammar is an instance variable in the database and is accessed from the parser. The list of allowed commands, i.e., a list of commands which are always permitted, is another instance variable. If a command does not match any script, the matcher checks if the command name is in the allowed commands list. Functions for updating the file system are also placed in the data base. Some state variables for monitoring the system behaviour (e.g., printing CPU and garbage collection times) and for changing it (e.g., starting to print traces) are also located there.

A number of commands are declared as internal. They affect the interface and are never sent to Idpac. They may perform things like listing the file database, exit into the Lisp system, dump the file system on disk, and so on. A list of these commands and the functions that handle them are also found in the data base.

Some commands are declared "special." A list of these and the functions which are called when such a command is encountered by the system are also located in the database.

The File System

The file system is a database which keeps track of the files in the system, and the relations between them.

A file in the file system is represented as an instance of a file flavor. Its instance variables are type, parents, children and command. The type tells which kind of file it is. There are two possibilities, data or text file. Most Idpac commands operate on files and usually take an infile, performs some operations on it and generates an outfile. The files generated from a certain file are called the children of that file, and the files from which a certain file is generated are called its parents. The instance variables children and parents are lists of the children and parents of a file. The instance variable command tells by which command the file was created.

The file system has methods for adding and deleting parents and children from a file object. The actual creation of files and managing of the file system is handled by functions in the database.

It is possible to ask the file system for all ancestors or descendants of a file, and get a graphical representation of the dependencies on the screen.

Files can be deleted with Idpac commands, either explicitly (with `delet`) or implicitly by overwriting them with a command giving an outfile with the same name as an already existing file. When this happens the database is updated so that the deleted file will be marked as deleted. This is necessary because other files may depend on a deleted file. When a file is asked for its ancestors or descendants, deleted files may appear in the tree of the relatives of a file, marked as deleted, and not accessable on the disk.

The User Interface

The user interface contains all functions needed for the communication with the user. All input and output goes through this module. In this way all terminal dependent information is kept in one place. Among other things, the user interface supplies methods for menu handling and reading expressions with a type check. Currently only VT100 and VISUAL terminals are supported.

The Idpac Interface

The Idpac interface is a small module which converts a list to a character string and sends it to Idpac. The actual communication is carried out by two small routines written in C. These are linked into the Lisp system.

When Idpac is run with the expert system interface, it resides in one VAX/VMS process and the expert system interface in another. These two processes communicate via VAX/VMS mailboxes. When the expert system interface is started, a C function creates a mailbox for the communication with Idpac. After this, Idpac is started from a command procedure where its input is assigned to the mailbox. In this way no changes at all had to be made in Idpac's source code.

The Idpac Interface module uses the VAX/VMS system services `SYS$QIO` and `SYS$QIOW` for putting messages in the mailbox. The routine `SYS$QIO` puts a message in a mailbox, and continues the execution of the sending program immediately. The routine `SYS$QIOW` also puts a message in a mailbox but suspends the execution of the sending program until the message has been read.

When Idpac is run interactively and a command is typed, Idpac reads and executes the command, the result is displayed and a prompt appear on the screen. When Idpac is run with

the interface the user should see the same thing. In order to get this behaviour, the command is sent to Idpac via a mailbox with the routine `SYS$QIO`. Immediately afterwards an empty command (a space) is sent with the routine `SYS$QIOW`. This makes the sending process wait for the prompt until Idpac has executed the command (and the empty command, which takes almost no time). If the command had been sent using the routine `SYS$QIOW` only, control had been returned to the interface before the execution of the command had been finished.

With these routines a Lisp interface to any program can easily be obtained. For readings on VMS system routines, see Digital Equipment Corporation [1984].

Utility Functions

The system also has an on-line dictionary, containing information about concepts mentioned in the help texts, commands and scripts. In this way all help may use the vocabulary of system identification, and the user will learn about the subject while utilizing the help system.

The behaviour of the interface may be changed by setting different switches. The switches affect tracing outputs for debugging purpose, statistics of the run-time behaviour of the Lisp system and the amount of help the user will get. These switches are controlled with a menu system. This menu can also be used to change between the expert and beginner modes.

Lisp, Flavors and YAPS

The system has been implemented in Franz Lisp, Foderaro and Sklower [1981], on a VAX 11/780. The object oriented framework Flavors, Allen [1984], and the production system shell YAPS, Allen [1983], was used on top of the Lisp.

All modules have been implemented as instances of flavors. The object oriented approach helps to enforce programming discipline. Dynamic creation has, however, only been used in two cases, in the scripts and in the file system. The inheritance mechanism has only been used in the script objects, to inherit the YAPS database.

The Flavors system is used on top of the Lisp system and this has some drawbacks. The Flavor system uses the global Lisp name space for the methods of the instances. This means that all methods will be coded into functions, with very long and peculiar names. There are also some bugs in the Flavors system. If we were to reprogram the system, we would probably not use the Flavors system.

YAPS uses a global rule discrimination network common to all database instances. This also is a clear violation of the object oriented philosophy. The expert interface sometimes demands copies to be made of YAPS databases. In order to accomplish this, YAPS had to be rewritten so that each database has its own discrimination network. The copy operation gets very complicated and time consuming anyway, as one must copy the entire network.

4

The Knowledge Database

Der Gedanke ist der sinnvolle Satz.

In this chapter an example of a small knowledge database for system identification will be presented. The knowledge database consists of three parts. The scripts contain procedural knowledge, i.e., knowledge of what commands to use, and their order. The rule base contains diagnostic knowledge about identification, signal analysis, etc. The command grammar is used by the parser and contains information about the commands and macros.

Building the Knowledge Database

When building a knowledge database, the knowledge engineer writes scripts and rules. Some experiences of the building of knowledge databases have been gained from examples constructed during the project.

Some rules are useful in all scripts. This might e.g., be rules for generating output, automatic documentation, etc. Therefore a list of global rules was introduced. These rules are added to all scripts at startup time.

Another problem is to make the help texts produced by the rule system appear at the right moment. A script is often made up of parts, were each part may be skipped, i.e., `or` clauses with empty alternatives. If there is a help text in the beginning of each of these parts, all texts will mix in the output when the script matcher reaches the first alternative. This is due to the fact that the script expansion algorithm "sees through" all the empty alternatives. Therefore one must write the help texts so that they will not produce rubbish when concatenated. Another way of solving the problem is to have only one help text, placed before the first part that may be left out.

The problem mentioned above also led to another observation. The lesser the number of alternatives encountered in a script, the better the help system works. The help texts get shorter and the possible next commands are also fewer. Thus it seems to be a good design philosophy to try and avoid alternatives, i.e., `or` and `all` constructs, whenever possible.

Allowing few commands in a script will also have the effect of quickly reducing the number of different scripts that will match a certain command history.

System Identification

Idpac is capable of doing most things in system identification and signal analysis. Idpac's capabilities are described in Wieslander, [1979 c, 1980], Gustavsson and Nilsson [1979], Gustavsson [1979], and Åström [1980]. In the following example Idpac is used for identifying the parameters of a process model, given the input and output signals of the process.

One way of doing process identification will now be described. It is assumed that the measured data is available as ASCII text files. These files are first converted to Idpac's standard binary format. The files are plotted to see if the signals are reasonable. To get enough information from the process for a successful identification the input signal must excite the dynamics of the system sufficiently.

After that, the datafiles are cut in two parts, one part for the actual parameter estimation and one for cross validation. A natural choice is to cut the signals in two halves, but if the system is affected by some unknown disturbance it may be wise to cut out pieces of the signals which are not affected by it.

The coherence between the input and output signals is computed to get an indication of the frequency range where the identified model will be trustworthy. A rule of thumb is that if a deterministic model with one input and one output is desired, the coherence between the signals must be greater than 0.8 to get a meaningful model.

After this, parameters of models of different orders are estimated and the residuals examined. A successful identification should give "white" residuals, i.e., residuals with zero autocorrelation except for $\tau = 0$.

It can also be useful to calculate the crosscorrelation between the residuals and the input. A successful identification should give zero crosscorrelation for positive lags. The presence of feedback in the experiment is seen from correlation at negative lags.

It is also recommended to compute the transfer functions of the estimated models and plot them in a Bode diagram. When the curves coincide well in regions with high coherence, it is a sign that the order of the transfer function is sufficiently high. Repeating the calculation of the transfer functions with random samples of the estimated distribution of the parameters gives an indication of the spread in the estimation.

When the parameter estimation is finished the second data set is used for cross validation. The residuals of the second data set are computed using the models estimated with the first data set. The model which gives the smallest loss function is chosen to be the correct one. Maximum likelihood estimation with Idpac is described in Åström [1980]. A general description of system identification is found in Eykhoff [1974, 1981] and in Åström and Eykhoff [1971].

A Script for System Identification

This method of estimating parameters can be expressed in the following way using the script language. The script is written in the system's normal Lisp format. The text following a ';' on a line is a comment and ignored.

```
(setq ML
 '(
   ; Convert ASCII files to binary files.

   (command conv (outfile IF) (globfile IA))
   (command conv (outfile OF) (globfile OA))

   ; Look at the signals.

   (command plot (infile IF) (infile OF))

   ; Cut out the first half of the signals and remove trends.

   (command cut (outfile IC) (infile IF) (number NC1) (number NR1))
   (command trend (outfile ICT) (infile IC) (number T1))
   (kscall (fact (parameter ICT) cut-trend (parameter IF NC1 NR1 T1)))
   (command cut (outfile OC) (infile OF) (number NC2) (number NR2))
   (command trend (outfile OCT) (infile OC) (number T2))
   (kscall (fact (parameter OCT) cut-trend (parameter OF NC2 NR2 T2)))

   ; Compute the coherence between input-signal and output-signal.

   (command coh (outfile CF) (infile ICT) (infile OCT))

   ; Estimate models of different orders, compute and look at
   ; the residuals and look at the computed transfer functions.

   (repeat
    ((kscall (suggest mlid identification))

   ; Perform the estimation.

     (command mlid (outfile SYS) (infile ICT) (infile OCT) (number  NS))
     (kscall
      (fact (parameter SYS) < (parameter ICT OCT) order (parameter NS)))

   ; Look at the residuals.

     (command
      residu (outfile R1) (infile SYS) (infile ICT) (infile OCT))

   ; If you want, look at the Bode plot of the transfer function.

     (or
      ()
      ((command sptrf (outfile F1) (infile SYS))
       (command bode (infile F1))))))
```

```
; Cut out the second half of the signals and remove trends

(kscall (suggest use for cross validation (parameter IF)))
(command cut (outfile IXC) (infile IF) (number NXC1) (number NXR1))
(command trend (outfile IXT) (infile IXC) (number TX1))
(kscall (suggest use for cross validation (parameter OF)))
(command cut (outfile OXC) (infile OF) (number NXC2) (number NXR2))
(command trend (outfile OXT) (infile OXT) (number TX2))

; Compute the loss function (= sum of the squares of the residuals)
; for the models.

(repeat
 ((kscall (suggest cross validation))

; A globfile is necessary in the residu command because
; different system files will be given, and only the last
; model identified can be defaulted from the script.

  (command residu
   (outfile RX) (globfile SX) (infile IXT) (infile OXT))

; Multiply the signal with itself.

  (command vecop (outfile SS) (infile RX) (infile RX))

; Among other things the stat command displays
; the sum of the signal.

  (command stat (outfile SS))
  (ks-call (ask loss function (parameter SX)))))
(command stop)))
```

The Rule Base

In this example certain conventions have been used for the syntax of the rules in order to structure them.

Facts beginning with **suggest** are used to fire rules which print messages to the user telling him about the alternatives available. These facts are entered with a `kscall` clause.

A `kscall` clause which puts facts beginning with **ask** in the database will fire rules which asks the user. The questions may concern the result of a command (e.g., "What is the value of the loss function?" after an identification) or the user's interpretation of a certain situation (e.g., "Are there any outliers present?").

A fact beginning with **fact** indicates that this is a fact about the session, e.g., a fact about a file, or a fact about what the user has done. These facts may fire demon rules, e.g., a rule which finds the model representing the minimum loss function.

The words **suggest**, **ask**, and **fact** are only words in the rules. They have no special meaning to the production system when they appear in the if-side of a rule. The rule base which handles the **ml-script** is given here.

```
; Rule trigged when entering the estimation part of the script.

((suggest mlid identification)
 -->
 (patom "Either do ML identification using the command MLID or")
 (patom "start the cross validation using the command CUT."))

; Rule trigged when entering the cross validation part of the script.

((suggest use for cross validation -F)
 (fact -CT cut-trend -F -C1 -R1 -T1)
 -->
 (mapc
   'patom
   `("The first half you CUT and TRENDed from " ,-F " was " ,-CT
     " the first record " ,-C1 " the number of records " ,-R1
     " and removed trends of order " -T1))
 (terpr))

; Each time the loss function of a new system is computed, the
; minimum system with current minimum loss function is printed.

((suggest cross validation)
 (minimum -SYS -IF -OF -ORDER -LOSS)
 -->
 (mapc
   'patom
   `("The system of least loss function is " ,-SYS
     " estimated with order" ,-ORDER)))

; There is no way of getting information from Idpac
; to the expert system, so the user must be asked.

((ask loss function -SYS)
 -->
 (let
  ((ans))
   (patom "What is the value of the loss function?")
   (setq ans (read))
   (<- self 'fact `(fact ,-SYS loss-function ,ans))))

; Rules for finding the system which represents the minimum
; loss function.
```

```
((fact -SYS < -IF -OF order -ORDER)
 (fact -SYS loss-function -LOSS)
 (~ (minimum - -IF -OF - -))
 -->
 (fact minimum -SYS -IF -OF -ORDER -LOSS))

((fact -SYS < -IF -OF order -ORDER)
 (fact -SYS loss-function -LOSS)
 (minimum - -IF -OF -ORDERX -LOSSX)
 test
 (< -LOSS -LOSSX)
 -->
 (remove 3)
 (fact minimum -SYS -IF -OF -ORDER -LOSS))
```

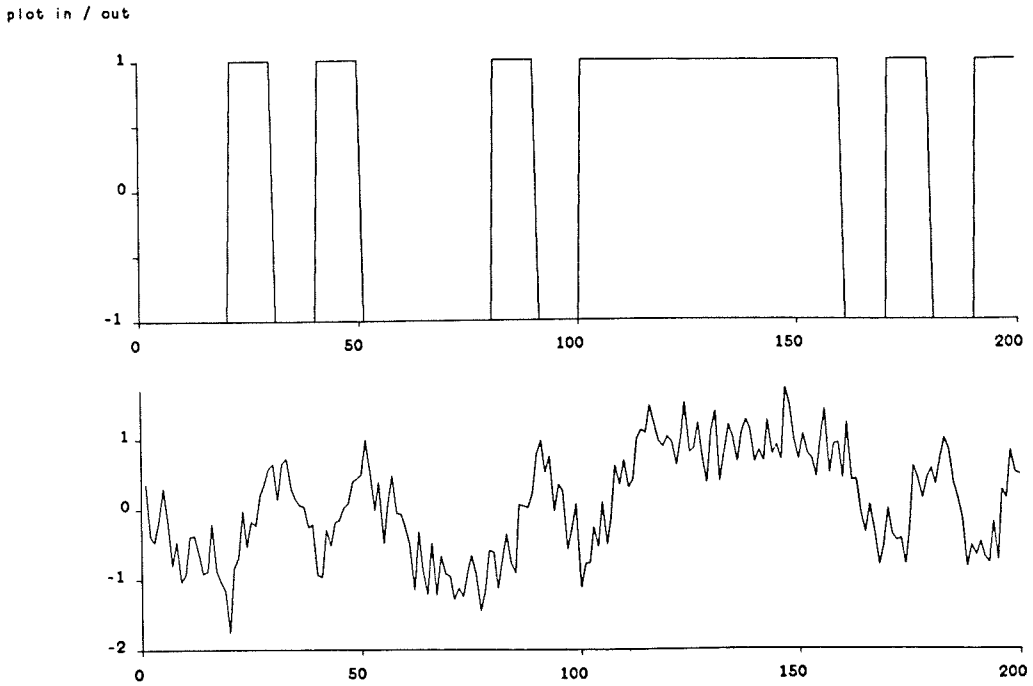
This example is a short extract a larger knowledge database, which has been developed in the project, Larsson and Persson [1987 c].

A Session with the System

To give a feel of what the system can do, here is an example run, using the script and rule base given previously in this chapter. The 'ML:' appearing before the help texts marks that the help comes from the ML-script. All text following a '"' on a line is a comment and is ignored.

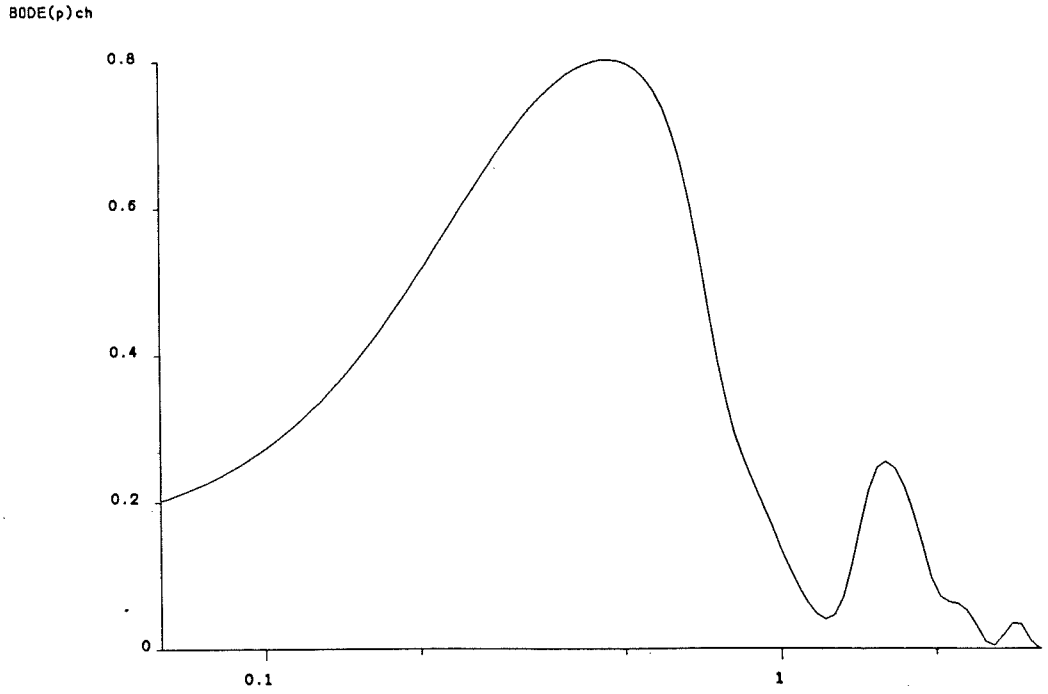
Initially the user is assumed to have the two text files `in.t` and `out.t`, containing the input signal and output signal of the system.

```
$ IDPAC
Dated 21-OKT-1986 11:37

IDPAC    V7A


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

>conv in < in 1                " Convert the in signal to a binary file.
>conv out < out 1              " Convert the out signal.
>?                             " What should I do next?
ML: plot                       " The system answers.
>plot in / out
```




```
>think                          " Enter beginner mode.
ML: cut
>cut inc < in 1 100             " Cut the first 100 datapoints of in.
ML: trend
>tr                             " Short form of command.
  trend ... < inc ...           " The file inc is defaulted.
  outfile? > inct
  trend inct < inc ...          " Only the polynomial order is missing.
  order of trend polynomial? > 0   " A question from the expert system.
ML: cut
>cut outc                       " Cut the first 100 datapoints of out.
  cut outc < out ...
  first record? > 1
  number of records? > 100
ML: trend
>trend outct < outc 0           " Remove trends of the 0'th order. (Bias)
ML: coh
>coh                            " Compute the coherence -- the user does
                                " not know the arguments of the command.
  coh ... < inct outct ...      " inct and outct are defaulted.
  outfile? > ch
  coh ch < inct outct ...
```

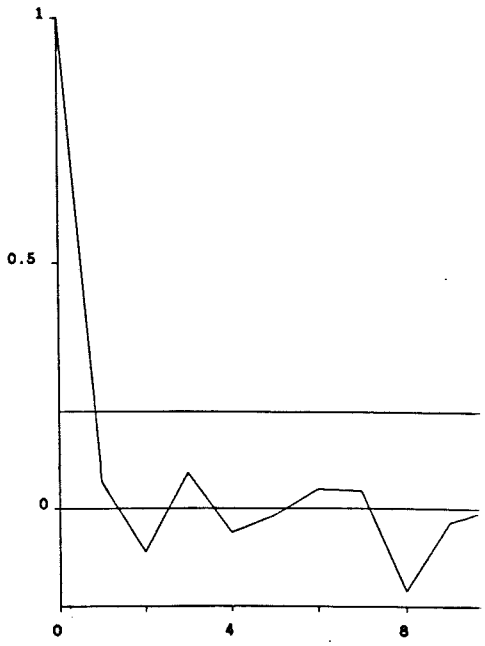
```
number of lags? > ?          " The user needs more help.
The number of lags for the computation, a good choice
is 10 to 15 % of the number of datapoints.
number of lags? > 15
```



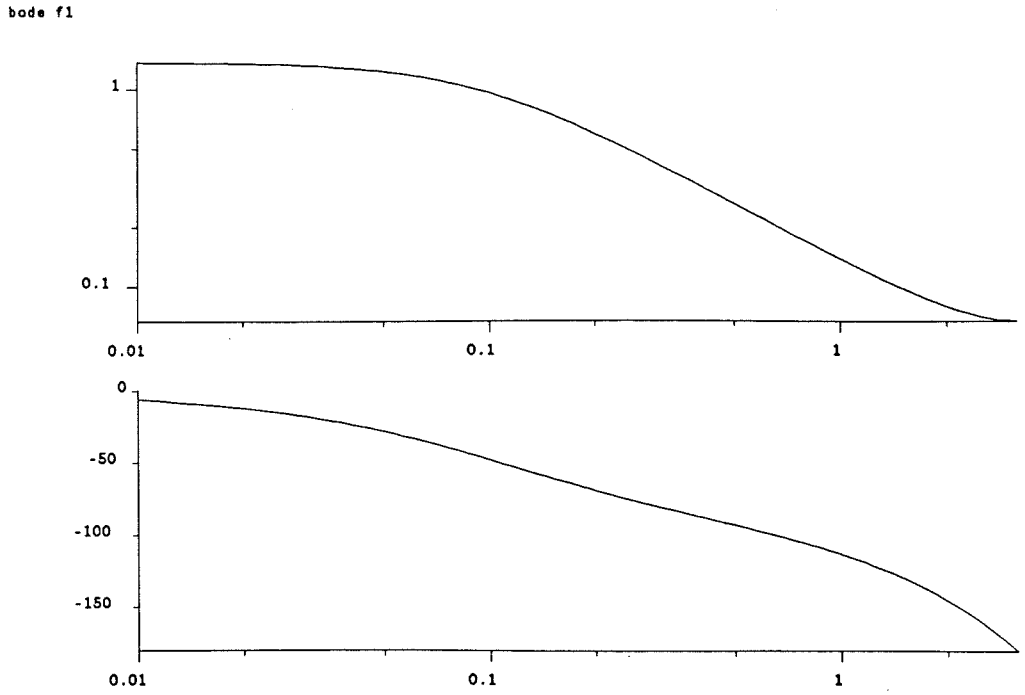

```
Either do ML identification using the command MLID or
start the cross validation using the command CUT.
ML: mlid
>mlid                          " Estimate the parameters.
  mlid ... < inct outct ...
  outfile? > s1
  model order? > 1

 CONVERGENCE (DV/V<  1.8E-06)  " Printout from Idpac.
 ****************************

 A1             -0.907205     +-  1.553992E-02
 B1              0.127194     +-  1.117196E-02
 C1             -0.828527     +-  5.851866E-02

 LAMBDA          0.328813     +-  2.325061E-02
 LOSS FUNCTION   5.40591
```

```
 AIC              67.3347

ML: residu
>residu r1                    " Other parameters defaulted from
                              " the script.
```

```
VARIANCE OF THE RESIDUALS:  0.107989

NUMBER OF CHANGES OF SIGN
OF THE RESIDUALS:    45

5 PERCENT TOLERANCE LIMITS:    39    59

TEST OF INDEPENDENCE OF THE RESIDUALS

E(RES(T)*RES(T+TAU)) FOR: 0<TAU<   11

TEST QUANTITY:   4.87828
DEGREES OF FREEDOM:    10

TEST OF NORMALITY

TEST QUANTITY:   8.10044
DEGREES OF FREEDOM:    17
```



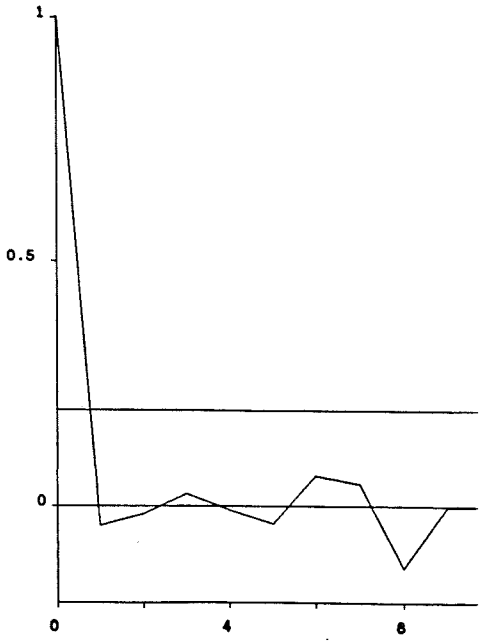

```
ML: mlid, sptrf, cut          " Several alternatives are possible.
>sptrf                        " Compute the frequency response.
  sptrf ... < s1 ... / ...
  frequency response outfile? > f1
  sptrf f1 < s1 ... / ...
  numerator polynomial type? > b
  sptrf f1 < s1 b / ...
  denominator polynomial type? > a
ML: bode
>bode f1
```




```
Either do ML identification using the command MLID or
start the cross validation using the command CUT.
The first half you CUT and TRENDed from in was inc,
the first record 1, the number of records 100
and removed trends of order 0.
ML: mlid, cut                    " Either estimation or cross validation.
>mlid s2 2

 CONVERGENCE (DV/V<  1.8E-06)  " Printout from Idpac.
 ****************************

 A1             -0.199583      +-  0.294728
 A2             -0.624053      +-  0.278128
 B1              6.788077E-02  +-  7.090448E-02
 B2              0.151931      +-  6.384738E-02
 C1             -1.944228E-02  +-  0.269345
 C2             -0.716464      +-  0.207658

 LAMBDA          0.322308      +-  2.279064E-02
 LOSS FUNCTION   5.19413
 AIC             69.3384
```

```
ML: residu
>residu r2
```

```
VARIANCE OF THE RESIDUALS:  0.103665

NUMBER OF CHANGES OF SIGN
OF THE RESIDUALS:    51

5 PERCENT TOLERANCE LIMITS:    39    59

TEST OF INDEPENDENCE OF THE RESIDUALS

E(RES(T)*RES(T+TAU)) FOR: 0<TAU<   11

TEST QUANTITY:   2.55587
DEGREES OF FREEDOM:    10

TEST OF NORMALITY

TEST QUANTITY:   21.0357
DEGREES OF FREEDOM:    17
```

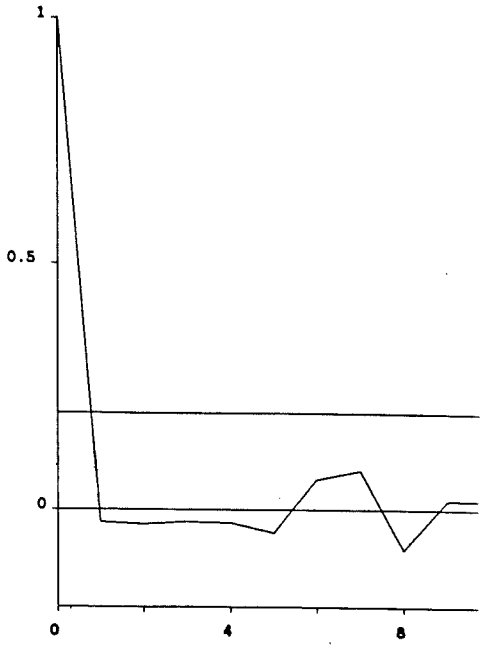



```
Either do ML identification using the command MLID or
start the cross validation using the command CUT.
The first half you CUT and TRENDed from in was inc,
the first record 1, the number of records 100
and removed trends of order 0.
ML: mlid, sptrf, cut
>mlid s3 3

 MAXIMUM NUMBER OF ITERATIONS REACHED    " Printout from Idpac.
 ************************************

 A1            -0.362876
 A2            -0.539601
 A3             8.183430E-02
 B1             4.421177E-02
 B2             8.038352E-02
 B3             7.452730E-02
 C1            -0.230977
 C2            -0.696315
 C3             0.146590
```

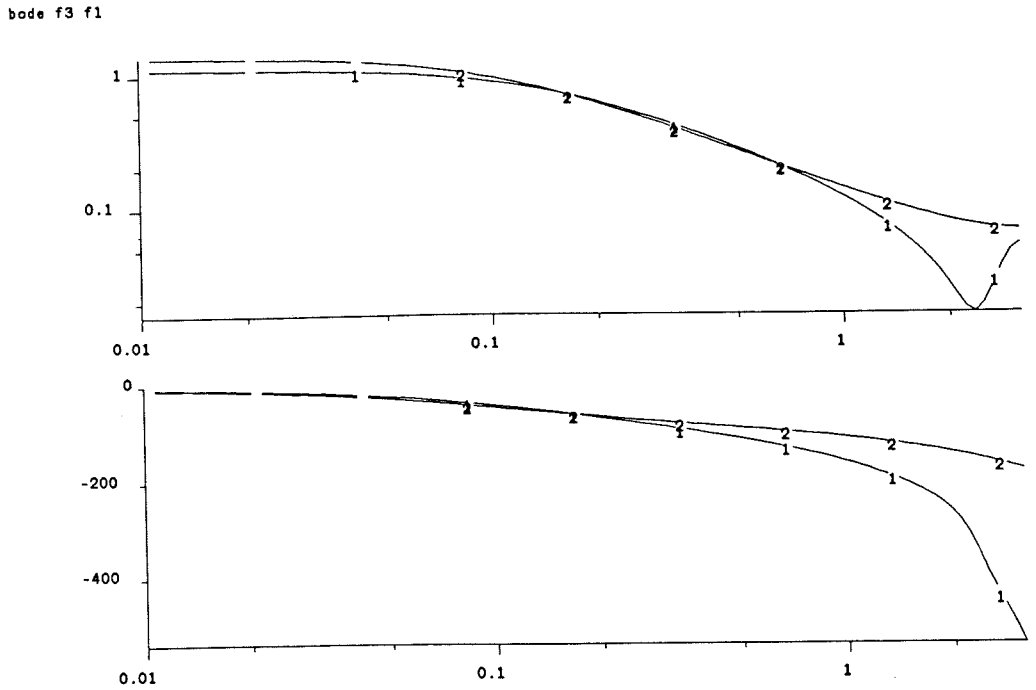
```
LAMBDA          0.317761     +-  2.246908E-02

LOSS FUNCTION   5.04860

AIC             72.4964
```

```
ML: residu

>residu r3
```

```
VARIANCE OF THE RESIDUALS:  0.100594

NUMBER OF CHANGES OF SIGN
OF THE RESIDUALS:    51

5 PERCENT TOLERANCE LIMITS:    39    59

TEST OF INDEPENDENCE OF THE RESIDUALS

E(RES(T)*RES(T+TAU)) FOR: 0<TAU<   11

TEST QUANTITY:   2.19629
DEGREES OF FREEDOM:    10

TEST OF NORMALITY

TEST QUANTITY:   15.9003
DEGREES OF FREEDOM:    17
```




```
ML: mlid, sptrf, cut

>sptrf f3 < s1 b / a

ML: bode

>bode f3 f1
```




```
Either do ML identification using the command MLID or
start the cross validation using the command CUT.
The first half you CUT and TRENDed from in was inc,
the first record 1, the number of records 100
and removed trends of order 0.
ML: mlid, cut
>cut                          " Start cross validation,
                              " cut second half of in.
  cut ... < in ... ...
  outfile? > inx
  cut inx < in ... ...
  first record? > 100
  number of records? > 100
ML: trend                     " Remove trends.
>trend
  trend ... < inx ...
  outfile? > inxt
  trend polynomial order? > 0
The first half you CUT and TRENDed from out was outc,
the first record 1, the number of records 100
and removed trends of order 0.
ML: cut                       " Cut second half of out.
```

```
>cut outx < out 100 100
ML: trend
>trend outxt 0
ML: residu
>residu                          " Compute residuals of the second half.
  residu ... < ... inxt outxt
  outfile? >rx1
  residu rx1 < ... inxt outxt
  system file?> s1
ML: vecop
>vecop                           " Square the residuals.
  vecop ... < rx1 ... rx1
  outfile? > v
  operator? > *
ML: stat
>stat                            " Compute the square sum.

 SUM      =   16.8306751        " <--
 MEAN     =  0.168306753
 VARIANCE =  8.684720099E-02
 ST.DEV.  =  0.294698477
 MINIMUM  =  4.636161748E-07 AT ROW     22
 MAXIMUM  =   2.38813901     AT ROW      1
 LENGTH   =    100

  What is the value of the loss function? > 16.83  " E.s. question

The system with least loss function is s1
estimated with order 1.
ML: residu, stop
>residu rx2 s2
ML: vecop
>vecop v *
ML: stat
>stat

 SUM      =   16.5365925        " <--
 MEAN     =  0.165365934
 VARIANCE =  9.021462500E-02
 ST.DEV.  =  0.300357491
 MINIMUM  =  4.303435344E-05 AT ROW     22
 MAXIMUM  =   2.38813901     AT ROW      1
 LENGTH   =    100

  What is the value of the loss function? > 16.54  " E.s. question

The system with least loss function is s2,
```

```
estimated with order 2.
ML: residu, stop
>residu rx3 s3
ML: vecop
>vecop v *
ML: stat
>stat

 SUM      =   16.9050331       " <--
 MEAN     =  0.169050336
 VARIANCE =  0.104850508
 ST.DEV.  =  0.323806286
 MINIMUM  =  6.266986929E-06 AT ROW     98
 MAXIMUM  =   2.38813901     AT ROW      1
 LENGTH   =    100

  What is the value of the loss function? > 16.90  " E.s. question

The system with least loss function is s2,
estimated with order 2.
ML: residu, stop
>stop
```

Experiences of the Knowledge Database Example

It is a major undertaking to write a reasonably large and complete knowledge database, even for a single task like maximum likelihood identification. For this reason we decided that the development of a full knowledge database was to be left out of the project. But, as a database is needed anyway in order for the expert interface to run, at least a small database had to be built.

A version of a maximum likelihood script has been developed. It is primarily intended for demonstration purposes. The script is about 200 lines long and the associated rule base contains some 60 rules. It is only concerned with system identification using the maximum likelihood algorithm and knows nothing about least squares estimation, correlation analysis, etc. Also, it does not fully cover maximum likelihood estimation. But we believe it to show that it is indeed possible to build a realistic knowledge database for use in an expert interface.

For this thesis a smaller version of the ML-script was developed. It contains all interesting features of the larger script. The main difference is that neither does the smaller version allow as many different ways of doing things, nor does it collect facts about the session or cares for if anything goes wrong. But as the larger script does not really give any extra insight in how the expert interface works, we decided not to present it here. A detailed description of the larger database is found in Larsson and Persson [1987 c].

During the development of these examples we have gained some experience of knowledge engineering. Probably the greatest problem is to find and talk to experts. Either one does not find any expert that is good enough, or the expert does not have any time for the effort

needed. We have not made any serious attempts to interview experts in the project. As our efforts have had a limit, our own expertise has been enough.

It soon became apparent that it takes quite some time to build even a small database. Rules interact with each other and the output may sometimes look messy. The output from the expert system can be very hard to predict just by looking at the scripts and rules. This forces the expert and the knowledge engineer to run through the scripts many times under different conditions during development, and change the scripts, rules and their order. This is one reason why the process of writing scripts and rules is very time consuming.

5

Further Developments

Der Satz ist eine Wahrheitsfunktion der Elementarsätze.
(Der Elementarsatz ist eine Wahrheitsfunktion seiner selbst.)

All the things described earlier in this thesis have been implemented, as described in chapter 3. But of course there are several ideas that remain to be investigated and tested. Some of these ideas will be described in this chapter.

Different Designs of a Script Matcher

The script matcher is a central part of the expert interface. It is the piece of code that takes care of matching incoming commands against the script database, and updating the scripts in the process. Several different ways of implementing this device were suggested during the project.

One way of building a script matcher is to define a language for scripts and the operations needed, and then to write one or several Lisp functions for every operation. This was the solution actually used in the project and it is described in chapter 3.

Expert system shells such as YAPS have a rather advanced pattern matcher built in. This facility might be used for matching and transforming scripts. In this way the script language is still there, but the script matcher is implemented by production rules instead of Lisp functions.

The simplest and most straight-forward way of implementing a script matcher is to use production rules to keep track of the user by state variables in the fact database of the rule system. In this way the implementation becomes very simple. But the script language would be lost and the scripts only implicitly stated in the rules describing the state transitions. Loops and alternatives would be easily described, but more complicated and computer language-type constructs, e.g., parameterized script macros, are hard to deal with. The knowledge engineer would have to take care of such things himself by using global state variables and similar tricks.

A fourth solution would be to describe the scripts as an explicitly stated language and then transform it into a rule driven state transition machine. In this way one would be able to use the script language when writing scripts and still get a very simple implementation of the script matcher. The trouble with this solution is of course to develop the script-to-rules compiler. Most of the difficulties that will bother the knowledge engineer in the simple, "nothing but rules" solution would have to be taken care of in the automatic translation of scripts to rules.

In conclusion, the solution used in the project is working properly and the code is easily understood and quite small, only a few pages. For this reason the alternatives seem to be either insufficient or unnecessarily complicated.

Integrating the Interface with a CACE Program

During the entire project there has been a focus on ideas and general facilities for help systems and we have tried to avoid Idpac specific things as far as possible. This means, among other things, that most of the ideas in the system would work with other types of communication as well as with the Idpac command dialog.

The most important alternative is probably a windows and mouse communication, as used in e.g., the Macintosh. Once the mouse-clicks have been translated to a stream of commands, the same expert interface would work fine. The commands to the computer produced by a window and mouse system are of course simpler than commands to Idpac, as they probably have no or few arguments, but this makes the task simpler for the expert interface. The help system would have to know about windows and mouse-clicks instead of commands and the output routines are also easily adapted to a window system.

The Idpac language is somewhat old-fashioned and has several shortcomings. Some commands have a syntax different from the usual one and there is quite a lot of special cases, some of which have the typical look of "programming tricks." Part of an explanation is that Idpac has been rebuilt and added to in several steps. It would be much easier to build an expert interface for Idpac if some changes were made in Idpac itself.

Many commands have their own special syntax, e.g., the `plot` command. Altogether, there is a lot of special cases to take care of when writing a parser for Idpac. A better command language would have a simple syntax with no exceptions, even at the expense of the user having to type a little more. The facilities of the help system would greatly ease the user's work anyway. So a first thing to do about the Idpac language would be to throw out all special tricks and exceptions.

A command language should have a simple and uniform syntax, making it easy both to learn and to write parsers for. The mere fact that it is possible to define a grammar for a language does not mean that it is very good in the above respects. So the next thing to do would be do redesign the command syntax in a more uniform way.

Idpac is designed as a stand-alone program with a top level that is only useful for communication with a user at a terminal. There are no facilities that enable another program to get values of parameters, results, or error messages from Idpac. The only way that the expert interface can get such data is either to analyse the resulting data files or to ask the user. Neither of these alternatives are really satisfying. Currently, the expert interface cannot even find out if Idpac has had an error without asking the user. If a user specifies a column number in a file as being 10000 say, the interface just sees an integer and accepts it, but Idpac

will not. It is very difficult to make sure that the interface will never send anything that would cause an error in Idpac. Therefore the help system must have the possibility to check for errors in Idpac. This could be done by supplying functions in Idpac whereby values could be sent to another program, making it easy to transfer results to the expert interface. This would have the beneficial effect of heavily reducing the number of questions asked by the help system. Most questions concern results of different operations, and these could instead be read into the expert interface automatically.

Idpac is built around Intrac, a communication module which handles numeric subroutines interactively, Wieslander and Elmqvist [1978]. Intrac can decode and handle commands typed by a user. Intrac has routines for input, output, and file handling, it has a subroutine facility and some other programming constructs, e.g., loops, goto statements, and if statements. In expressive power Intrac can be compared with Basic.

The best way of making Idpac suited for connection to other programs is probably to redesign it in the following way. There should be a kernel of numerical routines, written in Fortran or C, with well defined data structures and calling conventions. On top of this there would be a small command decoder. The command language of this parser would be very simple and low level, with no possibilities of defaults, etc. It might be e.g., a stack-oriented language, Åström and Schönthal [1982]. On this, a top level with command parsing could then be built. A possible choice for this top level would be a Lisp system, where all the interactive facilities of Lisp could be used in a CAD package. Of course the expert interface could be used as the top level program if it was to be included in the new Idpac, eliminating the need for duplicating the parsing and the use of inter-process communication via VMS mailboxes. This project would be a major effort in itself, and it would be the logical step to take after the design of a help system as described in this thesis.

When the expert interface is to be built into a CACE program, there may be reasons for implementing it in some other language than Lisp, e.g., Fortran, C, or Pascal. We believe that once it is known what features should be available, implementing an expert interface in C, say, is no problem. It will not be very easy to make big changes in the code, and the size of it will grow considerably. However, if the CACE program would use Lisp as its interactive user interface the incorporation of an expert interface would be very straight-forward. At the present stage the expert interface is far from finished and it would be a big mistake to leave the Lisp environment.

Additional Features in the Design

The expert interface is a result of a limited project of largely experimental programming. A full fledged implementation of a system for production use is far beyond the scope of our efforts. By necessity a few things have been left out from the actual implementation. These details have a certain interest, though, and will therefore be given a short description.

The Idpac command language has a few unusual and somewhat archaic constructs. In order to enable aggregation of several data signals, a data file may contain several columns, i.e., filenames is indexed with cardinal numbers, as a way of denoting different signals. In more modern programs concepts like this is missing. Certainly there is a need for some mechanism of data aggregation, but we very strongly feel that the concept of columns is not the solution. Taking care of columns in the command parsing proved to be complicated and the decision was made to skip it altogether. The expert system encourages the user to keep

his data in different files. The few commands that demand data to reside in certain columns where replaced with suitable macros with separate files as arguments.

The command parser of the expert system allows for short forms of the commands, as long as they are not ambiguous. A natural extension would be a general support of short forms, for filenames and parameters. In the case of filenames, the implementation might give some practical difficulties, since the system must use operating system calls to fetch information about the files in the current directory, etc. Since the implementation of this kind of facility is only a technical matter, we have left it out. Still, it is important to realize that an intelligent help system should have functions like this, making the use of it both more efficient and easy.

During the startup of the system, a part of the code reads the script database and performs a rough check for syntactical correctness. These functions could, with no large effort, be substantially developed. Presently, script clauses are checked for their type word and parameter clauses only. There is no consistency checking of the production rules. A full fledged script and rule verifier would in addition contain a more complete check of the syntax of each type of script clause, a simple semantic check of the use of internal file names, a consistency check of the production rules and a semantic check of the use of facts, both in the scripts and rules. For example, a filename should not be used in a `kscall` before the file has been created with a command.

The scripts and rules are currently entered in almost the same form as they have internally in the Lisp system. The syntax is very Lisp-like, and probably takes some time to get used to. An expert interface should have facilities for reading and printing the knowledge database in a more easily readable form, in order to make life easier for the knowledge engineer not so familiar with Lisp. The current scripts and rules are not that difficult to read though, so we did not bother to do anything about it.

The user of Idpac has to make up a lot of new filenames. In order to avoid confusion, these names must be well chosen. Automatic creation of file names would be a nice facility. In order to make this name giving more intelligent, information about name giving conventions could be kept in the scripts. In this way the system could suggest names depending not just on what command that created the file, but also on where in the identification procedure the file was created. But it is not clear whether this strategy would suffice to enable really good naming conventions. Not being a crucial part of an intelligent help system, such a facility was not implemented.

The documentation of sessions with Idpac is very important. The rule system of the expert interface can take care of this. Currently, rules that write a text string to a file are available in all scripts. When designing a script the knowledge engineer could use these rules to implement automatic documentation. This would involve putting facts about all important and successful sub-goals in the fact database, and using this information to produce a short description of the session, what has been and what has not been done, the reasons for trusting or distrusting certain results, and so on. The documentation of a session depends heavily on the structure of that session and general strategies tend to be rather simple-minded. Therefore we believe that automatic documentation should be done with the production rule system. This has, however, not been implemented.

In order to help the knowledge engineer to get new ideas for scripts, the expert interface should keep traces of all runs and also collect statistics of commands, scripts, and common errors. This would help to pinpoint deficiencies in both the interface and Idpac and give feedback for changes in the programs. It would also provide input for the design of new

command languages and program packages. We decided that the gathering of statistics was not very central to the project, and also of a rather trivial nature to implement. Thus it is not currently done in the system.

Something that is often very useful to have, but also very hard to implement is the possibility to undo different operations. When running the expert interface the user will certainly sometimes want to retract a command and back up the script matcher. Some cases of this may be handled simply by keeping a copy of the previous states, but there is currently no support for this in the system. Once again, we decided that this was not very central to showing how an expert interface should work.

The system currently supports two different user states, the very quiet expert state and the more verbose beginner state. There may be a need for one or more additional modes or settings. Experience will show this. A simple adaptive scheme for changing these modes might also be successful. A user making many errors in expert mode might be put into the safer beginner mode automatically. A strategy like this must allow a user to work outside of the scripts though, as he very well may be an expert doing things not in the scripts. In this case the command histories provide excellent material for new scripts. There is nothing of this kind in the current system.

One could think of even more advanced forms of adaptivity, e.g., the system learning new scripts on-line from an expert and automatic correction of errors, both syntactical errors and maybe also errors detected via script matching. The expert interface could throw in a few commands that it thought that the user forgot, etc. An interesting idea is the notion of "buggy scripts." This means that the knowledge engineer enters erroneous scripts into the database. They would be descriptions of common errors and of methods from the grey zone between correct and wrong. Thus the system would be able to give better diagnosis when the user errs. But this kind of things are difficult to outline, and even more difficult to implement.

A philosophy of design has been that the important thing is to show the user information, not to help him type it. Many advices are of the type "Now perform a `plot` command." and it would of course not be very difficult to supply defaults, so that the system would take a single carriage return as accepting the advice and issuing, in the case above, the `plot` command. Currently the system does not do very much of this kind, but it is surely straight-forward to implement if one decides to do so.

6

Conclusions

Die allgemeine Form der Wahrheitsfunktion ist: $[\bar{p}, \bar{\xi}, N(\bar{\xi})]$.
Dies ist die allgemeine Form des Satzes.

This thesis describes an intelligent help system for Idpac. The general idea of building a help system based on expert system techniques and including it in a CAD program is presented and a solution is given. This solution includes the development of *scripts*, a data type for describing sequences, and a description of the implementation of the expert interface. An example of a knowledge database and an actual session with the system is also shown.

First, the idea of an intelligent help system is presented. An outline of how to combine an expert system and a CAD program is given and the demands of an intelligent user interface are stated.

The need for handling sequences is stated and a data type for doing this, *scripts*, is introduced. The concept of scripts is then given a close description and several examples of scripts are shown.

A large part of the ideas presented in this thesis has been implemented in an experimental system. This implementation is discussed and the different parts of the program are described in detail.

The knowledge database and its contents are described and an example of a script and production rules is shown. An example of a session with the system is also given.

There is a discussion of extensions to and further developments of the system. Some shortcomings of Idpac are listed and an alternative way of design of a future system is outlined.

The main conclusion of the project is that it is indeed possible to use an expert system and still retain a command style dialog. A second conclusion is that not all knowledge in a database for system identification using Idpac need be implemented by production rules. Scripts are a better way to represent sequences, particularly in problems where both methods and goals are well known. A good rule is to use as much as possible of the structure of the problem in the solution. The use of scripts supported by rules in a forward chaining strategy will probably reduce the overall size of the knowledge data bases considerably.

Some experience have been gained with the system. The current knowledge database is to small to be of any use for an expert. For a non-expert it may be useful. Even the expert might find it nice to use the system though. It is quite clear that it is very convenient not to be bothered with all the details of running Idpac, as remembering file names, trivial next-commands, etc. This alone is a sufficient reason for having an intelligent help system. It must be pointed out, however, that the current system is easily fooled and in no way failsafe. A third conclusion is that an intelligent help system such as the one described in this thesis is clearly useful for all, except maybe for expert users.

7

References

Wovon man nicht sprechen kann, darüber muß man schweigen.

AHO, A. V., SETHI, R and J. D. ULLMAN (1986): *Compilers, Principles, Techniques and Tools*, Addison-Wesley, Reading, Massachusetts.

AKAIKE, H. (1972): "Use of an Information Theoretic Quantity for Statistical Model Identification," *Proceedings of the 5th Hawaii International Conference on System Science*, Honolulu, Hawaii.

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.

BIRDWELL, J. D *et al* (1984): *Issues in the Design of a Computer-Aided System and Control Analysis and Design Environment*, ORNL/TM–9038, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

BIRDWELL, J. D *et al* (1985): "CASCADE: Experiments in the Development of Knowledge-Based Computer-Aided Systems and Control Analysis and Design Environments," *Proceedings of the 2nd IEEE Control Systems Society Symposium on Computer-Aided Control System Design*, Santa Barbara, California.

BIRDWELL, J. D., J. R. B. COCKETT and J. R. GABRIEL (1986): "Domains of Artificial Intelligence Relevant to Systems," *Proceedings of the 1986 American Control Conference*, Seattle, Washington.

BOBROW, D. G. and T. WINOGRAD (1977): "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science*, 1, No. 1, 3.

BROWNSTON, L. *et al* (1985): *Programming Expert Systems in OPS5: An Introduction to Rule-Based programming.*, Addison-Wesley, Reading, Massachusetts.

CHARNIAK, E., C. K. RIESBECK and D. V. MCDERMOTT (1980): *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

CLOCKSIN, W, F. and C. S. MELLISH (1981): *Programming in Prolog*, Springer-Verlag, Berlin.

COX, D. R. (1958): *Planning of Experiments*, John Wiley & Sons, New York.

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

EYKHOFF, P. (1974): *System Identification, Parameter and State Estimation*, John Wiley & Sons, London.

EYKHOFF, P. (1981): *Trends and Progress in System Identification*, Pergamon Press, Oxford.

FEDOROV, V. V. (1972): *Theory of Optimal Experiments*, Academic Press, New York.

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

FORGY, C. L. (1981): "OPS5 User's Manual," Technical report CMU–CS–81–135, Carnegie Mellon University, Pittsburgh, Pennsylvania.

GALE, W. A. and D. PREGIBON (1982): "An Expert System for Regression Analysis," *Proceedings of the 14th Symposium on the Interface*, Troy, New York, July 5–7, Springer Verlag, New York, pp. 110–117.

GALE, W. A. and D. PREGIBON (1983): "Building an Expert Interface," Bell Telephone Laboratories, Murray Hill, New Jersey.

GUSTAVSSON, I. and A. B. NILSSON (1979): "Övningar för Idpac," Technical report, TFRT–7169, Department of Automatic Control, Lund Institute of Technology, Lund.

GUSTAVSSON, I. (1979): "Några macros för Idpac," Technical report, TFRT–7170, Department of Automatic Control, Lund Institute of Technology, Lund.

HARMON, P. and D. KING (1985): *Expert Systems, Artificial Intelligence in Business*, John Wiley & Sons, New York.

HAYES-ROTH, F. (1978): "The Role of Partial and Best Matches in Knowledge Systems," in D. A. Waterman and F. Hayes-Roth (Eds.): *Pattern-Directed Inference Systems*, Academic Press, New York.

HAYES-ROTH, F., D. WATERMAN and D. LENAT (1983): *Building Expert Systems*, Addison-Wesley, Reading, Massachusetts.

JAMES, J. R., J. H. TAYLOR and D. K. FREDERICK (1985): "An Expert System Architecture for Coping with Complexity in Computer-Aided Control Engineering," *Preprints of the 3rd IFAC/IFIP International Symposium*, The Technical University of Denmark, Lyngby, Copenhagen, pp. 47–52.

JOHNSON, S. C. (1975): "Yacc—Yet Another Compiler Compiler," Computing Center Technical Report No. 25, Bell Telephone Laboratories, Murray Hill, New Jersey.

LARSSON, J. E. (1984): *An Expert System Interface for Idpac*, Master thesis, TFRT–5310, Department of Automatic Control, Lund Institute of Technology, Lund.

LARSSON, J. E. and K. J. ÅSTRÖM (1985 a): "An Expert System Interface for Idpac," *Proceedings of the 2nd IEEE Control Systems Society Symposium on Computer-Aided Control System Design*, Santa Barbara, California.

LARSSON, J. E. and K. J. ÅSTRÖM (1985 b): "An Expert Interface for Idpac—Paper Presented at Santa Barbara '85," Technical report, TFRT–7308, Department of Automatic Control, Lund Institute of Technology, Lund.

LARSSON, J. E. and P. PERSSON (1986 a): "Ett expertsystemsnitt för Idpac, (An Expert System Interface for Idpac)," *SAIS '86*, The Swedish AI Society's Annual Workshop, Linköping, April 24–25, 1986.

LARSSON, J. E. and P. PERSSON (1986 b): "Knowledge Representation by Scripts in an Expert Interface," *Proceedings of the 1986 American Control Conference*, Seattle, Washington.

LARSSON, J. E. and P. PERSSON (1986 c): "Knowledge Representation by Scripts in an Expert Interface—Paper Presented in Seattle 1986," Technical report, TFRT–7332, Department of Automatic Control, Lund Institute of Technology, Lund.

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): "An Expert Interface for Idpac—Reference Manual," Technical report, TFRT–7341, Department of Automatic Control, Lund Institute of Technology, Lund.

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

LESK, M. E. (1975): "Lex—A Lexical Analyzer Generator," Computing Center Technical Report No. 39, Bell Telephone Laboratories, Murray Hill, New Jersey.

LJUNG, L. and T. SÖDERSTRÖM (1983): *Theory and Practice of Recursive Identification*, MIT Press, Cambridge, Massachusetts.

NII, H. P. and N. AIELLO (1979): "AGE: A Knowledge-Based Program for Building Knowledge-Based Programs," *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, William Kaufmann Inc., Los Altos, California, pp. 645–655.

RIMVALL, M. and L. BOMHOLT (1985): "A Flexible Man-Machine Interface for CACSD Applications," *Preprints of the 3rd IFAC/IFIP International Symposium*, The Technical University of Denmark, Lyngby, Copenhagen, pp. 98–103.

SCHANK, R. C. and R. P. ABELSON (1977): *Scripts, Plans, Goals and Understanding*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

SCHANK, R. C. and C. K. RIESBECK (1981): *Inside Computer Understanding*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

STEFIK, M. *et al* (1982): *The Organization of Expert Systems—A Prescriptive Tutorial*, Palo Alto Research Centers, Palo Alto, California.

TAYLOR, J. H. and D. K. FREDERICK (1984): "An Expert System Architecture for Computer-Aided Control Engineering," *IEEE Proceedings*, 72, 1795–1805.

TAYLOR, J. H., D. K. FREDERICK and J. R. JAMES (1984): "An Expert System Scenario for Computer-Aided Control Engineering," *Proceedings of the 1984 American Control Conference*, San Diego, California, pp. 120–128.

VAN MELLE, W. *et al* (1981): "The EMYCIN Manual," Technical report HPP–81–16, Computer Science Department, Stanford University, Palo Alto, California.

WATERMAN, D. A. (1986): *A Guide to Expert Systems*, Addison-Wesley, Reading, Massachusetts.

WATERS, R. C. (1985 a): "KBEmacs: A Step Toward the Programmer's Apprentice," Technical Report 753, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts.

WATERS, R. C. (1985 b): "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transactions on Software Engineering* **SE–11 No. 11**, November 1985, 1296–1320.

WEISS, S. *et al* (1982): "Building Expert Systems for Controlling Complex Programs," *Proceedings of the National Conference on Artificial Intelligence*, William Kaufmann Inc., Los Altos, California, pp. 322–326.

WELIN, C. W. and R. SKAGERWALL (1986): "Design of an Expert System and Man-Machine Interface for Operation and Maintenance of AXE Telephone Exchanges," *Proceedings of the IEEE 1986 International Zürich Seminar on Digital Communications*, Zürich, 11-13 March, Verlag der Fachvereine an den Schweizerischen Hochschulen und Techniken, Zürich.

WIESLANDER, J. and H. ELMQVIST (1978): "INTRAC a communication module for interactive programs. Language manual.," 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*, Zurich.

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

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

WILENSKY, R. *et al* (1986): "UC—A Progress Report," Report No. UCB/CSD 87/303, Computer Science Division (EECS), University of California, Berkely, California.

WINSTON, P. H. and B. K. P. HORN (1981): *Lisp*, Addison-Wesley, Reading, Massachusetts.

WIRTH, N. (1976): *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey.

WITTGENSTEIN, L. (1922): *Tractatus Logico-Philosophicus*, Routledge & Kegan Paul Ltd., London.

ÅSTRÖM, K. J. and P. EYKHOFF (1971): "System Identification—A Survey," *Automatica* **7**, 123–162.

ÅSTRÖM, K. J. (1980): "Maximum Likelihood and Prediction Error Methods," *Automatica* **16**, 551–574.

ÅSTRÖM, K. J and T. SCHÖNTHAL (1982): "PCALC—A Polynomial Calculator—A User's Manual," Department of Automatic Control, Lund Institute of Technology, Lund.

ÅSTRÖM, K. J. (1983 a): "Computer-Aided Modeling, Analysis and Design of Control Systems—A Perspective," *IEEE Control Systems Magazine*, No. 2, May 1983, 4–16.

ÅSTRÖM, K. J. (1983 b): "Modeling and Simulation Techniques," *Agard Lecture Series*, No. 128.

ÅSTRÖM, K. J. (1985): "Computer-Aided Tools for Control System Design," in Jamshidi, M. and C. J. Herget (Eds.): *Computer-Aided Control Systems Engineering*, North-Holland, Amsterdam.