CODEN: LUTFD2/(TFRT-5336)/1-80/(1985)

EXPERIMENTS WITH USER INTERFACES FOR COMPUTER AIDED CONTROL ENGINEERING

Lena Peterson

Department of Automatic Control

Lund Institute of Technology

September 1985

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* MASTER THESIS *Date of issue* September 1985 *Document Number* CODEN: LUTFD2/(TFRT-5336)/1-80/(1985)
Author(s) Lena Peterson	*Supervisor* Karl Johan Åström *Sponsoring organisation*

Title and subtitle

Experiments with User Interfaces for Computer Aided Control Engineering

Abstract

The development of new user friendly computer tools for Computer Aided Control Engineering requires new user interfaces which are more powerful and easier to use.

The idea of using an existing interactive symbol manipulating language such as LISP is investigated. Some existing interface modules used today in CACE programs are described and compared. The history behind LISP and the general structures in LISP are covered. Formal polynomial arithmetics is used as an example of formula manipulation. Represention of polynomials is discussed. A small package for polynomial arithmetics, written in LISP, is presented. The symbol manipulation program MACSYMA is used for investigation of the possibilities of a symbol manipulation features. MACSYMA is briefly described and a small package for polynomial synthesis, written in MACSYMA, is presented.

Finally some conclusions are drawn on the use of an interactive language such as LISP as user interface.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title		*ISBN*
Language English	*Number of pages* 80	*Recipient's notes*
Security classification		

CONTENTS

CHAPTER 1

Introduction

Extensive research projects on computer aided control systems design (CACSD) have been carried out during the last fifteen years at the Department of Automatic Control at Lund Institute of Technology. These resulted in several programs which have been used around the world. All these programs use a common interactive communication module, INTRAC, as user interface. The purpose of this thesis is to investigate the possibilities, advantages and disadvantages of exchanging INTRAC with an input processor based on symbol manipulating language such as LISP.

The work was carried out as follows. Some existing CACSD systems were first investigated, their input command structure was described by EBNF notation and compared. Formal and numerical manipulation of polynomials suit as an example for implementation of a minor experimental system. Some experience of LISP was obtained and representation of polynomials was investigated. A small polynomial package was written in LISP. MACSYMA, a formula manipulation programs was then used when writing a a small package for formula manipulation.

Chapter 1 introduces the subject. In Chapter 2 different kinds of interactive programs are described. Two different interactive programs Intrac and Ctrl-C are compared. The history behind LISP and the general structures used in LISP are covered in Chapter 3. General ideas on representation of polynomials are described in Chapter 4 and in Chapter 5 a small Lisp package for formula manipulation is described. Chapter 6 and 7 concerns Macsyma, a formula manipulation program. In Chapter 7 a package written in Macsyma for polynomial synthesis is presented. Finally, in Chapter 8, some conclusions are drawn considering the use of LISP system as user interface and the use of Macsyma.

CHAPTER 2

Implementation of Interactive Programs

Ideas and Examples

HISTORY

In the early days of the computer age persons involved with computer calculations were experts both in the calculations to be performed and in the computer programs to be written and run. The user could run his programs himself on the computer and could check the results and make corrections immediately. Later on large computer centres were created. The user no longer had direct contact with the computer. All computations were performed in batch i.e. all data was submitted in a batch, the program started and run without interaction with the rest of the world. Those were the days of the cardpunches and paper tapes.

By the Seventies the CRT terminal replaced the teletype used in the Sixties as the common interface between the user and the computer. This gave birth to a new kind of programs - interactive programs. The user interacts with the computer in order to achieve a solution to the problem. In the Eighties the personal computer and the concept of the workstation emerged. This brings the user back to the original position, in charge of his own machine, but now with much more powerful and user friendly tools at hand.

CONSIDERATIONS

When designing an interactive program package one must take into consideration that there will be several kinds of users of the system:

The one-time user may, for example, be a student solving a laboratory exercise.

In this case a fairly simple and rigid system would be preferred for solving a well-defined problem. The user is not interested in anything but the elementary facilities needed for his task.

The beginner is in the same situation as the one-time user. He wants simplicity to get started. But he also wants to become an advanced user one day. A facility of rapid help and instruction is what he needs.

The experienced user already has prior knowledge, skill and intuition. He requires a system which gives him a maximum of freedom. It is important to see the result of one step and to have the ability to make further steps directly.

MEANS OF INTERACTION

Question and Answer

The question and answer method is a commonly used technique of interaction. The programs asks questions regarding parameters, values and other information that is needed to solve the task. These questions are answered by the user. There may also be questions regarding the future steps of the program. For example in choosing from a list of procedures to be run next.

This kind of interaction gives good guidance to the one-time user and allows checks to be made on the answers to the questions to ensure security. The experienced user however might find this kind of interaction tedious. It restricts his choices and limits use of his knowledge of the subject.

Menu Driven Programs

Menu driven interaction is another method which is very much related to the question and answer approach. In this case the program becomes less rigid than in the general question and answer program since there are more answers to choose between. Menus do of course provide excellent guidance to the inexperienced user but as soon as he becomes familiar with the program the routine going through the menus will seem tedious and constraining. One reason why menu driven programs have become so popular is that they are very easy to write since only one big if-then-else structure for each menu is needed.

Command Driven Programs

Commands is another way to interact with the computer. The user submits a commandline telling the computer what to do next. A command usually consists of a keyword together with values and flags specifying what action to be taken. Here the the experienced user has full freedom to do what he choses, whereas the one-time user is left on his own.

Command Languages

A more versatile and advanced variant of command driven dialog is the command language. To the user, this kind of interaction may not look very different from command driven programs, however the commands are defined in a grammar for a language (the command language). A parser checks that a command submitted to the program is correct according to the grammar. This allows more flexible command constructions and permits different syntax for different commands.

Extra Features

If the user is given the possibility to create his own commands he can easily (or not so easily depending on the function structure) create his own tools. The easiest way to define commands is by means of macros. When a macro is called, the text of the macro is submitted to the command interpreter and is interpreted as if it was input directly by the user on the terminal.

To aid the one-time user and the beginner it is suitable to include extensive help-functions. These should give descriptions of all existing commands but also if possible general guidance to the in-experienced user. When new to a system, it is not always easy to remember the name of the commands.

Graphics facilities have recently been included in common computer systems. Newly developed graphic terminals in combination with increased computer speed and memory size has made it possible to use graphics features without increasing the response time too much. Windows and pop-up menus are commonly used features. These can aid the beginner by letting him see both help instructions and the working-area at the same time. All different kinds of symbols and small drawings can of course be used to aid the inexperienced but it is important not to trifle away the basic principles and bore the experienced user. Extra help should be optional but easy to access.

EXAMPLES

Two interactive modules commonly used in design of automatic control are INTRAC and Ctrl-C. To get some idea of what interactive programs used today can look like these will be described. EBNF (Extended Backus Normal Form) is used for describing their syntaxes.

EBNF

EBNF is a formal language that is used for describing context-free grammars. EBNF is an extension of the Normal Form designed by John Backus during the development of Algol 60. There are three kinds of symbols in EBNF. The first kind are the symbols of the grammar to be described. These are called terminal symbols. The second kind are symbols which are names of parts of the language, so called nonterminal symbols. The third group are metasymbols used in the rules. The grammar is described by a set of production rules. On the left side of each rule there is a nonterminal symbol. The right-hand side can be a arbitrary string of terminal and nonterminal symbols. Metasymbols that are used in EBNF are ::= | ( ) * and +. The symbol $ denotes an empty production. | means 'or' and parentheses are used to group symbols together. the symbol * means that the previous symbol or group of symbols may be included none or any other number of times. The symbol + means the same as * with the difference that the previous symbol or group of symbols must be included at least once.

INTRAC

Intrac is a interactive language developed at the Department of Automatic Control at Lund Institute of Technology, described in Wieslander & Elmqvist (1978). It is employed as a user interface in several programs developed at the same department for example Idpac, used for identification, Modpac, which is an interface program, Synpac used for synthesis, and Simnon which is a simulation program.

Intrac is totally command driven, i.e., action is initiated by the user by giving a command. There are two modes, in normal mode (or command mode) commands are entered from the keyboard of the users terminal. In this mode almost any command is legal though not necessarily meaningful.

The second program mode is the macro mode. In this mode the commands are read from a special macro file. There are special commands to be used in macro mode which allow looping testing and jumping such as if-goto and for-next.

In the macro mode, a special input command (READ) is available. Together with the above mentioned special commands it allows the user to write functions where parameters can be entered when the function is run. Alternate steps in the function can also be chosen by answering questions.

A command has this generic form:

```
   CMND LARG1 ... LARGNL < RARG1 ... RARGNR
```

EBNF description of Intrac

```
command :== 'MACRO' macro_identifier (formal_argument | delimiter |

                termination_ marker )+ |

                'FORMAL' (formal_argument | delimiter | termination_marker )* |

                'END' |

                'LET' (variable '=')* (number ( $ | ('+' | '-' | '*' | '/' ) number )

                                | ( '+' | '-' ) ( number | variable )

                                | identifier ( $ | '+' integer )

                                | delimiter

                                | unassigned variable ) |

                'DEFAULT' (variable '=')+ argument |

                'LABEL' label_identifier |

                'GOTO' (label_identifier | variable) |

                'IF' argument ('EQ' | 'NE' | 'GE' | 'LE' | 'GT' | 'LT' ) argument

                                'GOTO' (label_identifier | variable ) |

                'FOR' variable '=' (number | variable ) 'TO' ( number | variable )

                                ( $ | 'STEP' ( number| variable )) |

                'NEXT' variable |
'WRITE' ($ | variable | 'DIS | 'TP' | 'LP')($ | variable | 'FF' | 'LF')

            (variable | string )*

'READ' ( (variable ('INT' | 'REAL' | 'NUM' | 'NAME' | 'DELIM' |

            'YESNO') | termination_marker )+ |

'SUSPEND' |

'RESUME' |

'SWITCH' (variable | 'EXEC' | 'LOG' | 'TRACE')

            (variable | 'ON' | 'OFF') |

'FREE' (global_variable+ | '*.*') |

'STOP'
```

CTRL-C

Ctrl-C is an interactive program language for the analysis and design of multivariable systems. It was developed by Systems Control Technology, Palo Alto and the first version was released in January 1984. Ctrl-C is totally command driven. Ctrl-C is a derivative of Mathlab. Matrix-X and PC-Mathlab are similar in structure.

In Ctrl-C the only data structure is matrices.

Ctrl-C uses macros which are files of commands. The commands will be executed when typing DO FOO if the filename is FOO.CTR. There is also an INPUT command which facilitates the use of questions and answers and menus.

There is also a possibility for the user to define his own functions. User defined functions have the following structure

```
//[output1,output2,.....]=function_name(input1,input2,....)
.
. function body
.
```

Output1,output2 ... are the variables to be returned by the functions and input1, input2 ... are the input variables passed to the functions. Ctrl-C functions are call-by-value i.e. if the arguments are modified in the body of a User-Defined function, they do not affect the values of the variables in the calling routine. The user can easily create own "commands" by defining own functions.

EBNF description of Ctrl-C*

```
statement :== environ_state | flow_state | comment

flow_state :== assignment | for_state | if_state | while_state

assignment :== name '=' ( expr | "' string '" )

for_state :== 'FOR' name '=' expression ',' ( statement ( ',' | ';' ))*

                end_flow

if_state :== 'IF' expression conditionop expression

                ( statement ( ',' | ';' ))* end_flow

               ( $ | 'ELSE' ( statement | ( ',' | ';' ))* ) end_flow

while_state :== 'WHILE' expression conditionp expression
```

* Only the parts of Ctrl-C corresponding to the commands of Intrac are included in this description.

```
                  ( statement | ( ',' | ';' ))* end_flow

end_flow :== 'END' | 'CR'

environ_state :== 'BROWSE' |

                  'CLEAR' ( name* | '-' ( v | f | l | * )) |

                  'DO' filename |

                  'EDF' |

                  'EXIT' |

                  'HELP' |

                  'KEY' |

                  'LIB' 'name' ( '<' filename | $ )( '>' filename | $ )

                          ( $ | ( '-' ( d | e | l | b ))

                  'NEWS' |

                  'PAUSE' |

                  'QUIT' |

                  'WHO' |

                  'WHAT' |

                  '$' vms_command )

comment :== '//' ( commentline | '[' parameterlist ']' '=' name '(' parameterlist ')' )
```

condition_op :== < | > | <> | >< | =

expression :== numeric_expression | ']' string '[' |

name '(' parameterlist ')' | matrix | predefined

matrix :== '[' number^n ((';' | 'CR') number^n $)^*$ ']'

numeric_expression :== number (op rest | ':' number ($ | ':' number) |

name (op delim | '(' parameterlist ')' | ':' name ($ | ':' name))

op :== '+' | '' | '*' | '/' | '\' | './' | '.\' | '.*.' | '.\.' | ./. | '**'

rest :== (name | number)($ | op rest)

COMPARISON BETWEEN INTRAC AND CTRL-C

Intrac is influenced by the flow control structures of FORTRAN IV. The flow control statements in Ctrl-C resembles those of the C language. This shows that Ctrl-C is a younger language than Intrac. The structure of a complicated function will be easier to follow in Ctrl-C than in Intrac. since many jumps can be avoided.

The general command structure is "cleaner" in Intrac. This is partly because Intrac is smaller. The ending of flow control statements and the significance of CR in Ctrl-C is rather inconsequent. Some parts of Ctrl-C cannot be described using EBNF. The size of a matrix is defined implicitly as it is entered. All rows of the matrix must have the same length. This can not be described by a finite state machine, as is EBNF. A system with some kind of memory is required.

Both these interactives modules suffer from lack of generality. They both resemble another programming language but not as much as to enable someone used to the other language to easily switch to the module. This is the problem faced by every user of applied technical programs. For each new program there is a new input syntax to learn. If these programs used standard input modules as for example existing programming languages then life would be much easier for the designer engineer.

CHAPTER 3

LISP - History and Structure

HISTORY

LISP was one of the first symbolic programming languages. Today symbolic programming is very popular, but what was it that made John McCarthy start the LISP project in as early as in the fifties? In History of Programming Languages by Wexelblatt (1981) the history of LISP is told by McCarthy himself. Lisp's early history can be divided into two parts. The first one concerns John McCarthy's work with different artificial intelligence projects where he develops features of a programming language suitable for AI. The second part is the implementation of LISP at MIT.

Prehistory

LISP's prehistory runs from summer 1956 through summer 1958. In 1956 John McCarthy participated in the Darthmouth Summer Research Project on Artificial Intelligence, this was really the first organized study of AI. There his desire for an algebraic list-processing language for AI work arose. There already existed one list-processing language, which was described at the project, it was IPL 2 made for RAND Corporation's JOHNNIAC. But this language wasn't really what McCarthy wanted, he was attracted of the FORTRAN idea of writing programs algebraically.

The new language was to be developed for the IBM 704 and this was for two reasons. First, IBM was establishing a New England Computation Center at MIT, which Darthmouth would use. Second, IBM was undertaking to develop a program for proving theorems in plane geometry, where McCarthy was to be an consultant. It seemed likely that IBM would support future artificial intelligence projects.

McCarthy's own research in artificial intelligence led to his proposal of the Advice Taker in 1958. It involved representing information about the world by sentences in a suitable formal language and a reasoning program that would decide what to do by making logical inferences. Representing the sentences by list structures and using a list-processing language for programming the operations involved in deduction seemed appropriate.

This internal representation of symbolic information gave up the familiar infix notations in favour of a notation that simplifies the task of programming algebraic simplification, logical deduction and other substantive computations.

If infix notations are to be used externally, translation programs has to be written. Thus most LISP programs use a prefix notations for algebraic expressions, because they usually must determine the main connective before deciding what to do next. In this LISP differs from almost every other symbolic computation system.

In connection with IBM's plane geometry project it was decided to implement a list-processing language within FORTRAN, because this seemed to be the easiest way to get started and in those days it was believed that writing a compiler was to take many man-years. This work led to FLPL, standing for FORTRAN List Processing Language. While expressions could be easily handled in FLPL, and it was used successfully for the Geometry program, it had neither conditional expressions nor recursion, and erasing list structures was explicitly handled by the program.

The conditional expression was invented by McCarthy while working with a set chess legal move routines in FORTRAN. The IF statement in FORTRAN I and FORTRAN II was very awkward to use, and he then invented a function XIF(M,N1,N2) whose value was N1 or N2 according to whether the expression M was zero or not. The function was very useful but had to be used sparingly since all three arguments had to be evaluated before XIF was entered. This led to the invention of the true conditional expression which evaluates only one of N1 and N2 according to whether M is true or false and to McCarthy's desire for a language that would allow its use.

McCarthy spent the summer of 1958 at the IBM Information Research Department where he chose differentiating algebraic expressions as a sample problem. It led to the following innovations beyond FLPL:

Writing recursive function definitions by using conditional expressions.

The maplist function that forms a list of applications of a function to the elements of the lists.

When using functions as arguments, a notation for functions is needed, and McCarthy decided to use the λ-definition of Church (1941).

No solution for the erasure of abandoned list structures created by the recursive definitions was apparent att that time, but the idea of complicating the beautiful definition with explicit erasure was unattractive.

The differentiating program was never implemented that summer, because FLPL allows neither conditional expressions nor recursive use of functions. At that point a new programming language was necessary.

The Implementation of LISP

In the autumn of 1958 McCarthy became an Assistant Professor of Communications Sciences at MIT, and he and Marvin Minsky started the MIT Artificial Intelligence Project. Within this project the implementation of LISP began. The original idea was to produce a compiler, but this was considered a major undertaking, and some experimenting in order to get good conventions for subroutine linking stack handling and erasure was needed. Therefore various functions were hand-compiled into assembly language and subroutines were written to provide a "LISP environment". It is not clear whether the decision to use the parenthesized list notation was made then or whether it had already been in discussing the differentiation program.

The programs to be hand-compiled were written in an informal notation called M-expressions intended to resemble FORTRAN as much as possible. It also allowed conditional expressions and the basic functions of LISP. It was intended to compile from some approximation of the M-notation, but the M-notation was never fully defined, because representing LISP functions by LISP lists became the dominant programming language when the interpreter later became available.

The READ and PRINT programs induced a de facto standard external notation for symbolic information e.g. representing $x + 3y + z$ by (plus X(times 3 Y)Z) and $(\forall x)(P(x) \vee Q(x,y))$ by (ALL (X)(OR (P X)(Q X Y))). Any other notation necessarily requires special programming since standard mathematical notations treat different operators in syntactically different ways.

The erasure problem also had to be considered, and since McCarthy found it unaesthetic to use explicit erasure as did IPL, there were two alternatives left. The first alternative was to erase the old contents of a variable whenever it was updated. The second was to use a garbage collector, in which storage is abandoned until the free storage list is exhausted, the storage accessible from variables and stacks is marked, and the unmarked storage is made into a new free storage list. The second alternative was chosen since it didn't require reference counts, which turned out to be necessary in the first case. After deciding on garbage collection, its implementation could be postponed since only short programs were run.

Some simplifications were made to LISP during the implementation which gave a method to describe computable functions. McCarthy wanted to write a paper showing that LISP was neater than Turing machines for describing recursive functions. One way to do this was to write a universal LISP function EVAL[e, a] , which computes the value of a LISP expression $\underline{e}$, the second argument $\underline{a}$ being a list of assignments of values to variables (a is needed to make the recursion work). Writing EVAL required invention of a notation to represent LISP functions as LISP data. Such a notation was devised for the purposes of the paper with no thought that it would be used to express LISP programs in practice.

One of the students in the implementation group noticed that <u>eval</u> could serve as an interpreter for LISP, promptly hand coded it, and a programming language with an interpreter was obtained.

This unexpected appearance of an interpreter tended to freeze the form of the language, although some of the decisions made for the paper later proved unfortunate. Another reason for the initial acceptance of awkwardnesses of the internal form of LISP was that it was still expected that a switch would be made to writing programs as M-expressions. The project of defining M-expressions precisely and compiling them or at least translating them into S-expressions was neither finalized nor explicitly abandoned. It just receded inte the indefinite future.

<u>From LISP 1 to LISP 1.5 and beyond</u>

The LISP described above was LISP 1. This version was extended with several new features some of which fitted neatly in the LISP structure and others which did not fit so very well.

The two principal dialects of LISP developed after the LISP 1.5 are MACLISP and INTERLISP. MACLISP is the dialect used in Franz Lisp on VAX/VMS. The dialects have some principal differencies in their structures. However in both of them the basic functions are the same, although there are some discrepancies concerning the numbers of arguments of basic functions and how functions behave when applied to empty lists. The most important differences are described in Appendix 1 of LISP by Winston & Horn (1981). There are also greater differences in underlying structures, but these do not matter to the user.

STRUCTURE

<u>Lists and Atoms</u>

LISP has two basic types of data, atoms and lists. Atoms can look like these

```
johnny  apple  good
```

and a list may look like

```
(picking up good vibrations)
```

Lists may also include other lists as in this example

```
(life is (like a beanstalk) isnt it)
```

<u>Take Apart and Put Together</u>

Lists can be taken apart by using the functions CAR and CDR where CAR will return the first element in a list and CDR returns the list without the first element. In LISP the function name is written inside a parenthesis with the arguments following also within the same parenthesis. For example

```
(CAR '(we all came down to montreaux))
```

returns

```
we
```

```
(CDR '(we all came down to montreaux))
```

gives

```
(all came down to montreaux)
```

Incidentally these names are heritages from the IBM 704 used for the original LISP implementation. CAR originally stood for "Contents of the Address part of Register number" and in CDR "Data" was substituted for "Address". IBM 704 had special instructions that made the implementation of these two functions easy.

Since programs and data have the same structure, there must be a way of telling the LISP interpreter whether the argument should be evaluated as a function or not. For this purpose the function QUOTE is used. It indicates that its argument is not to be evaluated. To save space and writing effort 'apple can be substituted for (QUOTE apple).

To put together lists there are several functions. CONS will insert an element first in a list. APPEND makes one list out of two lists and LIST will make a list of the submitted elements.

```
(CONS 'alibaba '(and the forty robbers))
```

returns

```
(alibaba and the forty robbers)
```

and

```
(APPEND '(pandoras)'(box))
```

gives

```
(pandoras box)
```

Arithmetics

Arithmetic expressions in LISP are written the same way as other function calls i.e. the function name first and then the arguments. Addition can look like this

```
(PLUS 2.19 5.03)
 7.22
```

The other rules of arithmetic and some basic arithmetic functions like max, min, absolute value, square root and so on, are also available.

Atoms have Values

LISP's goal is always to evaluate an expression and return a value. If just typing in the name of an atom like this

```
Y
```

LISP tries to return a value for it, just as for any other expression. But in the case of an atom the value is something bound to the name of the atom and not the result of a computation. The value of an atom is established by the function set. The expression

```
(SET 'y '(a b c))
```

returns

```
(a b c)
```

but as a sideeffect it also makes (a b c) the value of y. The value can be either a list or an atom. Numbers are special atoms in that the value of a number is just the number itself.

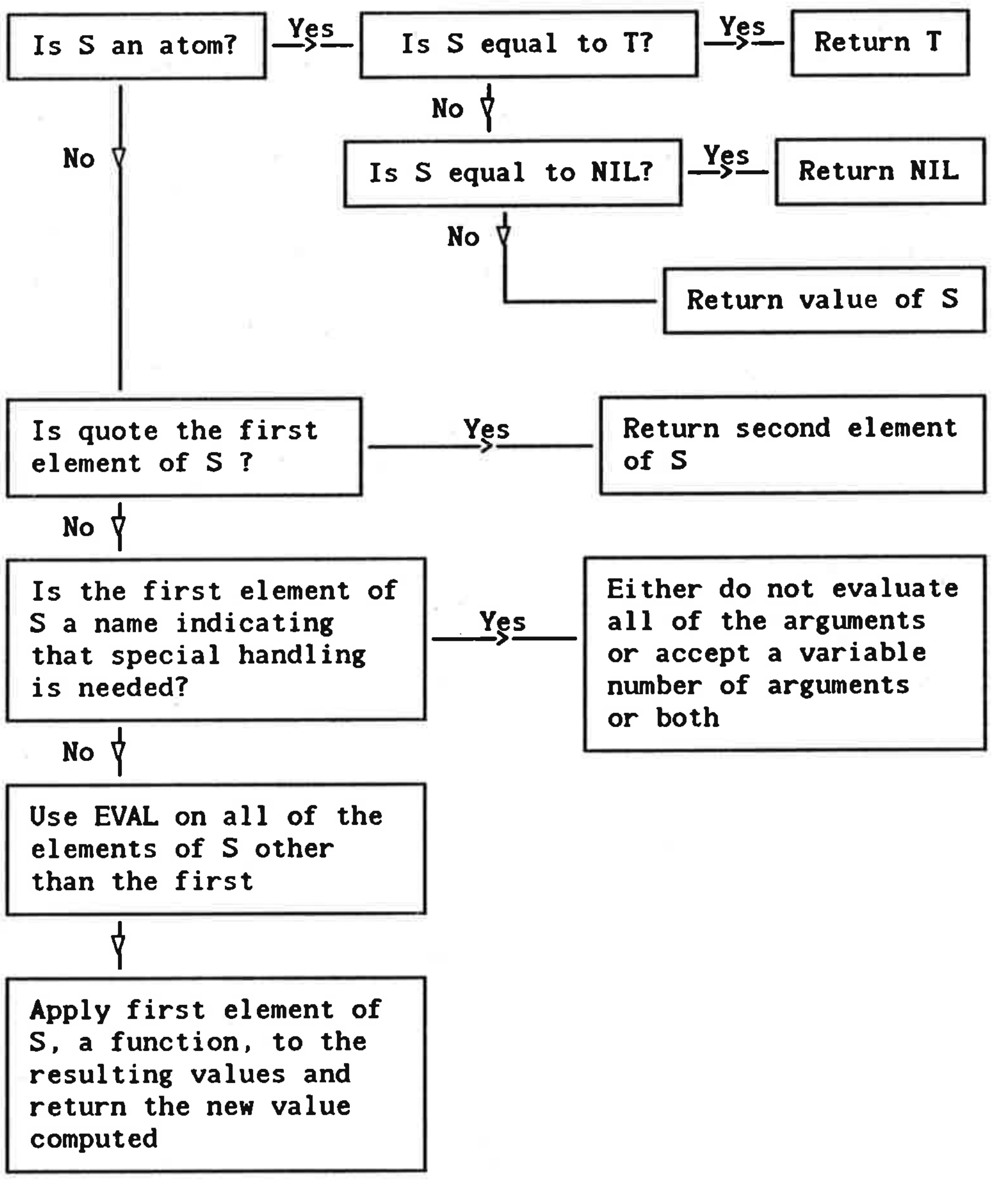
The function SETQ is even more used than SET. It treats the second argument as if it was already QUOTEd. The above example with SETQ looks like

```
(SETQ 'y (a b c))
```

but the result is of course the same.

EVAL

When an expression in typed in by a LISP user, it is automatically handed to the function eval. It evaluates the expression according to a certain scheme.


Is S an atom?
Yes
Is S equal to T?
Yes
Return T
No
Is S equal to NIL?
Yes
Return NIL
No
Return value of S
No
Is quote the first element of S ?
Yes
Return second element of S
No
Is the first element of S a name indicating that special handling is needed?
Yes
Either do not evaluate all of the arguments or accept a variable number of arguments or both
No
Use EVAL on all of the elements of S other than the first
Apply first element of S, a function, to the resulting values and return the new value computed

Definition of EVAL

EVAL can also be called explicitly causing an extra evaluation to be performed.

<u>New Functions</u>

New functions can be defined using the function DEFUN. The syntax of a function looks as follows

```
(DEFUN <function name>
        (<parameter 1> <parameter 2> ... <parameter n>)
        <process description>)
```

DEFUN establishes a function definition which can be referred to later on by having the function name appear as the first name of a list to be evaluated. The function name must be a symbolic atom.

Predicates

A predicate is a function that returns one of two special atoms, T or NIL. The atom T corresponds to logical true and NIL to logical false. The atoms T and NIL are special in that their values are preset: the value of T is T and the value of NIL is NIL. There are a lot of predicates defined in LISP most of them have a name ending in P e.g. LESSP, GREATERP, NUMBERP, ZEROP, MINUSP and LISTP. However some important ones does not end in P: ATOM, EQUAL and NULL. New predicates can be created by means of the AND, OR and NOT functions. AND and OR take any number of arguments. The arguments are evaluated from left to right, but only as many as needed to determine the value of the predicate.

The COND expression

The branching function in LISP is called COND. Together with predicates it can be used to determine which one of several expressions should be evaluated. The syntax is a bit peculiar:

```
(COND (<test 1> ... <result 1>)
   (<test 2> ... <result 2>)
   .
   .
   .
   (<test n> ... <result n>))
```

Each list following COND is called a clause. The first expression in each clause is evaluated until one is found that is nonNIL, i.e. if anything else than NIL is returned. Then the other expressions in the clause are evaluated and the last value computed is returned. Even though a clause may contain any number of expressions it usually only consists of the test expression and a result expression.

Recursion and "Normal" Programming

The traditional way to write a LISP function is to make it recursive i. e. it makes a function call to itself. The structure of LISP with every expression returning a value and no common statement by statement structure makes this the easiest approach. Algorithms can very often be defined recursively. Here is an example with a function computing the Fibonacci number for n. Fibonacci numbers are defined recursively as fib(n) = fib(n-1) + fib (n-2) and fib(1) and fib(2) equals 1.

```
(defun fibonacci (n)
        (cond ((equal n 0) 0)
              ((equal n 1) 1)
              (t (+ (fibonacci (- n 1))
                    (fibonacci (- n 2))))))
```

In the original LISP there were no possibility to make a statement program but this was later included. Actually it was one of the extensions made in LISP 1.5. The feature is called PROG and has this syntax

```
(PROG (<local variables>)
        (statement1)
        (statement2)
        .
        .
        .
        (statementn))
```

There are some functions that can only be used inside a PROG. They are

RETURN and GO. RETURN causes the PROG to be terminated returning the value of the of the argument to the RETURN that stopped the PROG. (GO <tag>) transfers control to the expression following the tag. A tag is a atom appearing as an argument to PROG. Tags will not be evaluated. PROG is not a very neat instruction and therefore in more modern LISP versions other language elements like FOR and WHILE have been included. These makes it easier to write structured code and the mess that frequent use of GO creates can be avoided. The fibonacci numbers can of course also be computed iteratively using a PROG. It could look like this:

```
(defun fibonacci (n)
        (prog (a b foo count)
             (setq a 1)
             (setq b 0)
             (setq count n)

             loop
             (cond ((equal count 0)(return b)))
             (setq foo a)
             (setq a (+ a b))
             (setq b foo)
             (setq count (- count 1))
             (go loop)))
```

What to Think about when Learning to Write LISP programs

It is easy to get confused when writing programs in LISP. At first the language itself and all the parentheses will confuse you. Its advisable to get a good LISP text-book and start by reading the first chapters and work through some of the examples. The first ones can be written down on paper only, but as the examples get more complicated its needed to get their behaviour confirmed by running them.

The next step is to get used to the run time system. It is important to learn the debugging as thoroughly as possible. Most LISP systems have facilities for tracing and stepping through the functions. Many systems also have the possibility to invoke an editor from inside the LISP. Since loading the whole LISP usually takes some time this will speed up work when testing new functions.

How to write in LISP is not obvious especially not if one is used to sequential programming languages like Fortran or Pascal. LISP is very suitable for writing recursive functions. It takes time to get used to this new approach to a given problem. When working with the exercises in the text-book it is important to really make an effort of solving the problem yourself. It is also important to compare the solution with the one of the text-book and to understand how the latter works.

As the problems get more complicated so do the programs and soon enough they get too big. In LISP it is necessary to break up a problem in smaller parts and code each of them as a small function. If each function is given an good mnemonic the main program will not look as incomprehensible as will an enormous pile of "cars", "cdrs" and "conses". When writing the "small parts" it is necessary that every part has a goal. Giving the different predicates, used for testing in cond statements, names by defining them as functions will also add readability to the programs.

A large LISP package has many function names. It is easier to keep track of the names if they are inserted in a special list automatically when they are defined. This can be done by using a special function instead of DEFUN in LISP which both defines and inserts the function name in a special list. Also the variables used inside the package may get mixed up with user variables if the same names are chosen. In newer LISP versions like Common LISP modules have been included. Writing a package of functions as a module gives the programmer the opportunity to decide which variables and functions are to be imported into and exported from the package.

When having written some programs in LISP it is important not to forget to go back and study the text-book to find out what useful features could have been used and which functions were forgotten.

Can LISP be Used as a Base for Command Language

To use LISP as a base for command language has several advantages over writing an entirely new input processor. Firstly a standard program is being used. This means it is being maintained by someone else. Further more a user familiar with LISP does not need to learn a new obscure command syntax to get started. A user not earlier acquainted to LISP will learn something that is generally used. Secondly LISP has much more power than a "home-made" input processor will get with reasonably much time and effort spent on it. Thirdly the symbolic manipulation functions in LISP which can be very useful are there for free.

There are of course also some disadvantages. If a modern LISP e.g. Common LISP is used problems with variable and function names getting mixed up can be eliminated. Of course there might be some difficulties when trying to interface LISP with the underlying programs written in another programming language.

To get some idea of what it is like to write a symbol manipulation package in LISP the subject of polynomial algebra is chosen. Writing a package of his own will be possible for the user of a future system equipped with LISP. The question is this: Is it enough to give a relatively unexperienced user the tools of LISP or does he need more powerful tools to get any result?

CHAPTER 4

Formal and Numerical Representation of Polynomials

WHAT IS A POLYNOMIAL?

Before deciding on how to represent polynomials it is essential to define what a polynomial is. Polynomials are normally defined relative to certain variables. For simplicity we will restrict ourselves to polynomials having just one indeterminate (so-called univariate polynomials). Usually we define a polynomial to be a sum of terms each of which is either a coefficient, a power of the indeterminate or a product of a coefficient and a power of the indeterminate. A coefficient is defined as an algebraic expression which is not dependent on the indeterminate of the polynomial.

Another question when dealing with polynomials is this: Is the polynomial

$$3x^3 + 7x + 2$$

the same as the polynomial

$$3y^3 + 7y + 2$$

or not? The answer is yes if just the purely mathematical function is considered but no if a polynomial is regarded as a syntactic form. In the following discussion of polynomial representation only the mathematical function will be considered. However the syntactical representation can be divided into one part representing the mathematical function of the polynomial and another part containing the syntactic information. Thus the restriction to the mathematical function of the polynomial is not very limiting.

The coefficients of the polynomials are restricted to real numbers and expressions, since the polynomials to be described are polynomials in control theory .

REPRESENTATION OF POLYNOMIALS

There are three basically different approaches to the representation of univariate polynomials, that is they can either be represented by their coefficients:

$$c_n x^n + c_{n-1} x^{n-1} + \dots + c_0$$

or by their roots in a factorized representation:

$$k(x + r_1)(x + r_2) \dots (x + r_n)$$

The third way is to represent a polynomial of the degree n by its values in n+1 points.

Converting from factorized form to coefficients form is done by multiplying the roots together. This can easily be done formally. Converting the other way around, from coefficients to factorized form is difficult since it means solving a n:th degree equation formally. That is only possible to do up to the fourth degree in the general case.

REPRESENTATIONS BY COEFFICIENTS

When representing a polynomial by its coefficients one straightforward approach is to just make a list of the coefficients

$$(c_n\ c_{n-1}\ \dots\ c_0)$$

The degree n is easily computed by the LENGTH function in LISP. In languages which do not have the possibility of using lists arrays could be used. Either the order could be reversed so that the coefficient of zero degree comes first

$(c_0\ c_1\ \dots\ c_n\ )$

or the number of the highest degree n could be inserted first in the array.

$(n\ c_n\ c_{n-1}\ \dots\ c_0)$

The latter has the advantage of the degree n being much easier to compute than in the former representation.

This way of representation is good for dense polynomials, but for sparse polynomials like this one

$x^{100} + 3x^3 + 2$

the list will become unpractically long. In this case one can either put exponents in one sublist and coefficients in another one like this

((100 3 0)(1 3 2))

or combine them as pairs, one for each term.

((100 1)(3 3)(0 2))

Representation by coefficients is suitable for formal polynomials since there is no need to transform to any other representation form when performing addition or multiplication.

REPRESENTATION BY ROOTS OR FACTORS

An unvariate polynomial can also be represented by its roots. In case of only real roots there is no problem. The polynomial will be represented by a list beginning with k, the coefficient off the highest power term, and then all the roots like this:

$(k\ r_0\ r_1\ \dots\ r_n)$

If there are also complex roots different ways of representing these may be chosen. Each root can be represented as a complex number e.g. like this

$(k\ (r_0\ i_0)(r_1\ i_1)\dots\ (r_n\ i_n)\ )$

where r_k denotes the real part and i_k the imaginary part of the complex number. If complex numbers are unwanted the complex poles could be combined to second order expressions. Since complex poles are complex-conjugated, when the coefficients are real, this is possible to do.

$(k\ (c_{00}\ c_{01})(c_{10}\ c_{11})\dots\dots\dots\ r_n)$

The complex pol pairs are represented by the two real coefficients of the second order expression c_{k0} and c_{k1} and real poles by r_k as in the first example.

Representation by factors is not very suitable when working with formal polynomials since generally only polynomials up to the fourth degree can be factorized formally. Multiplication is easily performed by just appending the lists. Addition and subtraction on the other hand may be impossible to perform since transformation to representation by coefficients and following factorization is needed.

REPRESENTATION BY POINTS

A n:th degree polynomial can also be defined by its values in n+1 points. To transform back to a more familiar representation the Lagrange Interpolation Formula can be used. Addition and subtraction of two polynomials of the same degree is very easy to perform by just adding or subtracting their values at corresponding points. When multiplying two polynomials of nth degree 2n+1 points are needed to make transformation back possible. Of course a fixed number of points could be calculated for each polynomial to be represented and thus a limit is set for how high degrees it is possible to calculate with. This way of representation may be suitable in languages which do not have lists since arrays of fixed length could be used.

CHAPTER 5

POLIS

A Polynomial Manipulation Package

INTRODUCTION

POLIS stands for polynomials in Lisp, it is a small package for manipulation of polynomials. It is run in Franz Lisp, a Lisp implemented at University of California, Berkeley. Franz Lisp can be run under Unix or other Unix resembling operating systems.

To run Franz Lisp on a Vax system under Eunice (which is a program that simulates Unix on a Vax running under the VMS operating system) some commands ought to be included in the login file. These however differ from one VMS/Eunice system to another wherefore it is advisable to ask the system manager for information.

With this piece of information added, the command "lisp" will start Franz Lisp and return some text and the prompt ->. It's important to notice that Franz Lisp <u>does</u> make a difference between upper and lower case letters. Subsequently since all commands described in this manual is defined in lower case letters they must also be typed in that way to be understood by the Lisp.

When having entered Franz Lisp the POLIS package, if it is available on your computer, will be loaded by the following command

```
(load 'polis)
```

Observe the single quote mark which denotes that "polis" is not to be evaluated.

REPRESENTATION AND DEFINITION OF POLYNOMIALS

In POLIS a polynomial is represented by its coefficients in a Lisp-list.* The zero-order coefficient is placed first followed by the others. For example the polynomial

$3x^3 + 2x^2 + 5$

is represented by the list

(5 0 2 3).

A polynomial can be entered by just specifying the coefficients. The above polynomial can be defined by this LISP expression

```
(setq pol '(5 0 2 3))
```

Standard Lisp functions, like 'setq' above, are used for different purposes as defining polynomials.

The coefficients of the polynomial does not have to be numbers. They can of course also be names or expressions like in this case:

$s^2 + 2\theta\varsigma s + \theta^2$

This polynomial can be entered like this

```
(setq pol2 '(1 2*w*z w^2))
```

* The acquaintance of Lisp necessary for running and understanding POLIS is provided in chapter 3.

If the polynomial is known defined by its poles it's not necessary to calculate its coefficients by hand. The function poles-to-coeffs can be used for conversion to the normal representation used in POLIS. If the poles are real then the conversion is straightforward. For example

$(x + 3)(x - 5)(x + 4)$

gives the representation

```
(3 -5 4)
```

It can be defined the same way as the polynomial above

```
(setq polyn '(3 -5 4))
```

In order to be able to use this polynomial in POLIS it must be transformed into the normal POLIS representation form. The list of values defined above is transformed by the command

```
(poles-to-coeffs polyn)
```

which returns

```
(-60 -23 21 2 1).
```

This is the representation of the polynomial

$x^4 + 2x^3 + 21x^2 - 23x - 60.$

If there are also complex poles as in this example

$(s + 2)(s + 1 + i)(s + 1 - i)(s - 3)$

then it's necessary to combine the complex pole-pair to a second-order expression with real coefficients. The example above thus yields

$(s + 2)(s^2 + 2s + 2)(s - 3)$

```
(setq complex-polyn '(2 (2 2) -3))
```

defines the polynomial above. Apparently polynomials with complex roots can only be introduced in POLIS if the complex poles are complex-conjugated. This however does not put any restrictions on the use of POLIS in control design where complex poles always are conjugated, since coefficients are real.

The complex-polyn polynomial can now easily be transformed by the poles-to-coeffs command.

```
(poles-to-coeffs complex-polyn)
```

returns (-1 -14 -6 1 1).

This is of course the representation of the polynomial

$x^4 + x^3 - 6x^2 - 14x - 12.$

ARITHMETICS

All arithmetic functions in POLIS take two Lisp-lists, representing polynomials as described in the former part of the manual, as arguments. The polynomials can be defined in advance with setq or explicitly defined when needed.

Addpol

Addpol performs an addition between two polynomials by adding each coefficient. The form is (addpol pol1 pol2) which means pol1 + pol2.

Example:

```
(addpol '(a 2 4 e)'(12 0 2 b 1))
```

returns

```
((12 + a) 2 6 (e + b) 1).
```

Subpol

Subpol performs a subtraction between the two polynomials. The syntax of the instruction is (subpol pol1 pol2) which means pol1 - pol2.

Example:

```
(subpol '(a 2 4 e) '(12 0 2 b 1))
```

returns

```
((- 12 + a) 2 2 (e - b) -1).
```

Multpol

Multpol performs multiplication between two polynomials. The syntax of the instruction is (multpol pol1 pol2) which means pol1 * pol2.

Example:

```
(multpol '(a 2 4 e)'(12 0 2 b 1))
```

returns

```
((12 * a) 24 (48 + (2 * a))(12 * e + 4 + (a * b))(8 + (2 * b) + a) (2 + (2 * e + (4
* b)))(4 + (b * e)).
```

Divpol

Divpol performs a div between two polynomials. The div function is defined such as pol1 divided by pol2 equals the highest degree non-zero term of pol2 divided by the non-zero term of highest degree of pol1.

$4x^2 + 4x + 2$ div $2x^2 + x - 6$ thus equals 2

whereas $4x^2 + 4x + 2$ div 2 equals $2x^2$.

Examples:

```
(divpol '(a b c)'(2 c f))
```

returns

```
(c / f)
```

```
(divpol '(a b c)'(2))
```

returns

```
(0 0 (c / 2))
```

<u>Modpol</u>

modpol performs an modulo function between pol1 and pol2. The syntax of the instruction is (modpol pol1 pol2). This means pol1 - (pol1 div pol2) * pol2 where div is defined as in divpol.

Example:

```
(modpol '(a b c)'(d e f))
```

returns

```
((a - (c / f * d)(b - (c / f * e)(c - (c / f * f)).
```

EVALUATING POLYNOMIALS

It is also possible to evaluate polynomials in POLIS. I.e. insert some value for x in an polynomial like

$4x^2 + 2x + 4$.

If x is given the value 2 this would give the result 24.

<u>Evalpol</u>

Evalpol will perform the evaluating of a polynomial. It is required that each variable that occurs in the coefficients is assigned a value. This can be done using lisp function setq.

Example:

```
(setq poly '(a b c))
```

returns

```
(a b c)
```

To evaluate the polynomial poly first assign values to a, b and c. This is done by the following commands.

```
(setq a 3)
3
(setq b 2)
2
(setq c 1)
1
```

After all variables have got their values the evaluation can be performed.

```
(evalpol 3 poly)
```

returns

```
19
```

If the value of one of the coefficients is changed for example c by

```
(setq c 3)
```

then (evalpol 3 poly) returns 36.

To get the current value of a variable just enter its name and the value will be returned.

It is also possible to evaluate a polynomial with a complex base. The complex number is then represented by a list with two atoms in it. 3 + i thus is represented by (3 1). The result will be returned the similar way.

Example:

The polynomial poly from above is used.

```
(evalpol '(3 1) poly)
```

returns

```
(33 22).
```

OBTAINING INFORMATION

In POLIS it is possible to obtain information about the system while working with it.

Info

The help command will provide information about available POLIS commands, their syntax and other subjects closely related to POLIS.

Example:

```
(info multpol)
```

returns information about the function multpol.

```
(info ?)
```

returns all the topics available in the help function.

```
(info all)
```

returns the help text for all topics available in help.

ERRORS AND TERMINATION

If an error occurs during the session, for example when trying to evaluate a polynomial with an undefined variable, an error message will occur on the screen and Lisp will enter another level.

Example:

```
(setq wrong)
Error: odd number of parameters to function setq
<1>:
```

To return to the top level just type in a Ctrl-Z.

```
<1>:Ctrl-Z
[Return to top level]
->
```

Thereafter the Lisp is ready to continue. All defined parameters are still defined.

The POLIS session is terminated by entering Ctrl-Z when in top level (i.e. when the prompt looks like ->).

Example:

```
->Ctrl-Z
Goodbye
```

```
$
```

When leaving Franz Lisp all defined variables are cleared and has to be redefined when entering Lisp the next time.

POLIS SESSION EXAMPLE

An entire POLIS session will be presented exactly as it appears on the screen. The command entered by the user are the ones appearing after the prompts ($ in VMS and -> in Franz Lisp).

```
$lisp
Franz Lisp, Opus 38.79
->(load 'polis)
[load polis.l]
t
->(setq pol1 '((w * w)(2 * z * w) 1))
((w * w)(2 * z * w) 1)
->(setq pol2 '((a * w) 1))
((a * w) 1)
->(setq denomg (multpol pol1 pol2))
((w * w * (w * a))(w * w + (2 * z * w * (w * a)))(2 *
z * w + (w * a)) 1)
->(setq z 0.7)
0.7
->(setq w 0.5)
0.5
->(evalpol 2 denomg)
Error: Unbound variable a
<1>:(return 1)
14.125
->Ctrl-Z
Goodbye

$
```

Comments

Franz Lisp is started and the POLIS package is loaded. Pol1 is set to

$s^2 + 2zws + w^2$

and pol2 is set to

$s + aw$.

These two polynomials are multiplied by the function multpol and the result is assigned to the atom denomg. To the atom z 0.7 is assigned and to w 0.5. The polynomial denomg is then evaluated but since no value is assigned to "a" an error occurs. the atom "a" is given the value 1 in the return statement and the result of the evaluation is displayed. The exit from Franz Lisp a Ctrl-Z is typed in.

Writing Formal Polynomial Algebra in LISP

The problem is principally rather simple and writing the short functions performing the arithemtic operations on polynomials is easily done. The results returned will however look rather messy and will be hard to read. One would want something that cleans up and simplifies the results. This is however not so easily done. The major part of the programming time spent on a project like this is spent on postprocessors for the results. These tend to get very complicated.

A symbolic simplification package would make the problems smaller. Functions for simplification, factorization and other similar functions would be available and these could be included in functions written by the user.

It is suitable for a future input processor with LISP as a base to be equipped with a symbol manipulation and simplification package. Such a package will make it easier for the user to make own symbol manipulation commands. If the LISP is not enhanced in this way there is a obvious risk for the not so experienced user to get stuck.

To investigate the possibilities of a symbol manipulation package some functions were written in MACSYMA.

CHAPTER 6

MACSYMA

A Symbolics Manipulation Program

INTRODUCTION

MACSYMA is a symbolics manipulation program written in Franz Lisp. It was developed at MIT laboratory for Computer Science and is now supported by Symbolics Inc.*

MACSYMA is an interactive program. It is command driven and can perform symbolic manipulations like formal integration, solving differential equations formally etcetera. As control design often means a lot of formal calculations it can be very useful e.g. for calculating the transfer function from an equation system of node equations. Pattern matching and plotting facilities are also featured in MACSYMA.

There are several hundreds of commands (functions) and over one hundred flags to control the behaviour of MACSYMA during execution. Therefore MACSYMA may seem a bit vast to the beginner.

GENERAL STRUCTURE

Every command (the lines written by the user) in MACSYMA has a label which is

* MACSYMA is a large symbolic manipulation program developed at the MIT Laboratory for Computer Science and supported from 1975 to 1983 by the National Aeronautics and Space Administration under grant NSG 1323, by the Office od Naval Research under grant N0014-77-C-0641, by the US Department of Energy under grant ET-78-C-02-4687, and by the US Air Force under grant F49620-79-C-020, and since 1982 by Symbolics, Inc. of Cambridge, Mass. MACSYMA is a trademark of Symbolics, Inc.

written to the left on the screen e.g.

```
(c13) a : 3;
```

where (c13) is the C label of the command where 3 is assigned to the atomic variable a. Every command must be concluded with a semicolon for the value to be returned. If concluded with an dollar sign $ the result will not be displayed on the screen. The above command will return

```
(d13)                              3
```

The D labels are used for output expression and there are also E labels for intermediate output expressions. Earlier output expressions can be referred to by their label in following commands.

In MACSYMA the user can define his own functions. The approach is very LISP-like, naturally enough, with every expression returning a value. The syntax of the functions is :

```
foo (bar) := <expression> $
```

The angle brackets are used to delineate descriptions of things. A user defined function will look exactly as the default functions of MACSYMA when used and will become a part of the "language" just as in LISP.

An atomic variable may have different properties. A property is a piece of information which may be used during the user's session with MACSYMA. In the above example the atomic variable "foo" will get the property "function". There are other properties like value, array, macro and so on. Most of the properties are set automatically while working with MACSYMA but there are special commands to display properties and to remove them.

There is a conditional expression working like the "cond"-expression in LISP i.e. by returning a value, but with a syntax similar to Algol with IF ... THEN ... ELSE.

When wanting to perform traditional statement by statement programming a BLOCK can be used. A BLOCK is a set of statements separated by commas. The first statement is a list of the local variables. Statements to use in BLOCKS are :

> GO statements which causes control to transfer to the statement which is labelled by the argument of the GO.
>
> RETURN statements which causes the block to exit with the value of the argument of the RETURN. A BLOCK whose execution is not terminated by a RETURN statement will return the value of the last statement.
>
> FOR statements which are used to repeat a statement (or a set of statements) a certain number of times or until a certain condition is fulfilled.

Here is two examples of a functions which computes fibonacci numbers, the first one recursively and the second one iteratively using a block.

```
fibonacci-rec (n) := if n = 0 then 0 else
            if n = 1 then 1
            else fibonacci-rec(n-1) + fibonacci-rec(n-2)$

fibonacci-iter (n) :=
              block([fo,fn,foo],
              fo:1,
              fn:1,
              loop,
              if n = 0 then return (fn),
              foo:fn,
              fn:fn+fo,
              fo:foo,
              n:n-1,
              go(loop) )$
```

SOME IMPORTANT FUNCTIONS

The EV function is one of the most powerful functions in MACSYMA. EV (exp, arg1,...,argn) evaluates the expression exp in the environment specified by the arguments. This is done in steps, as follows:

1. First the environment is set up by scanning the arguments. The arguments causes different actions to be taken e.g. extra post-evaluation, expansion, evaluation of predicates. The arguments can be given in any order but since they are picked up from left to right the order may effect the result.
2. The variables, including subscripted variables, in exp are replaced by their global values. Variables whose substitution is indicated by arguments are not replaced.
3. If any substitutions are indicated by the arguments they are carried out in this step.
4. The resulting expression is re-evaluated (unless one of the arguments is NOEVAL) and simplified according to the arguments.
5. If one of the arguments are EVAL steps (3) and (4) will be repeated.

SOLVE (exp, var) solves the algebraic equation "exp" for the variable "var" and returns a list of solution equations in "var". There is also a version SOLVE ([eq1, ..., eqn], [v1, ..., vn]) which solves a system of simultaneous (linear or non-linear) polynomial equations.

WHAT TO THINK OF WHEN WRITING IN MACSYMA

Writing in MACSYMA very much resembles writing in LISP, only that in MACSYMA there are many more functions. LISP techniques with recursion and lists can be used in MACSYMA.

The first main problem with MACSYMA is to get a view of the functions that are already there. It is not quickly done to read through the manual. The easiest way to get started is to talk to someone who has already got some MACSYMA experience and can guide and give hints on what functions to use. However if no such person is available there is a short introduction available from Symbolics Inc. In MACSYMA there is also a function called PRIMER(). It provides an on-line primer for the novice including an introduction to MACSYMA syntax, assignment and function definition, and the simplification commands. If really wanting to explore the possibilities of MACSYMA there is no other way than reading the manual. Of course some chapters are more important where other special functions are just interesting to specialists.

To load MACSYMA into the system takes at least a minute, therefore it is not desired to exit and enter very often. There is an built-in command editor but when writing longer functions it is better to use a standard editor. Since it is possible to exit to LISP inside MACSYMA, a LISP function that invokes the editor can be used. Other operating system functions can also be used from inside MACSYMA this way. Especially getting rid of old program versions can be useful.

Since there is no text-book or tutorial the only way of overcoming the obstacles of MACSYMA is by trial and error and by reading the manual. It is likely that a few days of reading the MACSYMA manual save many days of programming effort.

CHAPTER 7

MACPOL

Polynomial Synthesis using MACSYMA

INTRODUCTION

Polynomial synthesis is one important way of performing synthesis in automatic control. It is fairly simple once decisions have been made concerning the wanted system and the observer, but the calculations get rather extensive with a lot of unknowns and that's where Macsyma can aid the designer.

USER'S MANUAL

This package is used when wanting to calculate the controller of a system that is described by its transfer function. The observer function has to be determined manually and also the desired denominator of the controlled system has to be supplied.

The transfer function of the system will be regarded as

$$H(s) = \frac{B(s)}{A(s)}$$

if its a continuos time system and as

$$H(z) = \frac{B(z)}{A(z)}$$

if is a discrete time system. The observer polynomial will be called $A_0(s)$ and

$A_0(z)$ respectively and the desired system denominator as Am(s) or $A_m(z)$ respectively.

If the B-polynomial of the given system is known not to have any zeros, that could cause instability if cancelled, then the easiest way is to use the polynomial synthesis package is to apply the polsynth-function to the polynomials A, B, A_0 and A_m. There are to different versions ,one for continuos time systems called polsynthc and one for discrete time called polsynthd.

The closed-loop characteristic equation of the total system will be

$$AR + BS = A_0A_m$$

The function will return the polynomials R(s) and S(s) or R(z) and S(z) and these will form the transfer function of the controller

$$H_{cont} = \frac{S}{R}$$

If however the B-polynomial does have zeros, which will cause instability when cancelled, then the B-polynomial will have to be divided such as:

$$B = B^+B^-$$

where B^- has all its zeros in the stability zone and B^+ is monic. In this case the function polsynthb will take the arguments (A, B^-, B^+, A_0, A_m). Observe that the closed-loop characteristic equation in this case becomes

$$AR + BS = B^+A_0A_m$$

Function Syntax

Both functions look like this in continuos time version :

polsynthc (A(s), B(s) , A_0(s) , A_m(s))

polsynthbc (A(s), B^-(s), B^+(s), A_0(s), A_m(s))

The discrete time versions of course look very much the same:

polsynthd ($A(z)$, $B(z)$, $A_0(z)$, $A_m(z)$)

polsynthbd ($A(z)$, $B^-(z)$, $B^+(z)$, $A_0(z)$, $A_m(z)$)

The polynomials submitted can be in factor or coefficient form or any combination of the two. Naturally they can also be expressed as a Macsyma variable i.e.

```
poly : s^2 + x;
```

or as a Macsyma function

```
poly(z) := z^2 + x*z + y;
```

Error Messages

Some conditions have to be fulfilled by the submitted polynomials, otherwise error messages will be returned. In case there is a polynomial equal to zero, the package will send an error message e.g.

```
A0-polynomial is zero
```

if the observer polynomial is equal to zero. Checks will also be made on the degrees of the polynomials. In order to describe a real system the degree of the A-polynomial must be greater or equal to the degree of the B-polynomial. If that is not the case this error message occurs on the screen:

```
degree of Apol less than degree of Bpol
```

Also the right-hand side of the closed-loop characteristic function must have a degree that is greater or equal to the one of the A-polynomial. In case of the polsynthc and polsynthd functions this message will be returned:

```
degree of A0*Am less than degree of Apol
```

and similarly in the divided B version in the functions polsynthbc and polsynthbd:

```
degree of Bplus*A0*Am less than degree of Apol.
```

IMPLEMENTATION MANUAL

The package consists of four input functions and one main function with five helpfunctions. The input functions are polsynthc, polsynthd, polsynthcb and polynthdb. AXBYC is the main function and the subfunctions are CreateX, CreateY, Equsys and Removelist.

Input Functions

The input functions are used as interface with the user. Polsynthc and polsynthd are used for systems with totally stable inverses of the B-polynomial and take the arguments A, B, A_0, A_m in this order. The c stands for continuos time and polynomials must be in s whereas d stands for discrete time and there polynomials must be in z. Polsynthbc and polsynthbd are used when B has to be divided in two parts and thus take five arguments A, B^-, B^+, A_0 and A_m. The letters c and d at the end of the function names defines the same as above. All four functions first expand the passed polynomials whereafter checks are made to ensure validity of the indata. If no error is detected any of the input functions will result in one single call of the function AXBYC.

Main Function

AXBYC first expands all the polynomials to get them on term form. Then it checks that input data is valid, i.e. polynomial is nonequal to zero and degree(Apol) $\geq$ degree(Bpol) and degree(Cpol) $\geq$ degree(Apol). The degrees of the x-polynomial and the y-polynomial are computed and those polynomials are created by the subfunctions CreateX and CreateY. The righthand side of the equation AX+BY=C is computed and expanded (local variable inter) and all coefficients from the X-polynomial and the Y-polynomial are put in the list xandy. Subfunction equsys makes an equation system from inter and cpol. This equation system is solved by MACSYMA function solve and the its solution is assigned to the local variable solution. To obtain the answer the X and Y-polynomials are evaluated with the values from "solution" assigned to the variables. The evaluated x and y-polynomials are then returned.

Subfunctions

createX (base, deg) creates a polynomial in the base "base" with the degree "deg". The polynomial is monic that is the coefficient of the highest degree is one. createx (z,3) returns z^3 + x2 z^2 + x1 z + x0.

createY (base, deg) creates a polynomial in the base "base" with the degree "deg". createy (s,2) returns y2 s^2 + y1 s + y0.

equsys (exp1, exp2, base) creates an equation system from the two polynomials "exp1" and "exp2 " of base "base". "Exp1" must be of the highest degree and one equation is created for each exponent of "base" from the highest degree to zero.

removelist (var, list) removes "var" from "list" if "var" is a member of the "list" (though not inside expressions). If "var" is a member more than once all occurences are removed. removelist (a, [a, f, d, a, g]) returns [f, d, g].

AN EXAMPLE

As an example for the polynomial synthesis function the following problem was chosen. A motor is connected to a heavy wheel which is connected via a spring to another heavy wheel.

The transfer function from the motor current to the position of the second wheel will then be:

$$\frac{B}{A} = \frac{k}{s\ (\ s^2 + 2\zeta\omega s + \omega^2)}$$

The desired denominator will look like this:

$$A_m = (\ s^2 + 2\zeta_m\omega_m\ s + \omega_m^2\ )(\ s + a\ )$$

The observer will have the following form:

$$A_0 = (\ s^2 + 2\zeta_0\omega_0 s + \omega_0^2\ )$$

Since B doesn't have any zeros in the right halfplane the equation which has to be solved in order to determine the regulator S/R looks like this:

$$AR + BS = A_0A_m$$

The degree of the R and S polynomials must be chosen properly.

$$R = s^2 + r_1 s + r_0$$

$$S = s_2 s^2 + s_1 s + s_0$$

The obtained equation system when all exponents of s have been removed:

$$1 = 1$$

$$2\omega\zeta + r_1 = 2\omega_m\zeta_m + 2\omega_0\zeta_0 + a$$

$$2\omega\zeta r_1 + r_0 + \omega^2 = 4\omega_0\zeta_0\omega_m\zeta_m + 2a\omega_m\zeta_m + 2a\omega_0\zeta_0 + \omega_m^2 + \omega_m^2$$

$$2\omega\zeta r_0 + ks_2 + \omega^2 r_1 = 4a\omega_0\omega_m\zeta_0\zeta_m + 2\omega_0^2\omega_m\zeta_m + 2\omega_0\omega_m^2\zeta_0 + a\omega_m^2 + a\omega_0^2$$

$$ks_1 + \omega^2 r_0 = sa\omega_0^2\omega_m\zeta_m + 2a\omega_0\omega_m^2\zeta_0 + \omega_0^2\omega_m^2$$

$$ks_0 = a\omega_m^2\omega_0^2$$

The above can with some effort and carefulness be solved by hand and will give the following solution:

$$r_0 = 4\omega_0\zeta_0\omega_m\zeta_m - 4\omega\zeta\omega_0\zeta_0 + 2a\omega_m\zeta_m + 2a\omega_0\zeta_0 - 2\omega\zeta\omega_0\zeta_0 + 4\omega^2\zeta^2 - 2a\omega\zeta - \omega^2 + \omega_m^2 + \omega_0^2$$

$$r_1 = 2\omega_m\zeta_m + 2\omega_0\omega_0 - 2\omega\zeta + a$$

$$s_0 = \frac{a\omega_0^2\omega_m^2}{k}$$

$$s_1 = \Big[-4\omega^2\omega_0\omega_m\zeta_0\zeta_m + 4\omega^3\omega_m\zeta\ \zeta_m + 2a\omega_0^2\omega_m\zeta_m - 2a\omega^2\omega_m\zeta_m + 4\omega^3\omega_0\zeta\zeta_0 - 2a\omega^2\omega_0\zeta_0 + 2a\omega_0\omega_m^2\zeta_m + \omega_0^2\omega_m^2 - \omega^2\omega_m^2 + \omega^4 - 4\omega^4\zeta^2 + 2a\omega^3\zeta - \omega^2\omega_0^2 \Big] \ / \ k$$

$$s_2 = \Big[4a\omega_0\omega_m\zeta_0\zeta_m - 8\omega\omega_0\omega_m\zeta\zeta_0\zeta_m + 8\omega^2\omega_m\zeta^2\zeta_m - 2\omega^2\omega_m\zeta_m - 4a\omega\omega_m\zeta\zeta_m + 2\omega_0^2\omega_m\zeta_m + 8\omega^2\omega_0\zeta^2\zeta_0 - 2\omega^2\omega_0\zeta_0 - 2a\omega^3\zeta -$$

$$4a\omega\omega_0\zeta\zeta_0 + 2\omega_0\omega_m^2\zeta_0 - 2a\omega^2\omega_0\zeta_0 + 2a\omega_0\omega_m^2\zeta_0 + \omega_0^2\omega_m^2 -$$

$$\omega^2\omega_m^2 + \omega^4 - 4\omega^4\zeta^2 + \omega^2\omega_0^2 \,]\ /\ k$$

These calculations are of course not mathematically difficult but to ensure the correctness of the results, checks and rechecks, which will take valuable time for the control designer, are needed. Also when numerical values of these functions are wanted a lot of calculations has to be performed. The engineer is nowadays always aided by a calculator or even a computer, but even so the complex functions has to be entered into the machine, without any errors being added. When using the Polynomial Synthesis package, all that has to be done is entering the polynomials in Macsyma and use the function polsynthc. The session for the above example could look like this:

```
$macsyma
This is REX MACSYMA Release 305
(c) 1976,1979 Massachusetts Institute of Technology
All Rights reserved.
Enhancements (c) 1983, Symbolics, Inc. All Rights Reserved
Type describe(trade_secrets); to see Trade Secret notice.


(c1) batch(macpol);

                                          .
                                          .
                             The macpol code appears here
                                          .
                                          .

(d11)   BATCH DONE

(c12) apoly: s*(s^2+2*w*z*s+w^2);

                                               2     2
(d12)                            s (2 s w z + w   + s )

(c13) bpoly: k;

(d13)                                      k

(c14) a0poly: s^2+2*w0*z0*s+w0^2;

                                                2     2
(d14)                            2 s w0 z0 + w0   + s
```

```
(c15) ampoly: (s^2+2*zm*wm*s+wm^2)*(s+a);

                                                     2    2
(d15)                        (s + a) (2 s wm zm + wm  + s )

(c16) polsynthc(apoly,bpoly,a0poly,ampoly);

Dependent equations eliminated:  (6)

(d16) [wm (4 w0 z0 zm - 4 w z zm + 2 a zm) + s (2 wm zm + 2 w0 z0 - 2 w z + a)

                                2    2                      2     2    2
 + w0 (2 a z0 - 4 w z z0) + w  (4 z  - 1) - 2 a w z + wm  + w0  + s ,

 2                                          2     2
s  (wm (w0 (4 a z0 zm - 8 w z z0 zm) + w  (8 z  - 2) zm - 4 a w z zm

      2          2      2                              2
 + 2 w0  zm) + w0 (w  (8 z  - 2) z0 - 4 a w z z0) + wm  (2 w0 z0 - 2 w z + a)

    3           3       2     2          2
 + w  (4 z - 8 z ) + a w  (4 z  - 1) + w0  (a - 2 w z))/k

             2          3                   2   2
 + s (wm (- 4 w  w0 z0 zm + 4 w  z zm + 2 a w0  zm - 2 a w  zm)

         3              2          2                2     2    4             2
 + w0 (4 w  z z0 - 2 a w  z0) + wm  (2 a w0 z0 + w0  - w ) + w  (1 - 4 z )

                                  2   2
          3      2  2         a w0  wm
 + 2 a w  z - w  w0 )/k + ---------]
                              k

(c17) ev(d16,[w=1,z=0.01,wm=1,zm=0.5,w0=2,z0=0.707,k=1]);

        2
(d17) [s  + (a + 3.808) s + 2 (1.414 a - 0.02828) + 0.98 a + 6.8084,

(2 (1.414 a - 0.02828) - 0.0196 a + 4 (a - 0.02) + 2 (- 0.02828 a - 1.4134344)

                 2
 + 5.848392) s  + (5.848 a + 2 (0.02828 - 1.414 a) - 2.8084) s + 4 a]

(c18) ev(d17,[a=1]);

        2                                  2
(d18) [s  + 4.808 s + 10.55984, 9.6368032 s  + 0.2681600000000001 s + 4]

(c19) ev(d78,[w=1,z=0.01,wm=1,zm=0.5,w0=2,z0=0.707,a=1]);
```

```
                                                              2
         2                                     9.6368032 s      0.2681600000000001 s   4
(d19) [s  + 4.808 s + 10.55984, ------------ + -------------------- + -]
                                                     k                        k          k
```

COMMENTS

In (c1) the package is loaded into Macsyma. All the listings of the functions have been left out. Command (c12) to (c15) defines the polynomials describing the existing system, the observer and the wanted denominator of the controlled system. (c16) is the call of the polynomial synthesis package and (d16) contains the expressions for R(s) and S(s). (c17) on the next line evaluates the solution (d16) with the values and the result is (d17) where the only remaining variable is a. In (c18) an evaluation of (d17) is performed with the variable a set to 1. In (c19) (d16) is evaluated another time but without a value being assigned to the variable k.

CHAPTER 8

Conclusions

To use LISP as a base for command language has several advantages over writing an entirely new input processor. Firstly a standard program which is maintained by someone else is used, secondly LISP has more power than a homemade system will get with reasonable effort spent on it and thirdly the symbol manipulation functions of LISP will be there for free. Furthermore users that already have experience of LISP need not learn another language and those unfamiliar with LISP will learn something which can be generally used.

If symbol manipulation facilities are included in this programming environment then the possibilities for the user to create his own tools are even greater. These facilities may very well resemble those of MACSYMA, but they need to have a cleaner structure. It is also of great importance that all functions added to the LISP system are thoroughly documented for those functions to be useful.

The input system described above has potential of being a solution for the computer programs to be used in control systems design in the future.

CHAPTER 9

References

Abelson H. and Sussman G. J. (1983): Structure and Interpretations of Computer Programs, Scheme. MIT Press Boston, Mass. and McGraw Hill New York?????

Anonymous (1981): The muMATH/muSIMP-80 Symbolic Mathemayhics System Reference Manual for the Apple II Computer. Document No. 8742-210-04 Microsoft Corp. 10700 Northup Way, Bellevue, WA.

Anonymous (1983): CTRL-C: A Language for the Computer-Aided Design of Multivariable Control Systems, User's Guide. Systems Control Technology, Palo Alto, CA.

Baase S. (1978): Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley Publishing Company, Reading, Mass.

Bagley S. C. and Shrager J. (1982): The P-LISP Tutorial. Pegasys Systems Inc.

Cherry S. (1982): P-LISP Version 3.0 for the Apple II/II+. Pegasys Systems Inc.

Goodfellow S. (1984): Input Processors. Control Systems Centre UMIST.

Goodfellow S. (1984): CILT - A Draft Functional Specification Version 1.0. Control Sustems Centre UMIST.

Mathlab Group Laboratory for Computer Science MIT (1983): MACSYMA Reference Manual Volume One. Massachusetts Institute of Technology & Symbolics Inc. Cambridge Mass.

Moses J. (1984): An Introduction to MACSYMA. Masschusetts Institute of Technology & Symbolics Inc. of Cambridge Mass.

Shrager J. and Bagley S. C. (1982): The P-LISP Tutorial. Pegasys Systems Inc.

Wexelblatt R. L. (editor)(1981): History of Programming Languages. Academic Press, New York, NY.

Wieslander J. (1980): Interactive Programs - General Guide. Dept of Automatic Control, Lund Institute of Technology, Lund, Sweden, Report CODEN: LUTFD2/(TFRT-3156)/1-30/(1980)

Wieslander J. and Elmqvist H. (1978): INTRAC, A Communication Module for Interactive Programs. Language Manual. Dept of Automatic Control, Lund Institute of Technology, Lund, Sweden, Report CODEN: LUTFD2/(TFRT-3149)/1-60/(1978).

Winston P. H. and Horn B. K. P. (1981): LISP. Addison Wesley Publishing Company, Reading, Mass.

Wittenmark B. (1984): Analysis and design of control systems using Ctrl-C. Dept of Automatic Control, Lund Institute of Technology, Lund, Sweden, Report CODEN: LUTFD2/(TFRT-7272)/1-022/(1984)

Åström K. J. (1982): A Simnon Tutorial. Introduction to language. Dept of Automatic Control, Lund Institute of Technology, Lund, Sweden, Report CODEN: LUTFD2/(TFRT-3168)/1-52/(1982).

Åström K. J. (1983): Computer Aided Modeling, Analysis and Design of Control Systems - A Perspective. Control Systems Magazine, May 1983.

Åström K. J. and Schöntal T. (????): Pcalc a Polynomial Calculator - A User's Manual. Dept of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Åström K.J. and Wittenmark B. (1984): Computer Controlled Systems - Theory and Design. Prentice Hall Inc. Englewood Cliffs. N.J.

APPENDIX A

Code for POLIS

```
;****************************************************************************
;*
;*                          POLIS
;*   This package can transform a polynomial from zeros-representation
;*   to coefficient-representation. It can perform formal arithmetics
;*   with two polynomials. It can evaluate a polynomial either with a
;*   real variable or with a complex variable. The coefficients which
;*   are written in infix are converted to prefix notation and ex-
;*   pressions are evaluated.
;*
;****************************************************************************
;*
;*   This version was created 20 February 1985 at LTH
;*
;****************************************************************************



;****************************************************************************
;*
;*   Arithmetic section.
;*   All arithmetic procedures use infix representation
;*
;****************************************************************************

;Adds two polynomials represented by their coefficients starting with the
;degree of zero.
;Example: (addpolc '(a 3 b) '(2 a c b)) returns ((a + 2)(3 + a)(b + c)(b)).
;****************************************************************************

(defun addpol (pol1 pol2)
       (mapcar 'sortout (addpolc pol1 pol2)]

(defun addpolc (pol1 pol2)
       (cond   ((and (null pol1)(null pol2)) nil)
               ((null pol1) pol2)
               ((null pol2) pol1)
               ((zerop (car pol1))(cons(car pol2)
                                   (addpolc(cdr pol1)(cdr pol2))))
               ((zerop (car pol2))(cons(car pol1)
                                  (addpolc(cdr pol1)(cdr pol2))))
               (t (cons (list (car pol1) '+ (car pol2))
                        (addpolc (cdr pol1)(cdr pol2)]
;*************************************************************************
;Subtracts two polynomials represented by their coefficients starting with
;the degree of zero.
;Example (subpolc '(2 3) '(a b c)) returns ((2 - a)(3 - b)(- c))
;*************************************************************************

(defun subpol (pol1 pol2)
       (mapcar 'sortout (subpolc pol1 pol2)]

(defun subpolc (pol1 pol2)
       (cond   ((and (null pol1)(null pol2)) nil)
               ((null pol2) pol1)
               ((null pol1) (mapcar '(lambda (e) (cond ((equal e 0) 0)
                                                    (t (list '- e))))
                                    pol2))
               ((zerop (car pol2))(cons (car pol1)
                                     (subpolc(cdr pol1)(cdr pol2))))
               ((zerop (car pol1))(cons (list '- (car pol2))
                                     (subpolc(cdr pol1)(cdr pol2))))
               (t (cons (list (car pol1) '- (car pol2))
                        (subpolc (cdr pol1)(cdr pol2)]


;*************************************************************************
;These functions multiply two polynomials represented by their coefficients
;starting with the zero degree coefficient.
;Example: (multpolc '(1 2)'(a b)) returns ((1 * a)((1 * b) + (2 * a))
;(2 * b)).
;*************************************************************************

(defun multpol (pol1 pol2)
       (mapcar 'sortout (multpolc pol1 pol2)]

(defun multpolc (pol1 pol2)
       (cond  ((or (null pol1)(null pol2))nil)
              ((lessp (length pol1)(length pol2))(multpaux 0 pol1 pol2))
              (t (multpaux 0 pol2 pol1)]

(defun multpaux (exponent short long)
       (cond  ((null short) nil)
              ((zerop (car short))(multpaux (add1 exponent) (cdr short) long))
              (t (addpolc (mult1paux 0 exponent (car short) long)
                      (multpaux (add1 exponent)(cdr short) long)]
```

```
(defun mult1paux (m termexp term pol)
        (cond  ((null pol) nil)
               ((equal 0 term)(list 0))
               ((lessp m termexp)
                (cons 0 (mult1paux (add1 m) termexp term pol)))
               ((equal 1 term) pol)
               ((equal 0 (car pol))
                (cons 0 (mult1paux m termexp term (cdr pol))))
               ((equal 1 (car pol))
                (cons term (mult1paux m termexp term (cdr pol))))
               (t (cons (list term '* (car pol))
                        (mult1paux m termexp term (cdr pol))]


;*****************************************************************************
;Gives you div of the two polynomials.
; Examples:
; (ax^2 + bx + c) div (dx^2 + ex + f) equals a/d and
; (ax^2 + bx + c) div d equals a/d*x^2
;*****************************************************************************

(defun divpol (pol1 pol2)
       (mapcar 'sortout (divpolc pol1 pol2)]


(defun divpolc (pol1 pol2)
       (divaux (car (reverse pol1))(car (reverse pol2))
               (diff (length pol1)(length pol2)]

(defun divaux (first1 first2 n)
       (cond ((lessp n 0)(print '(This div operation is not possible)))
             ((zerop n)(list (list first1 '/ first2)))
             (t (cons 0 (divaux first1 first2 (sub1 n))]


;*****************************************************************************
;Gives you mode of the two polynomials.
;Pol1 mode Pol2 can be defined as:
;Pol1 - (Pol1 div Pol2)*Pol2
;*****************************************************************************

(defun modpol (pol1 pol2)
       (mapcar 'sortout (modepolc pol1 pol2)]

(defun modepolc (pol1 pol2)
       (subpolc pol1 (multpolc (list (divpolc pol1 pol2)) pol2]

(defun firstn (lista n)
       (cond ((equal n 0) nil)
             (t (append (list '* (car lista))(firstn(cdr lista)(sub1 n)]
(defun kombn (lista n)
      (cond ((lessp (length lista) n) nil )
            (t (append (list '+ (cons (car lista)(firstn(cdr lista)(sub1 n))))
                       (kombn (cdr lista) n)]

(defun komb (lista n)
      (cond ((equal n 1) (list '1))
            ((equal (length lista) n) (cons (car lista)(firstn(cdr lista)
                                                          (sub1 n))))
            (t (cons (cons (car lista)(firstn (reverse lista)(sub1 n)))
                     (kombn lista n)]

(defun sortout (exp)
      (calcnumbers (numbersfirst exp)]


;****************************************************************************
; Poles-to-coeffs converts a polynomial represented by its poles (real or
; complex) to coefficients representation. Complex poles must be complex-
; conjugated and multiplied to a second order polynomial with real
; coefficients.
;****************************************************************************


(defun poles-to-coeffs (polyn)
      (mapcar 'sortout (p-t-c polyn)]

(defun p-t-c (polyn)
      (cond ((and (null (cdr polyn))(atom (car polyn)))(list (car polyn) 1))
            ((null (cdr polyn))(reverse (cons 1 (car polyn))))
            ((atom (car polyn))(multpolc(list (car polyn) 1)
                                        (p-t-c (cdr polyn))))
            ( t (multpolc (reverse (cons 1 (car polyn)))
                          (p-t-c (cdr polyn))]

;********************************************************************************
;
; Evalpol is a polynomial evaluator . If x is a real number evalpolr will per-
; form the evaluation, if x is a complex number evalpolc will do it in stead.
;
;******************************************************************************


(defun evalpol (x pol)
      (cond ((atom x)(evalpolr x pol))
            (t (evalpolc x pol)]
;****************************************************************************
; Evalpolr is a polynomial evaluator
; It operates with Horners scheme
; x is a number, possibly negative and pol is the
; description of the polynomial
;****************************************************************************

(defun evalpolr ( x pol )
       (cond ((equal nil pol)
              'no_polynomial)
             (( not (cdr pol))
              (getnum (car pol)))
             (t (add (getnum(car pol))
                     (times x (evalpolr x (cdr pol))]


;****************************************************************************
; This is another polynomial evaluator where xc is
; a complex number like (3 -4) which means 3 - 4i
;****************************************************************************

(defun evalpolc ( xc pol )
       (cond ((equal nil pol)
              'no_polynomial)
             (( not (cdr pol))
              (list (getnum (car pol)) 0))
             (t (cons (add (getnum (car pol))
                           (car (compmul xc
                                         (evalpolc xc (cdr pol)))))
                      (cdr (compmul xc (evalpolc xc (cdr pol))]

;****************************************************************************
;This function performs an complex multiplication
;****************************************************************************

(defun compmul ( comp1 comp2 )
       (list (difference (times (car comp1)(car comp2))
                         (times (cadr comp1)(cadr comp2)))
             (add (times (car comp1)(cadr comp2))
                  (times (cadr comp1)(car comp2)))]

(defun getnum (infexpr)
       (eval (inf-to-pre infexpr)]
;******************************************************************************
; In this file I intend to try patternmatching to do some things
; for example inverting infix to prefix notation.
; This matcher can also give atoms in the patterns values as it
; moves along in the matching. Atoms that begin with > and + act
; as > and + for matching purposes, but if match succeeds, there
; values are made to be whatever they matched.
;******************************************************************************


(defun match ( p d )
       (cond ((and (null p)(null d)) t)
             ((or (null p)(null d)) nil)
             ((and (not (atom (car p)))          ;Restricted >.
                   (equal (caar p) 'restrict)
                   (equal (cadar p) '>)
                   (testpred (cddar p)(car d)))
              (match (cdr p)(cdr d)))            ;Restricted > variable.
             ((and (not (atom (car p)))
                   (equal (caar p) 'restrict)
                   (equal (atomcar (cadar p)) '>)
                   (testpred (cddar p)(car d))
                   (match (cdr p)(cdr d)))
              (set (atomcdr (cadar p))(car d))
              t )
             ((or (equal (car p) '>)             ;Equality or >.
                  (equal (car p)(car d)))
              (match (cdr p)(cdr d)))
             ((and (equal (atomcar (car p)) '>)   ;> variable
                   (match (cdr p)(cdr d)))
              (set (atomcdr (car p))(car d))
              t)
             ((equal (car p) '+)                 ;+
              (cond ((match (cdr p)(cdr d)))      ;drop +
                    ((match p (cdr d)))))        ;keep +
             ((equal (atomcar (car p)) '+)        ;+ variable
              (cond ((match (cdr p)(cdr d))
                     (set (atomcdr (car p))(list (car d)))
                     t)
                    ((match p (cdr d))
                     (set (atomcdr (car p))
                          (cons (car d)(eval (atomcdr (car p)))))
                          t ]
```

```
;***************************************************************************
; Atomcar and atomcdr returns the first letter of "word"
; and the rest of it respectively.
;***************************************************************************

(defun atomcar (word)
       (car (explode word)))


(defun atomcdr (word)
       (implode (cdr (explode word))))


;***************************************************************************
; Testpred tests if the predicates "predicates" are fulfilled
; by "argument", if so it returns t.
;***************************************************************************

(defun testpred (predicates argument)
       (prog ()
         loop
            (cond ((null predicates)(return t)))       ; All tests T?
            (cond ((funcall (car predicates) argument); This test T?
                   (setq predicates (cdr predicates))
                   (go loop))
                  (t (return nil))]



;***************************************************************************
;Here comes a converter between infix and prefix notation.
;It will never truncate when performing division.
;***************************************************************************

(defun inf-to-pre (e)
       (prog (v l r)
             (return
              (cond ((atom e) e)
                    ((match '(>v) e)
                     (inf-to-pre v))
                    ((match '(+l (restrict > oneplus) +r) e)
                     (list 'plus (inf-to-pre l)(inf-to-pre r)))
                    ((match '(+l - +r) e)
                     (list 'difference (inf-to-pre l)(inf-to-pre r)))
                    ((match '(+l * +r) e)
                     (list 'times (inf-to-pre l)(inf-to-pre r)))
                    ((match '(+l / +r) e)
                     (list 'quotient (float (inf-to-pre l))(inf-to-pre r)))
                    ((match '(+l ^ +r) e)
                     (list 'expt (inf-to-pre l)(inf-to-pre r)))
                    ((match '(- +r) e)
                     (list 'minus (inf-to-pre r)))
                    (t e)]


;******************************************************************************
; For avoiding confusion between the match symbol + and the arithmetic
; symbol + the following predicate is used.
;******************************************************************************

(defun oneplus (X)
       (equal X '+))




;******************************************************************************
;This function will place numbers first in the expression exp.
;******************************************************************************

(defun numbersfirst (exp)
       (prog (new) (setq new (numbersfirsta exp))
                   (cond ((or (atom new)(noop (car new))(equal (car new) '-))
                          (return new))
                         ((equal (car new) '/)(return (cons 1 new)))
                         (t (return (cdr new))]

(defun numbersfirsta (exp)
       (cond ((not exp) nil)
             ((atom exp) exp)
             ((nonumbers exp) exp)
             ((and (noop (car exp))(numberp (car exp)))       ; (3 + ...
              (cons (car exp)(numbersfirsta (cdr exp))))
             ((and (noop (car exp))(atom (car exp)))          ; (a + ...
              (inlast (cadr exp)(car exp)(numbersfirsta (cdr exp))))
             ((and (noop (car exp))(onlynumbers (car exp)))   ; ((3 + 2) * ..
              (cons (car exp)(numbersfirsta (cdr exp))))
             ((noop (car exp))                                ; ((3 + a) * ..
              (inlast (cadr exp)(numbersfirst (car exp))
                      (numbersfirsta (cdr exp))))
             ((numberp (cadr exp))                            ; (- 4 +...
              (append (list (car exp)(cadr exp))(numbersfirsta (cddr exp))))
             ((atom (cadr exp))                               ; (+ a - ....
              (append (numbersfirsta (cddr exp))(list (car exp)(cadr exp))))
             ((onlynumbers (cadr exp))                        ; (+ (3 * 2) - ..
              (append (list (car exp)(cadr exp))(numbersfirsta (cddr exp))))
             (t                                               ; (+ (2 * a) - ..
              (append (numbersfirsta (cddr exp))
                      (list (car exp)(numbersfirst (cadr exp))))]
```

```
;*****************************************************************************
; Onlynumbers returns t if exp only contains numbers and operators (+,-,*,/)
;*****************************************************************************

(defun onlynumbers (exp)
       (cond ((not exp) t)
             ((and (atom exp)(numberp exp) t)
             ((atom exp) nil)
             ((or (numberp (car exp))(operp (car exp))
                  (and (listp (car exp))(onlynumbers (car exp))))
              (onlynumbers (cdr exp)))
             (t nil)]

;*****************************************************************************
; Nonumbers returns t if exp doesn't contain any number.
;*****************************************************************************

(defun nonumbers (exp)
       (cond ((or (not exp)(and (atom exp)(not (numberp exp)))) t)
             ((atom exp) nil)
             ((or (and (atom (car exp))(not (numberp (car exp))))
                  (nonumbers (car exp)))
              (nonumbers (cdr exp)))
             (t nil)]


(defun operp (a)
       (cond ((and (not (listp a))(or (= a '+)(= a '-)(= a '*)(= a '/)))
             t]

(defun noop (a)
       (not (operp a)]

(defun inlast (sign part whole)
       (cond ((or (equal sign '-)(equal sign '+))
              (append whole (list '+ part)))
             (t
              (append whole (list '* part))]


;*****************************************************************************
; Calcnumbers will calculate the numbers in an expression which first has
;been processed numbersfirst
;*****************************************************************************

(defun calcnumbers (e)
       (prog (result)(cond ((atom e)(setq result (list e)))
                           (t (setq result (calcaux (reverse e)))))
                  (return
                    (cond  ((equal (cdr result) nil)(car result))
                           (t (reverse result)]
(defun calcaux (exp)
       (cond ((not exp) nil)                              ;exp = ()
             ((atom exp) (list exp))                      ;exp = atom
             ((and (listp (car exp))(not (cdr exp)))      ;exp = ((exp2))
              (calcaux (reverse (car exp))))
             ((and (listp (car exp))(onlynumbers (car exp))) ;exp = ((2 + 2))
              (calcaux (cons (calcnumbers (car exp)) (cdr exp))))
             ((listp (car exp))                           ;exp = ((exp2)+..))
              (cons (calcnumbers (car exp))(calcaux (cdr exp))))
             ((not (numberp (car exp)))                      ;exp = ( a + ...
              (cons (car exp)(calcaux (cdr exp))))
             (t (list (eval (inf-to-pre (reverse exp))]   ;exp = ( 3 + ...


;****************************************************************************
;
; Info is a function to provide information on commands in POLIS and other
; subjects closely related to POLIS.
;
;****************************************************************************


(def info (nlambda (command)
       (prog ()
       (cond ((equal (car command) 'addpol)                        ;ADDPOL
              (print 'ADDPOL)(terpr)(terpr)
              (print '(Performs an addition between two polynomials represented
                       by their coefficients))
              (terpr)
              (return '->))
             ((equal (car command) 'all)                           ;ALL
              (print 'ALL)(terpr)(terpr)
              (info 'addpol)(terpr)
              (info 'commands)(terpr)
              (info 'divpol)(terpr)
              (info 'evaluate)(terpr)
              (info 'info)(terpr)
              (info 'multpol)(terpr)
              (info 'poles-to-coeffs)(terpr)
              (info 'subpol)(terpr)
              (info '?)(terpr)
              (return '->))
             ((equal (car command) 'commands)                      ;COMMANDS
              (print 'COMMANDS)(terpr)(terpr)
              (print '(The following POLIS commands are available ))
              (terpr)
              (print '(addpol subpol multpol divpol modepol))
              (terpr)
              (print '(poles-to-coeffs evaluate help))
              (terpr)
              (return '->))
```

```
((equal (car command) 'divpol)                      ;DIVPOL
 (print 'DIVPOL)(terpr)(terpr)
 (return '->))
((equal (car command) 'info)                        ;INFO
 (print 'INFO)(terpr)(terpr)
 (print '(Info is a function to provide information on POLIS
         commands))
 (terpr)
 (print '(and subjects related to POLIS.  The arguments available
          in info))
 (terpr)
 (print '(are given by (info ?)))
 (terpr)
 (return '->))
((equal (car command) 'modepol)                     ;MODEPOL
 (print 'MODEPOL)(terpr)(terpr)
 (print '->))
((equal (car command) 'multpol)                     ;MULTPOL
 (print 'MULTPOL)(terpr)(terpr)
 (print '->))
((equal (car command) 'subpol)                      ;SUBPOL
 (print 'SUBPOL)(terpr)(terpr)
 (print '(Performs a subtraction between two polynomials
          represented by their coefficients))
 (terpr)
 (return '->))
((equal (car command) '?)                         ;?
 (print '?)(terpr)(terpr)
 (print '(Info can be obtained for the following arguments))
 (terpr)
 (print '(addpol subpol multpol divpol modepol))
 (terpr)
 (print '(poles-to-coeffs evaluate info ))
 (terpr)
 (print '(all commands polynomial ?))
 (terpr)
 (return '->))

 (t                                                ;UNKNOWN
  (print (append '(There is no information available on)
                   (list (car command))))
   (terpr)
   (print '(To obtain additional information type (info ?) ))
   (terpr)
   (return '->)
                          ]
```

APPENDIX B

Code for MACPOL

```
/* Four input functions as interface to the user */

polsynthc (apol,bpol,a0pol,ampol) := block ([],

           /* Expand all polynomials to get them on termform */
        apol : expand(apol),
        bpol : expand(bpol),
        a0pol : expand(a0pol),
        ampol : expand(ampol),

           /* Check if all conditions are fullfilled otherwise */
           /* send error message */

        if is (apol = 0) then
                error ( "A-polynomial is zero" ),
        if is (bpol = 0) then
                error ( "B-polynomial is zero" ),
        if is (a0pol = 0) then
                error ( "A0-polynomial is zero" ),
        if is (ampol = 0) then
                error ( "Am-polynomial is zero" ),
        if is (hipow(apol,s) < hipow (bpol,s)) then
                error ( "degree of Apol less then degree of Bpol"),
        if is (hipow(expand(a0pol*ampol),s) < hipow (apol,s)) then
                error ( "degree of A0*Am less then degree of Apol"),

        axbyc(apol,bpol,a0pol*ampol,s) )$


polsynthd (apol,bpol,a0pol,ampol) := block ([],

           /* Expand all polynomials to get them on termform */
        apol : expand(apol),
        bpol : expand(bpol),
        a0pol : expand(a0pol),
        ampol : expand(ampol),

           /* Check if all conditions are fullfilled otherwise */
           /* send error message */
        if is (apol = 0) then
                error ( "A-polynomial is zero" ),
        if is (bpol = 0) then
                error ( "B-polynomial is zero" ),
                if is (a0pol = 0) then
                        error ( "A0-polynomial is zero" ),
                if is (ampol = 0) then
                        error ( "Am-polynomial is zero" ),
                if is (hipow(apol,z) < hipow (bpol,z)) then
                        error ( "degree of Apol less then degree of Bpol"),
                if is (hipow(expand(a0pol*ampol),z) < hipow (apol,z)) then
                        error ( "degree of A0*Am less then degree of Apol"),

                axbyc(apol,bpol,a0pol*ampol,z) )$


polsynthbc (apol,bminuspol,bpluspol,a0pol,ampol) := block ([],

                  /* Expand all polynomials to get them on termform */
                apol : expand(apol),
                bminuspol : expand(bminuspol),
                bpluspol : expand(bpluspol),
                a0pol : expand(a0pol),
                ampol : expand(ampol),

                  /* Check if all conditions are fullfilled otherwise */
                  /* send error message */
                if is (apol = 0) then
                        error ( "A-polynomial is zero" ),
                if is (bminuspol = 0) then
                        error ( "Bminus-polynomial is zero" ),
                if is (bpluspol = 0) then
                        error ( "Bplus-polynomial is zero" ),
                if is (a0pol = 0) then
                        error ( "A0-polynomial is zero" ),
                if is (ampol = 0) then
                        error ( "Am-polynomial is zero" ),
                if is (hipow(apol,s)
                    < hipow (expand(bminuspol*bpluspol),s)) then
                        error ( "degree of Apol less then degree of Bpol"),
                if is (hipow(expand(bpluspol*ampol*ampol),s)
                    < hipow (apol,s)) then
                        error ( "degree of Bplus*A0*Am
                                less then degree of Apol"),

                axbyc(apol,bpluspol*bminuspol,bpluspol*a0pol*ampol,s) )$


polsynthbd (apol,bminuspol,bpluspol,a0pol,ampol) := block ([],

                  /* Expand all polynomials to get them on termform */
                apol : expand(apol),
                bpol : expand(bminuspol),
                bpluspol : expand(bpluspol),
                a0pol : expand(a0pol),
                ampol : expand(ampol),
                    /* Check if all conditions are fullfilled otherwise */
                    /* send error message */
                 if is (apol = 0) then
                        error ( "A-polynomial is zero" ),
                 if is (bminuspol = 0) then
                        error ( "Bminus-polynomial is zero" ),
                 if is (bpluspol = 0) then
                        error ( "Bplus-polynomial is zero" ),
                 if is (a0pol = 0) then
                        error ( "A0-polynomial is zero" ),
                 if is (ampol = 0) then
                        error ( "Am-polynomial is zero" ),
                 if is (hipow(apol,z)
                      < hipow (expand(bminuspol*bpluspol),z)) then
                        error ( "degree of Apol less then degree of Bpol"),
                 if is (hipow(expand(bpluspol*ampol*ampol),z)
                      < hipow (apol,z)) then
                        error ( "degree of Bplus*A0*Am
                                 less then degree of Apol"),

                 axbyc(apol,bpluspol*bminuspol,bpluspol*a0pol*ampol,z) )$


/* Main function */


axbyc (apol,bpol,cpol,base) := block ([degx,degy,inter,xandy,i,solution,
                    xpol,ypol],


                    /* Expand all polynomials to get them on term form */
                 apol : expand (apol),
                 bpol : expand (bpol),
                 cpol : expand (cpol),

                    /* Check if all conditions are fullfilled otherwise */
                    /* send error message */
                 if is (apol = 0) then
                        error ( "A-polynomial is zero" ),
                 if is (bpol = 0) then
                        error ( "B-polynomial is zero" ),
                 if is (cpol = 0) then
                        error ( "C-polynomial is zero" ),
                 if is (hipow(apol,base) < hipow (bpol,base)) then
                        error ( "degree of Apol less then degree of Bpol"),
                 if is (hipow(cpol,base) < hipow (apol,base)) then
                        error ( "degree of Cpol less then degree of Apol"),

                    /* Compute the degrees of X and Y polynomials */
                 degx : hipow(cpol,base) - hipow(apol,base),
                 degy : hipow(apol,base) - 1,
                 /* Create X and Y of the right degrees, observe that */
                 /* X is monic */
               xpol : createx(base, degx),
               ypol : createy(base, degy),

                 /* Inter is the left side of the equation; AX+BY=C */
               inter : expand(apol * xpol + bpol * ypol),

                 /* xandy contains all unknown xn and yn */
               xandy : removelist(base , listofvars (xpol + ypol)),

                 /* equsys creates the system of equations which is solved */
               solution : first(solve (equsys(inter,cpol,base),xandy )),

                 /* xpol and ypol are evaluated */
                 /* "solution" contains values to be applied before evaluation */
               return (ev ( [xpol,ypol], solution ))
               )$


/* creates a polynomial of the base 'base' with X-coefficients from */
/* X0 (lowest coefficient) to Xdeg-1 (i.e. Xpol is monic) */

createX(base,deg) := block ([xpol,x,i],

               if is (deg = 0) then
                    (xpol : 1)
               else
                    (xpol : 1,
                    for i : deg-1 step -1 thru 0 do
                         (xpol:xpol * base + concat(x,i)) ),
                    return (expand(xpol)) )$


/* creates a polynomial of the base 'base' with Y-coefficients from */
/* Y0 (lowest coefficient) to Ydeg */

createY(base,deg) := block ([ypol,y,i],
               ypol : concat(y,deg),
               for i :deg-1 step -1 thru 0 do
                    (ypol: ypol * base + concat(y,i)),
               return (expand(ypol)) )$
```

```
/* creates an equation system from two polynomials of the base 'base' */
/* and of the same degree by equaling the coefficients of the both */
/* polynomials for each exponent */

equsys(exp1,exp2,base) := block ([list],
              list : [],
              for i : hipow(exp1,base) step -1 thru 0 do
                    (list:cons(coeff(exp1,base,i)=coeff(exp2,base,i), list)),
              return(list) )$


/* removes var from the list 'list' if var is a member of list */

removelist(var,list) :=
              if is (list = [] )
                    then []
              else if is ( first(list) = var )
                    then removelist(var,rest(list))
              else
                    cons(first(list) , removelist(var,rest(list)))$
```