

SIMATH-Manual

An Introduction to the Computer Algebra System

SIMATH

and the Calculator

SIMCALC

Contents

Chapter 1. Introduction to SIMATH	5
1. Introducing SIMATH	5
2. The logical structure of SIMATH	7
3. The installation of SIMATH	8
Chapter 2. The Interface	9
1. Starting SIMATH	9
2. Example session	9
3. Description of SIMATH commands	17
Chapter 3. Programming in SIMATH	33
1. The extended programming language C	33
2. Include files in SIMATH	33
3. Data types in SIMATH	34
4. Example	39
5. Nested function calls	42
6. Nomenclature of SIMATH functions	43
7. Global parameters in SIMATH	46
Chapter 4. The SIMATH system	51
1. The base system	51
2. The arithmetic package	67
3. The polynomial package	80
4. The matrix/vector package	89
5. The elliptic curves package	98
6. The NON-SIMATH package	108
Chapter 5. The calculator simcalc	109
1. What is simcalc?	109

2. Editing command lines	110
3. Example session	111
4. Operators and functions	116
5. Variable administration	133
6. Computing in $\mathbb{Z}/m\mathbb{Z}$	136
7. Computing in Galois fields	137
8. Computing in number fields	138
9. Elliptic curves	139
10. Computing over \mathbb{R} and \mathbb{C}	140
11. Variable substitution	140
12. Reading from files	141
13. Predefinitions	141
14. Log files	142
15. Statistical functions	143
16. Suppressing the output	143
17. Interrupting output and computations	143
18. Special characters as commands	144
19. Error messages	145

CHAPTER 1

Introduction to SIMATH

1. Introducing SIMATH

SIMATH, that is *SInix-MATHematik*, is a computer algebra system focusing mainly on *algebraic number theory*. It is being developed at the Universität des Saarlandes in Saarbrücken (Germany). System development started on SINIX PCs and currently, we are running SIMATH on

- HP 9000 series 700 under HP-UX 9.0x and HP-UX 10.x
- SGI machines under IRIX 5.3
- Sun SPARCstation under SunOS 4.1.1
- Intel based PCs under Linux 1.x and 2.x

It should not be difficult to compile and run SIMATH on most 32 bit UNIX platforms. For example, SIMATH is known to run on SPARCstations under Solaris 2.x and Intel based PCs under FreeBSD 2.x

How does SIMATH differ from other systems ?

The difference between SIMATH and most other systems lies essentially in

- the main area of application: *algebraic number theory*.
- the concept of the system: SIMATH is a *transparent system* which means that all sources are part of the system so the user can adapt existing general algorithms to specific problems and integrate her/his own algorithms at any point within the system.
- the programming language: SIMATH is written in C; likewise, the user works in C. SIMATH functions are integrated into a C-program simply by function calls.

SIMATH may also be accessed via the interactive calculator *simcalc* which features

- many of the existing SIMATH algorithms,
- comprehensive error checking,
- detailed “help facilities”.

This makes *simcalc* particularly suitable for a quick calculation on the side and users with little programming experience.

We have set up an experimental mailing list related to the SIMATH computer algebra system. If you want to subscribe to this list send e-mail to

majordomo@@emmy.math.uni-sb.de

and include the line

subscribe simath-list

in the body of your mail. The purpose of the list is to announce new versions of SIMATH and to discuss anything which is related to SIMATH and of public interest.

The latest version of SIMATH may be obtained by anonymous ftp from ftp.math.uni-sb.de (134.96.32.23) in the directory /pub/simath. For more information about the installation of SIMATH see §1.3.

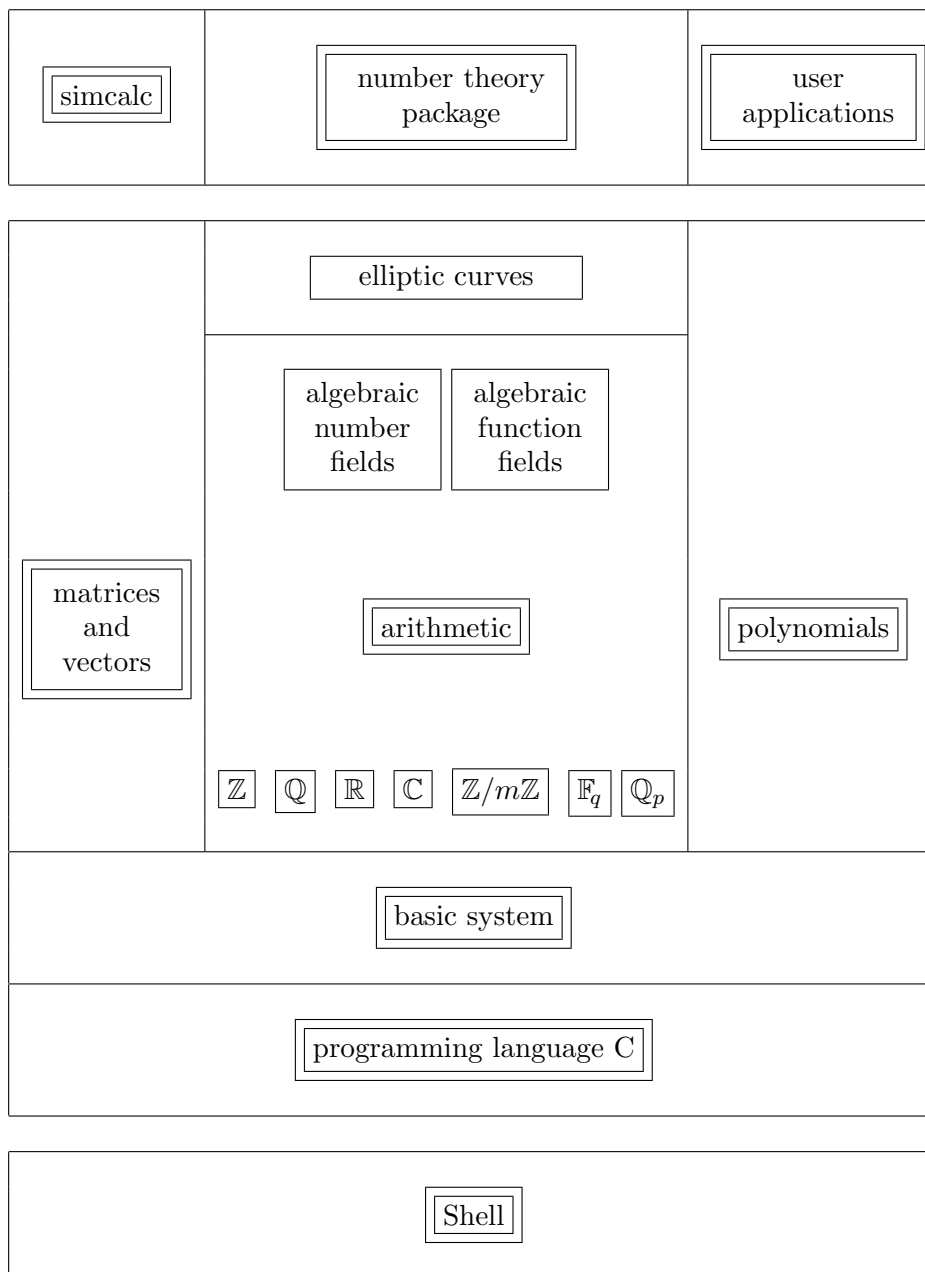
If you have any questions or problems, please contact the following address:

SIMATH-Gruppe
Lehrstuhl Prof. Dr. H.G. Zimmer
Fachbereich 9 Mathematik
Universität des Saarlandes
Postfach 151150
D-66041 Saarbrücken

e-mail: simath@@math.uni-sb.de
Phone: +49 681 3022206

2. The logical structure of SIMATH

The following diagram illustrates the structure of the system.



SIMATH consists of

- an *interface* between the operating system and SIMATH;
- the *programming language C*;
- the *basic* system which consists of modified input/output functions, and a *list* system with an automatic *garbage collector* and dynamic memory administration;
- a *multiple precision arithmetic* package for computations over \mathbb{Z} , \mathbb{Q} , \mathbb{R} , \mathbb{C} , $\mathbb{Z}/m\mathbb{Z}$, \mathbb{Q}_p , finite fields, and global fields, i.e. algebraic number fields and function fields;
- a *polynomial* package for computations with polynomials in any number of unknowns over any of the structures contained in the arithmetic package;
- a *matrix-vector* package for matrix/vector computations over the structures contained in the arithmetic package and over polynomial rings;
- an *elliptic curves* package with elliptic-curve-specific functions over the rational numbers, prime fields, finite fields of characteristic 2 and algebraic number fields;
- the interactive SIMATH calculator *simcalc*.

The *number theory* package contains higher algorithms for algebraic number theory such as

- integral bases, extension of valuations, and the decomposition law for number fields and congruence function fields;
- for quadratic congruence function fields: regulators, unit groups, divisor and ideal class number, and generators and type of isomorphism of the ideal class group and the zero class group;
- conductor, minimal model, and an algorithm for finding the rank and a basis of an elliptic curve over rational numbers, as well as Tate's algorithm over the rational numbers and quadratic number fields;
- combined Schoof-Shanks algorithm for counting points on elliptic curves over prime fields and finite fields of characteristic 2;
- an algorithm for constructing elliptic curves with a given number of points over a given prime field;
- LLL-algorithm.

The system libraries each consist of a package of functions which – except for some internal initialization and managing procedures – can be integrated into any C program by a simple function call.

3. The installation of SIMATH

The SIMATH system is installed by Makefiles and shell scripts. You will need about 50 MB disk space while compiling SIMATH. On all platforms the GNU readline library will be used in *simcalc*. This has many advantages such as emacs style command line editing and function name completion. Starting with Version 3.10.3, the readline library is part of the SIMATH distribution (see `./readline-2.0`) and compiled automatically.

The latest version of SIMATH and more information about installing the system may be obtained by anonymous ftp from `ftp.math.uni-sb.de` (134.96.32.23) in the directory `/pub/simath`.

CHAPTER 2

The Interface

The SIMATH system uses a special *interface* to its libraries permitting single-letter commands for editing, compiling, and printing of SIMATH programs.

1. Starting SIMATH

From your system shell type the command ¹

% **SM** ↵

SIMATH's introductory screen will appear followed by the five-line menu of SIMATH commands and the SIMATH prompt symbol < on the last line (see p. 10). SIMATH is now ready to receive your command.

2. Example session

We now give an example of a SIMATH session to show how the interface works. For now, we shall restrict the examples to C programs. Later we will develop actual SIMATH programs. We will create two C functions, **gcd()** and **lcm()**, then use these in a short program, the **main()** function.

Note: SIMATH files are not necessarily located under **/usr/local/simath**. Any directory can be chosen upon installation, for example **/usr/simath**. Here, we use **/usr/local/simath** as the default directory. For this session, we assume that the file **.SM.default** does not exist (see §2.4.1).

¹We use % as the standard prompt symbol.

We now begin by executing SM

% SM ↵

```
*****
*
*      S I M A T H      *
*
*****
```

Version 4.2 started at 11:34:17

no source library
no object library
no file name

(N)AME:	(e,E)dit	(p)re	(o)bj	(c)omp	(C)comp
object:	(a)r name	(d)ir	ar(+)obj	ar(-)obj	(r)anlib
USR-SRC:	(A)R NAME	(D)IR	AR(*)SRC	AR(_)SRC	(S)ELECT
SM-SRC:	E(X)TRACT	(~,@,.)			
	(!) (\$) (?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT

<

SIMATH commands are abbreviated and listed in a five-line menu. For each command, it suffices to type a single letter (given in parenthesis) followed by the RETURN key (except for \$).

Type **N** at the prompt to name the first C function to be created.

< **N** ↵
file name:gcd ↵
<

All input must be followed by RETURN. Typing errors can be corrected by using the BACKSPACE and/or the DELETE key (depending on your machine).

Invoke an editor (either with **e** or with **E**; see §2.4.1) and type in the following program; it will automatically be saved under the name **gcd.S**.

```
< e ↵
#include <_simath.h>

int      gcd(a, b)
  int      a, b;
{
  int      c;

  if (a < b)
    {c = a; a = b; b = c;}

  while (b != 0)
    {c = a % b; a = b; b = c;}

  return (a);
}

<
```

Type **H** to display the SIMATH menu (if necessary).

```
< H ↵

no source library
no object library
current file name:      gcd

(N)AME:      (e,E)dit      (p)re      (o)bj      (c)omp      (C)comp
object:      (a)r name      (d)ir      ar(+)obj      ar(-)obj      (r)anlib
USR-SRC:      (A)R NAME      (D)IR      AR(*)SRC      AR(_)SRC      (S)ELECT
SM-SRC:      E(X)TRACT      (~,@,.)
              (!) ($) (?)      (P)RINT      (H)ELP      (R)UN      (Q)UIT

<
```

We now want to create an *object module library* which can archive our compiled subroutines and make them available to the linker for later use. (Notice that all the libraries you create will be located in **/usr/local/simath/lib**.)

```
< a ↵
object library: /usr/local/simath/lib/lib*.a,
               * = example ↵
<
```

The library **libexample.a**, in the directory **/usr/local/simath/lib**, is now the working library for SIMATH. We compile the program **gcd.S** to the object module **gcd.o**. A *preprocessor* creates a temporary file **gcd.c** and passes it to the C compiler.

< o ↵

SM preprocessor started at 11:48:57

SM preprocessor terminated correctly at 11:48:58

Compiler started at 11:48:58

Compiler terminated correctly at 11:49:03

If the compiler reports an error, edit the program, using **e** or **E**, to make the necessary corrections.

We now add the object module **gcd.o** to the library.

< + ↵

ar: creating /usr/local/simath/lib/libexample.a
'ar r /usr/local/simath/lib/libexample.a gcd.o' done

<

If the library **libexample.a** already exists, only the line

'ar r /usr/local/simath/lib/libexample.a gcd.o' done

will appear on your screen. If the file **gcd.o** is already in the library, you will get the following message on your screen:

SM: 'gcd' already exists in '/usr/local/simath/lib/libexample.a'

rw-r--r-- 431/42 164 Sep 2 09:13 1996 gcd.o

overwrite? (y/n)

You must decide if you want to overwrite the existing file or not.

Now use the commands **N** (new file name: lcm), **e**, **o**, **+** to create a second C function.

```
#include <_simath.h>

int          lcm(a, b)
  int        a, b;
{
  int        c;

  /* Result zero */
  if (a == 0 || b == 0)
    c = 0;
  else
  {
    /* General case */
    a = abs(a); b = abs(b); c = a / gcd(a, b);

    /* if result representable */
    if (~(1 << 31) / b >= c)
      c *= b;

    /* Replacement value if result too large */
    else
      c = -1;
  }
  return (c);
}
```

If you need help, enter **H**. The commands referring to your current module library are listed on the second line of the menu. Before we turn to the main program, we will run **ranlib** and look at the contents of our object module library.

Note: On many machines it is not necessary to apply the **ranlib** command to a library. The command on the menu calls the corresponding (inoperative) command.

< **r** ↵

'ranlib /usr/local/simath/lib/libexample.a' done

Let us look at the library contents.

```
< d ↵
(rw-r--r--431/42      34 Sep 2 11:57 1996 __.SYMDEF)
rw-r--r--431/42      164 Sep 2 11:51 1996 gcd.o
rw-r--r--431/42      332 Sep 2 11:57 1996 lcm.o
```

The file __.SYMDEF is the table of contents of our library created by the **ranlib** command.

We now come to the main program.

```
< N ↵
current file name :  lcm
new file name      :  mainp
< e ↵

#include <_simath.h>

main()
{
    int          a, b, c;
    char          as[11], bs[11];

    printf("\nEnter the numbers: \na = ");
    while (gets(as) == NULL)
        printf("Reading error \na = ");
    printf("b = ");
    while (gets(bs) == NULL)
        printf("Reading error \nb = ");

    /* converting the strings to numbers */
    a = atoi(as);
    b = atoi(bs);

    /* lcm */
    if ((c = lcm(a, b)) == -1)
        printf("Result is too large\n");
    else
        printf("lcm( %d, %d ) = %d\n", a, b, c);
}
```

Compile and link the program **mainp.S** with the command

```
< c ↵
```

```
SM preprocessor started at 12:02:42
```

```
SM preprocessor terminated correctly at 12:02:43
```

```
Compiler started at 12:02:43
```

```
Compiler terminated correctly at 12:03:03
```

The linker uses the modules from the library **libexample.a**. If the compiler or the linker reports an error, edit the main program, using **e** or **E**, to make the necessary corrections. The **executable program** is now stored under the name **mainp.x**. Type **R** to run it. Here is an example.

```
< R ↵
```

```
RUN "mainp"
```

```
Enter the numbers:
```

```
a = 55 ↵
```

```
b = 22 ↵
```

```
lcm( 55, 22 ) = 110
```

```
*** # GCs: 1          GC time: 0.00 s      # collected cells: 0      ***
*** # blocks: 1       block size: 16383    # free cells:      16376   ***
*** total CPU time: 0.03 s                  ***
```

```
<
```

At the end of the execution, we automatically get information about memory administration

- number of garbage collections (**# GCs**),
- total time (in seconds) spent garbage collecting (**GC time**),
- total number of memory cells collected during garbage collection (**# collected cells**),
- number of memory blocks used for the computation (**# blocks**),
- size (in cells) of one block (**block size**),
- number of free memory cells (**# free cells**),

and the total computation time (in seconds) (**total CPU time**), including the time spent for memory administration.

A second test: type in **R** once again.

```
< R ↵
```

```
RUN "mainp"
```

```
Enter the numbers:
```

```
a = 44444 ↵
```

```
b = 10000000 ↵
```

```
Result is too large
```

```
***   # GCs: 1          GC time: 0.00 s      # collected cells: 0      ***
***   # blocks: 1       block size: 16383     # free cells:      16376   ***
***   total CPU time: 0.03 s                  ***
```

```
<
```

The result exceeds the maximum representable number in C, $2^{31} - 1$. Enter **Q** to end the SIMATH session.

```
< Q ↵
```

```
SM terminated at 12:06:47
```

```
%
```


3. Description of SIMATH commands

3.1. Overview. The SIMATH commands are listed in a five-line menu. Execution of those commands necessitates only typing the symbol in parenthesis followed by the RETURN key (except for **\$**). All input is done in *buffered* mode, i.e. typing errors can be corrected by using the BACKSPACE and/or DELETE key.

Whenever a SIMATH command requires the input of a file (or library) name, the last name given as input for that SIMATH command will be shown on your screen as the current file (or library) name (see §2.3, **mainp** function example). You must either hit the RETURN key to keep the current name or enter a new name (in which case the current name will be overwritten by the new name).

Note: At the beginning of a SIMATH session, the names of the source library, object library, and program defined by the last **A**, **a**, and **N** commands of the previous session will automatically be read in from the file **.SM.default**; they are shown on your screen in the introductory lines. If the file **.SM.default** does not exist or if there is no preinstallation, the following lines will appear on your screen

```
no source library
no object library
no file name
```

At the end of a SIMATH session, the names last preinstalled with the **A**, **a**, and **N** commands will be written into the file **.SM.default** in the directory from which SIMATH was started (if you have writing permission).

Next is a short description of the SIMATH commands. A detailed description of each command will be given in the following sections.

Command	Input	Description
NAME	N	enter program name
edit	e	start editor (see note below)
Edit	E	start editor (see note below)
pre	p	start preprocessor
obj	o	create object module
comp	c	compile and link executable program
Comp	C	compile and link executable program in the background
ar name	a	enter module library name
dir	d	list contents of the module library
ar+obj	+	archive object module into the module library
ar−obj	−	remove object module from the module library
ranlib	r	supply module library with table of contents
AR NAME	A	enter source library name
DIR	D	list contents of the source library
AR*SRC	*	archive source file into the source library
AR_SRC	−	remove source file from the source library
EXTRACT	X	extracts the sources of a SIMATH-function to the current working directory. The function-name must be defined by 'N'.
SELECT	S	copy source file from the usr-source library to the current working directory
!	!	branch into a subshell
\$	\$	execute a SHELL command
?	?	look up keywords
~, @	~, @	look up on-line documentation
.	.	look up short description
PRINT	P	print source file
HELP	H	display menu
RUN	R	run program
QUIT	Q	end SIMATH session

Note: **e** usually calls vi and **E** emacs, but this depends on your installation (see installation instructions).

3.2. Menu-line 1: Editing and compiling programs. Note: “similar shell commands” refer to standard UNIX commands; they may differ from your machine’s shell commands.

(N)AME:	(e,E)dit	(p)re	(o)bj	(c)omp	(C)omp
---------	----------	-------	-------	--------	--------

(N)AME: Enter program name

Key < **N** ↵
 file name: **pname** ↵
 <

or < **N** ↵
 current file name: pnameold
 new file name: **pname** ↵
 <

Function **pname** is the new preinstalled program, i.e. the new current program to be worked on. If only the RETURN key is hit, the current file **pnameold** remains valid.

Remark Suffixes are used internally to differentiate between the SM source file **pname.S**, the SM preprocessor versions **pname.P** and **pname.c**, the C preprocessor version **pname.i**, the object module **pname.o**, the list of compiler errors **pname_CC** (when compiling in the background), and the executable program **pname.x**.

(N)AME:	(e,E)dit	(p)re	(o)bj	(c)omp	(C)omp
---------	----------	-------	-------	--------	--------

(e)dit Start editor

Key < **e** ↵

Function The preinstalled file is edited using the editor corresponding to **e**, usually vi (see note §2.4.1).

similar

shell command % **vi pname.S** ↵

(N)AME:	(e,E)dit	(p)re	(o)bj	(c)omp	(C)omp
---------	----------	-------	-------	--------	--------

(E)dit Start editor

Key < **E** ↵

Function The preinstalled file is edited using the editor corresponding to **E**, usually emacs (see note §2.4.1).

similar

shell command % **emacs pname.S** ↵

(N)AME:	(e,E)dit	(p)re	(o)bj	(c)omp	(C)omp
---------	----------	-------	-------	--------	--------

(p)re Start preprocessor

Key < **p** ↵

Function The SIMATH preprocessors and C preprocessor are run with the preinstalled file **pname.S** as input. The SIMATH preprocessor creates the temporary files **pname.P** and **pname.c**. **pname.c**, the C program equivalent of **pname.S**, is then passed to the C preprocessor which creates the file **pname.i**.

similar

shell command % **cc -P -I. -I/usr/local/simath/include pname.c** ↵

Remark (**p**)re is meant to be used mainly as a debugging tool.

(N)AME:	(e,E)dit	(p)re	(o)bj	(c)omp	(C)omp
---------	----------	-------	-------	--------	--------

(o)bj Create object module

Key < **o** ↵

Function The C compiler compiles the file **pname.c** to the object module **pname.o** after the SM preprocessors have been run automatically. Any compiling error will appear on your screen; use the **e** or **E** command to correct the error(s) in **pname.S**.

similar

shell command % **cc -O -I. -I/usr/local/simath/include -c pname.c** ↵

(N)AME: (e,E)dit (p)re (o)bj (c)omp (C)omp	
(c)omp	Compile and link an executable program
Key	< c ↵
Function	The C compiler creates an executable program pname.x from pname.S where pname.S must contain exactly one main function. The SM preprocessors are run automatically. Required modules from C and/or SIMATH libraries are linked automatically; any preinstalled, user-defined module library (see command (a)r name) has priority over the system libraries. Any compiling or linking error will be shown on your screen; use the e or E command to correct them.
similar shell command	% cc -O -I. -I/usr/local/simath/include -L/usr/local/simath/lib [-lexample] -lkern -lec4 ... -llist -lm -ltermcap -s -o pname.x pname.c ↵
Remark	see remark for (C)omp

(N)AME:	(e,E)dit	(p)re	(o)bj	(c)omp	(C)omp
(C)omp	Compile in the background				
Key	<code>< C ↵</code> Confirm with RETURN or enter additional module libraries ar1 ar2 ... ; ar1 will be linked first followed by ar2 <code>: [ar1 ...] ↵</code>				
Function	This is the same function as (c)omp except that everything runs in the background. Optional, additional module libraries ar1 <code>...</code> have priority over system libraries during linking (ar* stands for <code>/usr/local/simath/lib/libar*.a</code>); uppermost priority is still given to the preinstalled, user-defined module library (see command (a)r name). Compiling or linking errors will be listed in the file pname_CC ; use the e or E command to correct them.				
Remark	In order to save space, the c and C commands automatically execute a strip command (i.e the linker ld is run with the <code>-s</code> option) which removes the symbol table and relocation bits ordinarily attached to the output of the assembler and linker. If you want to use a debugger, use the corresponding shell commands <code>c: % CCC pname [ar1 ar2 ...] ↵</code> <code>C: % CCC pname [ar1 ar2 ...] & ↵</code> where ar* stands for <code>/usr/local/simath/lib/libar*.a</code> . The libraries ar1 , ar2 ... will take precedence over the system libraries during linking.				

3.3. Menu-line 2: Archiving object modules.

object:		(a)r name	(d)ir	ar(+)obj	ar(-)obj	(r)anlib
(a)r name	Enter module library name					
Key	<p>< a ↵</p> <p>object library: /usr/local/simath/lib/lib*.a,</p> <p style="padding-left: 150px;">* = lname ↵</p> <p><</p>					
or	<p>< a ↵</p> <p>current object library: /usr/local/simath/lib/liblnameold.a</p> <p>new object library: /usr/local/simath/lib/lib*.a ,</p> <p style="padding-left: 150px;">* = lname ↵</p> <p><</p>					
Function	<p>liblname.a is the new preinstalled object module library, i.e. your own object module library to be used with highest priority by the linker; it is located in the directory /usr/local/simath/lib. If only the RETURN key is hit, the current object library liblnameold.a will remain valid.</p>					
Remark	<p>If you no longer need your preinstalled module library, type in . for lname to cancel the preinstallation.</p>					
object:		(a)r name	(d)ir	ar(+)obj	ar(-)obj	(r)anlib
(d)ir	List contents of the module library					
Key	< d ↵					
Function	<p>The files contained in the preinstalled object module library are listed; additional information such as file attributes, file size, and date of creation are also given.</p>					
similar						
shell command	% ar tv /usr/local/simath/lib/liblname.a page -d ↵					
Remark	<p>If there is no preinstalled library or if the library does not exist yet, an error message will appear on your screen.</p>					

object:	(a)r name	(d)ir	ar(+)obj	ar(-)obj	(r)anlib
ar(+)obj	Archive object module				
Key	< + ↵				
Function	<p>The object module pname.o, created from the preinstalled program pname.S by the command o, is added to the preinstalled object module library and deleted from the current working directory.</p> <p>If the preinstalled library does not exist, it will be created.</p> <p>If pname.o already exists in the library, the user will be asked whether or not to replace the pname.o already in the library by the pname.o from the current working directory.</p> <p>If pname.o does not exist, an error message will appear on your screen.</p>				
similar shell commands	<pre>% ar r /usr/local/simath/lib/liblname.a pname.o ↵ % rm pname.o ↵</pre>				
Remarks	<p>If no library or program name is preinstalled or if an error occurs during archiving, an error message will be displayed on your screen and the module pname.o will not be deleted from the current working directory.</p> <p>After successful archiving, the following message will be displayed on your screen:</p> <pre>'ar r /usr/local/simath/lib/liblname.a pname.o' done</pre>				

object:	(a)r name	(d)ir	ar(+)obj	ar(-)obj	(r)anlib
ar(-)obj	Remove object module from the library				
Key	< - ↵				
Function	<p>The object module pname.o, created from the preinstalled program pname.S by the command o, will be removed from the preinstalled object module library.</p> <p>If pname.o is not in the library, an error message will appear on your screen.</p>				
similar					
shell command	% ar d /usr/local/simath/lib/liblname.a pname.o ↵				
Remarks	<p>If no library or program name is preinstalled or if an error occurs, an error message will be displayed on your screen.</p> <p>After successful execution, the following message will be displayed on your screen:</p> <p>‘pname’ deleted from ‘/usr/local/simath/lib/liblname.a’</p>				

object:	(a)r name	(d)ir	ar(+)obj	ar(-)obj	(r)anlib
(r)anlib	Supply module library with table of contents				
Key	< r ↵				
Function	<p>The preinstalled object module library will be supplied with a table of contents __SYMDEF. On many workstations, ranlib is not necessary; the command r has no effect (see note p. 13).</p>				
similar					
shell command	% ranlib /usr/local/simath/lib/liblname.a ↵				
Remarks	<p>If ranlib is necessary on your machine, then you should run ranlib again after any change to the object library in order to keep the table of contents for archive up to date; otherwise, you will get a warning from the linker when you use the c or C command.</p> <p>If no library is preinstalled or if an error occurs, an error message will be displayed on your screen.</p>				

3.4. Menu-line 3: Archiving source files. Lines 2 and 3 in the menu contain similar commands referring to object module libraries and source libraries respectively; the only difference is that you also have to press the Shift key for the commands in line 3.

These SIMATH commands are meant to help you organize your program source files into some meaningful order. You do not have to use them.

SOURCE:	(A)R NAME	(D)IR	AR(*)SRC	AR(_)SRC	(S)ELECT
---------	-----------	-------	----------	----------	----------

(A)R NAME Enter source library name

Key < **A** ↵
 source library: **USR-SNAME** ↵
 <

or < **A** ↵
 current source library: USR-SNAMEOLD
 new source library: **USR-SNAME** ↵

Function **USR-SNAME** is the new preinstalled source library, i.e. a library where you may archive your program source files; it is located in the working directory. If only the RETURN key is hit, the current source library **USR-SNAMEOLD** will remain valid.

SOURCE:	(A)R NAME	(D)IR	AR(*)SRC	AR(_)SRC	(S)ELECT
(D)IR	List contents of the source library				
Key	< D ↵				
Function	The files contained in the preinstalled source library are listed; additional information such as file attributes, file size, and date of creation are also given.				
similar					
shell command	% ar tv USR-SNAME page -d ↵				
Remark	If there is no preinstalled library or if the library does not exist yet, an error message will appear on your screen.				

SOURCE:	(A)R NAME	(D)IR	AR(*)SRC	AR(_)SRC	(S)ELECT
AR(*)SRC	Archive source file				
Key	< * ↵				
Function	<p>The preinstalled file pname.S is added to the preinstalled source library and deleted from the current working directory.</p> <p>If the preinstalled library does not exist, it will be created.</p> <p>If pname.S already exists in the library, the user will be asked whether or not to replace the pname.S already in the library by the pname.S from the current working directory.</p> <p>If pname.S does not exist, an error message will appear on your screen.</p>				
similar	% ar r USR-SNAME pname.S ↵				
shell commands	% rm pname.S ↵				
Remarks	<p>If no library or program name is preinstalled or if an error occurs during archiving, an error message will be displayed on your screen and the file pname.S will not be deleted from the current working directory.</p> <p>After successful archiving, the following message will be displayed on your screen:</p> <p>'ar r USR-SNAME pname.S' done</p>				

SOURCE:	(A)R NAME	(D)IR	AR(*)SRC	AR(_)SRC	(S)ELECT
---------	-----------	-------	----------	----------	----------

AR(_)SRC Remove source file from the library

Key < - ↵

Function The preinstalled source file **pname.S** will be removed from the preinstalled source library.
If **pname.S** is not in the library, an error message will appear on your screen.

similar

shell command % ar d USR-SNAME pname.S ↵

Remarks If no library or program name is preinstalled or if an error occurs, an error message will be displayed on your screen.
After successful execution, the following message will be displayed on your screen:
'pname' deleted from 'USR-SNAME'

SOURCE:	(A)R NAME	(D)IR	AR(*)SRC	AR(_)SRC	(S)ELECT
---------	-----------	-------	----------	----------	----------

SELECT Copy source file from the source library to the current directory

Key < S ↵

Function The preinstalled file **pname.S** will be copied from the preinstalled source library to the current working directory; **pname.S** is not deleted from the library.
If **pname.S** already exists in the current directory, the user will be asked whether or not to replace the **pname.S** already in the current directory by the **pname.S** from the source library.
If **pname.S** is not in the library, an error message will appear on your screen.

similar

shell command % ar x USR-SNAME pname.S ↵

Remarks If no library or program name is preinstalled or if an error occurs, an error message will be displayed on your screen.
After successful execution, the following message will be displayed on your screen:
'pname.S' extracted from 'USR-SNAME'

3.5. Menu-line 4: SIMATH-sources commands.

SM-SRC:	E(X)TRACT	(~,@,.)
---------	-----------	---------

E(X)TRACT Extracts a SIMATH-function.

Key < **X**

Function If you have defined a Name of a SIMATH-function with the command 'N', X will try to find this function in the SIMATH-source directory and copies it into the current working directory.

SM-SRC:	E(X)TRACT	(~,@,.)
---------	-----------	---------

(~,@,.) Look up on-line documentation

Key < ~ ↵ or < @ ↵
documentation of: **xyz** ↵

< . ↵
short description of: **xyz** ↵

Function In the first two cases a detailed description of the SIMATH function **xyz** is displayed. In the third case, the description is displayed in TeX format, if the SIMATH function does support a TeX documentation. If the description is longer than one page, type **SPACE** to continue or **q** to quit.
If you use the command . , the short description of the SIMATH function **xyz** is displayed, i.e. the function name with parameters and the detailed function name.
Note: the search is case sensitive, e.g **Xyz** would not be found.

Remark The tilde ~ serves as an escape character when a remote login, using **rlogin**, is performed. In this case, SM might not accept ~; you should use @ instead.

3.6. Menu-line 5: General commands.

(!)	(\$)	(?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT
-----	------	-----	---------	--------	-------	--------

(!) Branch into a subshell

Key < ! ↵

Function Branch into a subshell, i.e. the shell program specified in your environment variable SHELL is invoked. You can work in the subshell without losing any information preinstalled in SM. To leave the subshell, type 'exit' or 'CTRL/D'.

(!)	(\$)	(?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT
-----	------	-----	---------	--------	-------	--------

(\$)

Execute a SHELL command

Key < \$ **command** ↵

Function The shell command **command** is executed; no need to branch into a subshell using !.

Remark The **cd** command does not change your working directory.

(!)	(\$)	(?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT
-----	------	-----	---------	--------	-------	--------

(?) Look up keywords

Key < ? ↵

Function With the SM keyword index, you can search for SIMATH functions which match (or match not) your keywords. After entering the SM keyword index, type ? for help. A key is an *egrep regular expression*. You can add a key to the list (+), remove a key (-), change a key (**c/C**), or delete the entire list (**0**). The number of keys is limited by 9. The list is printed by the command **l/L**. With the commands **p/P**, **@/~**, **s/S** you can print the matching SIMATH functions, print the documentation of a function or print the source code of a function, respectively. You can quit the keyword index to SM with **q/Q**.

(!)	(\$)	(?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT
-----	------	-----	---------	--------	-------	--------

(P)RINT Print source file

Key < **P** ↵

Function **/usr/local/simath/bin/dr** prints the file **pname.S**.
(By default, **/usr/local/simath/bin/dr** contains two lines

```
#!/bin/sh
lpr $ *
```

You should change it according to your needs.)

similar

shell command % **/usr/local/simath/bin/dr pname.S** ↵

(!)	(\$)	(?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT
-----	------	-----	---------	--------	-------	--------

(H)ELP Display menu

Key < **H** ↵

Function The menu of the SIMATH commands is displayed.

(!)	(\$)	(?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT
-----	------	-----	---------	--------	-------	--------

(R)UN Run program

Key < **R** ↵

Function The executable program **pname.x**, created from the preinstalled
program **pname.S** by the command **c** or **C** will be executed.

similar

shell command % **pname.x** ↵

(!)	(\$)	(?)	(P)RINT	(H)ELP	(R)UN	(Q)UIT
-----	------	-----	---------	--------	-------	--------

(Q)UIT End SIMATH session

Key < **Q** ↵

Function SM is terminated.

Note: In SM, you can enter several commands on one line, e.g.

< **o + r** ↵

instead of

< **o** ↵

< **+** ↵

< **r** ↵

Trial and error will tell you what can be entered on a single line !

CHAPTER 3

Programming in SIMATH

1. The extended programming language C

The programs of the SIMATH user and the internal functions of the SIMATH system are, essentially, C programs. C was extended by adding

- **include files** for the preprocessor (§3.2),
- **data types** (§3.3),
- **format arguments** to *printf* and *fprintf* for the new data types (see on-line documentation of *printf* and *fprintf*, e.g. with `@(f)printf`),
- **functions**, collected in libraries (§3.6 and ch. 4),
- **global parameters** (§3.7). x

There are some changes from standard C programs in the declaration and instruction parts of a function due to the new data types (§3.3).

2. Include files in SIMATH

In order to develop SM programs, you must have the preprocessor instruction

include <_simath.h>

before using SIMATH data types and SIMATH functions. **_simath.h** makes all of the include files of the SIMATH system and the include files of the C libraries which are required by SIMATH available.

The include files of the SIMATH system contain

- *define* instructions for the implementation of *global constants*,
- *macros* for the different program packages,
- *type definitions*,
- declaration of *global parameters*.

The following C libraries are automatically included into any SM program via **_simath.h**:

<stdio.h>
<setjmp.h>
<ctype.h>
<math.h>
<sys/types.h>
<sys/times.h>
<sys/param.h>
<sys/stat.h> .

3. Data types in SIMATH

3.1. C data types. The data types of C remain valid in SIMATH.

Exception: Instead of **int**, integers with bounded values have the type designation **single** (*single precision integer*).

Variables **x** of type **single** are bounded by

$$|x| < 2^{30}.$$

Variables of type **single** are subject to the same conditions, rules, and applications as variables of type **int** in C. For reasons of program readability, the type modifiers **short** and **long** should be avoided.

3.2. Additional data types in SIMATH. Here is a short description of each new data type and in which part of SIMATH it is defined.

Type	Description	Package
CELL ¹	cell: structure (see §4.1.3)	Base
PCELL ¹	pointer to a cell	Base
atom	integer x with $ x < 2^{30}$	Base
obj	object: atom or list	Base
list	pointer to a list of objects	Base
single	single precision integer (x with $ x < 2^{30}$)	Arithmetic
int	integer of arbitrary size	Arithmetic
rat	rational number	Arithmetic
floating	real number	Arithmetic
complex	complex number	Arithmetic
gfel	element of a Galois field	Arithmetic
gf2el	element of a Galois field of characteristic 2	Arithmetic
nfel	element of a number field	Arithmetic
pfel	p -adic number	Arithmetic
rfunc	rational function	Arithmetic
afunc	algebraic function	Arithmetic
pol	polynomial	Polynomial
matrix	matrix	Matrix-vector
vec	vector	Matrix-vector

Each new data type will be described in detail in chapter 4.

¹used only by the system

3.3. Definitions using SIMATH data types. These are the rules pertaining to variables of the type described in §3.3.2 (except for **atom**); we will refer to those variables as “SIMATH-type” variables. Ignoring these rules may lead to inconsistencies in the SIMATH memory administration. The rules apply analogously to any new type definitions introduced by **typedef**.

1. The C instructions **longjmp()**, **goto**, **break**, **continue**, and **setjmp()** should not be used within the scope of SIMATH-type variables.
2. SIMATH-type variables cannot be declared with the **static** storage class specifier. Exception: see the on-line documentation for **globinit()**.
3. SIMATH-type variables cannot be declared as an element of a **union** definition.
4. SIMATH-type variables (of the same type) cannot be initialized simultaneously (as is often done in C programs).

3.4. The **bind() and **init()** instructions.** The **bind()** and **init()** instructions are used between the declaration and instruction parts of a main function or block to create a reference in the SIMATH stack for each SIMATH-type variable; variables which have no reference in the stack will be returned by the garbage collector to the list of available cells (thus their contents will be destroyed). These are *independent* instructions, i.e. they must be terminated by a semicolon and a line feed is not allowed within the instruction. **init()** differs from **bind()** in that it also initializes each parameter to its corresponding nil value.

For any SIMATH function having SIMATH-type variables x_1, \dots, x_m as parameters and SIMATH-type variables y_1, \dots, y_n in the declaration part, we must use the instructions

$$\begin{aligned} &\mathbf{bind}(x_1, \dots, x_m); \\ &\mathbf{init}(y_1, \dots, y_n); \end{aligned}$$

before the instruction part of the function.

Note that we do **not** use **init**(x_1, \dots, x_m) since arguments used to call a function usually already have a value assigned to them!

For a main function, all SIMATH-type global variables must be “initialized” by **init()** or **globinit()** before the instruction part.

For structures, e.g.

```
typedef struct {
    char NAME;
    list L;
} nlist;
```

```
nlist S;
nlist *p = &S;
```

```
struct T {
    list A;
    nlist NL;
}
```

init() and **bind()** must be used for each component which is of SIMATH-type; e.g.

```
init(T.A, p->L, T.NL.L);
```

For SIMATH-type arrays, e.g.

```
pol PV[10];
int IV[3][100];
```

init() and **bind()** must be used for each component; e.g.

```
init(PV[0..9], IV[0..2][0..99]);
```

The values in the square brackets must be constants or variables; e.g.

```
up(n)
single n;
{
    int VEC[1000];
    init(VEC[0..n]);
    :
}
```

In order not to overload the reference list of the SIMATH memory administration, in some specific cases, **init()** and **bind()** can be called with single components of structures or partial areas of arrays:

- if you are sure that **only** those components or array areas will be used throughout the current computation,
- if you know that the other components or array areas have already been “bound” or “initialized” in higher calling functions,
- for arrays of type **int**, if only **single** values will occur in that array area throughout the current computation,

i.e. statements such as

```
init(PV[2], IV[2][20]);
```

and

```
init(IV[0..2][0..19], IV[0][20..99]);
```

are possible.

When using **bind()** and **init()** on single components of an array, pointer notation is allowed; e.g. one can write

```
init(*(PV+2), (*(IV+2))[20]);
```

for

```
init(PV[2], IV[2][20]);
```

Finally, for variables defined as a mixture of arrays and structure components, e.g.

```
typedef struct {
    list paar;
    list s;
} feld[10];

feld str1;

struct name {
    feld strfeld;
    char lname[20];
}
```

bind() and **init()** must be called as follows:

```
bind(str1[1..n].paar);
init(name.strfeld[0].s);
```

“Binding” and “initializing” whole array areas can be done only for the first component of a structure, i.e.

```
init(name.strfeld[1..n].s);
```

is not allowed.

Note: **bind()** and **init()** are interpreted by the SIMATH-preprocessor. To avoid the SIMATH-preprocessor you can use a combination of **Sbind()-Sfree()** or **Sinit()-Sfree()**. See the documentations to these functions.

3.5. Computations with SIMATH data types. All operations which refer to SIMATH data types are implemented using SIMATH functions, i.e. C operators must be replaced by the corresponding SIMATH functions. For example, the C instruction

$$\mathbf{c} = \mathbf{a} * \mathbf{b};$$

for integers **a**, **b**, and **c** must be replaced by

$$\mathbf{c} = \mathbf{iprod}(\mathbf{a}, \mathbf{b});$$

in a SIMATH program.

Note: In order to avoid strange behavior in your SIMATH programs, always be wary of nested function calls which return SIMATH-type results; see §3.5.

4. Example

We now give an example of a SIMATH program; we will modify the functions **gcd()**, **lcm()** and **mainp()** from §2.2. A detailed description of the modifications will be given in chapter 4.

Now change the file **gcd.S** as follows:

```
#include <_simath.h>

int          gcd(a, b)
    int      a, b;
{
    /* declaration part */
    int      c;

    /* binding and initialization */
    bind(a, b);
    init(c);

    /* instruction part */

    if (icomp(a, b) < 0)
        { c = a; a = b; b = c; }

    while (b != 0)
        { c = irem(a, b); a = b; b = c; }

    return (a);
}
```

Some remarks about the modifications: as mentioned in §3.2, we included the SIMATH system file **<_simath.h>**. We added the **bind()** and **init()** functions (as seen in §3.3.4) and the **icomp()** and **irem()** functions from the arithmetic package (described in §4.2.4). Note that equality comparison with zero can be carried out with the usual C operators, i.e. **==** and **!=**, independently of the data type (see §4.2).

Now let us modify the file **lcm.S**.

```
#include <_simath.h>

int          lcm(a, b)
    int      a, b;
{
    /* declaration part */
    int      c;

    /* binding and initialization */
    bind(a, b);
    init(c);

    /* result zero */
    if (a == 0 || b == 0)
        c = 0;
    else
    {
        /* general case */
        a = iabs(a);
        b = iabs(b);

        /* c = (a / gcd(a,b)) * b */
        c = iprod(iquot(a, gcd(a, b)), b);
    }
    return (c);
}
```

Some more remarks: here we added the SIMATH functions **iabs()**, **iprod()**, and **iquot()** all from the arithmetic package. Note that this time we do not need to return a replacement value for non-representable results.

Finally change **mainp.S**.

```
#include <_simath.h>

main()
{
    /* declaration part */
    int          a, b;

    /* binding and initialization */
    init(a, b);

    /* read the numbers */
    printf("\nEnter the numbers: \na = ");
    while ((a = geti()) == ERROR)
        printf("Reading error \na = ");
    printf("b = ");
    while ((b = geti()) == ERROR)
        printf("Reading error \nb = ");

    /* lcm */
    printf("lcm(%i, %i) = %i \n", a, b, lcm(a, b));
}
```

Last remarks: since we are using the SIMATH function **geti()** to read in the numbers, we do not need to change the symbol order to numbers as in §2.2. Note the use of the argument **%i** in the **printf()** command instead of **%d**.

Do not forget to compile and archive the modified **gcd()** and **lcm()** functions, and to compile and link the modified **mainp()** function as described in §2.2.

Now carry out the computation in the example session of §2.2 which printed an error message.

```
< R ↵
RUN "mainp"

Enter the numbers:
a = 44444 ↵
b = 10000000 ↵
lcm(44444, 10000000) = 111110000000

*** # GCs: 1      GC time: 0.00 s    # collected cells: 0      ***
*** # blocks: 1   block size: 16383  # free cells:      16369 ***
*** total CPU time: 0.01 s          ***

<
```

As you can see, computing with large numbers is no problem in SIMATH. At the end of the program, memory administration information and computation time are given. In

the above example, most of the time is used by the initialization of the SIMATH memory administration; only a small fraction of the time is used for the actual computation.

In the directory `/usr/local/simath/examples/basics/` you can find further examples of simple SIMATH-programs.

5. Nested function calls

In general, nested function calls are permitted in SIMATH. For example, for integers **M**, **N**, and **P**, the assignment statement

$$\mathbf{P} = (\mathbf{M}+7) \cdot \mathbf{N}$$

is replaced by

$$\mathbf{P} = \mathbf{iprod}(\mathbf{isum}(\mathbf{M},7), \mathbf{N}).$$

Nested function calls will not cause any problems if at most one of the parameters is a function or a macro.

In the current SIMATH version, the nesting of functions which return SIMATH-type results (except for **atom**) might cause inconsistencies in the dynamic memory administration. This problem occurs only if the automatic garbage collector **gc()** is started while the return value of a function or a macro is evaluated. An enlarged list store and a explicitly started **gc()** may prevent this. This limitation shall be removed in a later SIMATH version.

In general, macros should never be called with a function or a macro as a parameter. If you want to nest macros, you should check how the macros are expanded by the C compiler; use the **(p)re** command (see §2.4.2) to see how the C preprocessor expands them.

6. Nomenclature of SIMATH functions

In every SIMATH function, you will find a title containing a few keywords which briefly describe what the function does, e.g.

**integer product,
polynomial over modular singles resultant.**

Suitable abbreviations of these titles have been chosen as function names, e.g. for the examples above

**iprod(),
pmsres().**

Because of our strict nomenclature rules, you will know the most important function names after using the system for only a short time.

As examples of those rules, here is a short list of function names and their application area.

Package	Function name	Function application
Base	l ...	list
	o ...	object
	s ...	set
	us ...	unordered set
Arithmetic	cset ...	characteristic set
	s ...	single precision
	i ...	integer
	s ...	single-precision integer
	ms ...	modular single
	mi ...	modular integer
	r ...	rational
	fl ...	floating point number
	c ...	complex number
	nf ...	number field
	qnf ...	quadratic number field
	gfs ...	Galois field of single characteristic
	gf2 ...	Galois field of characteristic 2
	pf ...	p -adic field
	qff ...	quadratic function field
	rfmsp ...	rational function over modular single primes
	rfr ...	rational function over the rationals
	afmsp ...	algebraic function over modular single primes

Package	Function name	Function application
Elliptic curves	ec ...	elliptic curve (general)
	eci with integer coefficients
	ecr over the rationals
	ecnf over number field
	ecqnf over quadratic number field
	ecmp over modular primes
	ecgf2 over Galois field of characteristic 2
	ec*ac , actual model
	ec*snf , short normal form
	ec*min , minimal model
Polynomial	p ...	polynomial (general)
	pi over integers
	pms over modular single
	pmi over modular integer
	pr over the rationals
	pfl over floating point numbers
	pc over complex numbers
	pnf over number field
	pgfs over Galois field of single characteristic
	pgf2 over Galois field of characteristic 2
	ppf over p-adic field
	prfmisp over rational functions over modular single primes
	dip ...	distributive polynomial
	dp ...	dense polynomial
	up ...	univariate polynomial
	udp ...	univariate dense polynomial
Matrix	ma ...	matrix (general)
	mai with integer entries
	mams with modular single entries
	mami with modular integer entries
	mam2 with entries of $\mathbb{Z}/2\mathbb{Z}$ in special bit repre- sentation
	mar with rational entries
	manf with number field entries
	magfs with Galois field of single characteristic entries
	magf2 with Galois field of characteristic 2 entries

Package	Function name	Function application
	maup ...	matrix of univariate polynomials
	map / mp ...	matrix of polynomials (general)
	mapi over integers
	mapms over modular single
	mapmi over modular integer
	mapr over the rationals
	mapnf over number field
	mapgfs / mpgfs over Galois field of single characteristic
	mapgf2 / mpgf2 over Galois field of characteristic 2
	marfr of rational functions over the rationals
	marfmsp of rational functions over modular single primes
Vector	vec ...	vector (general)
	veci with integer entries
	vecms with modular integer entries
	vecmi with modular single entries
	vecr with rational entries
	vecnf with number field entries
	vecgfs with Galois field of single characteristic entries
	vecgf2 with Galois field of characteristic 2 entries
	vecup ...	vector of univariate polynomials
	vecp / vp ...	vector of polynomials (general)
	vecpi over integers
	vecpms / vpms over modular single
	vecpmi / vpmi over modular integer
	vecpr over the rationals
	vecpnf / vpnf over number field
	vecpgfs / vpgfs over Galois field of single characteristic
	vecpgf2 / vpgf2 over Galois field of characteristic 2
	vecrfr of rational functions over the rationals
	vecrfmsp of rational functions over modular single primes

Input and output functions are adapted to the nomenclature in C.

```

get ...
put ...
fget ...
fput ...

```

Functions which serve to test certain conditions of the data (“is single precision?”, “is integer?”) have the form

is....

7. Global parameters in SIMATH

Here is a list of all the parameters used in SIMATH; parameters required for programming in SIMATH and not mentioned here will be considered in detail in chapter 4. Some parameters have a default value (d), others an initial value (i) which is changed automatically by the system during program execution.

The abbreviations B, A, E, P, and M (in the second column of the table) indicate in which package – base, arithmetic, elliptic curves, polynomial, or matrix/vector – the parameters find their application.

The third column (*owner*) indicates which parameters can be changed either automatically by the system or by the user; constants are identified as such and cannot be changed.

NOTE: The default value of parameters “owned” by user(E) (E for Expert) should in NO CASE be changed by programmers with little or no programming experience.

The last column gives a short description of each parameter.

The system variable **BL_START[0]** points to the first of the memory blocks currently allocated for the program execution; we shall call it the current program “workspace”. Within the workspace, **AVAIL** points to the list of free memory cells.

Parameter, value (d)/(i)		owner	description
AVAIL = _0	(i)	B system	list of free memory cells;
BASIS = 2^{30}	(d)	B constant	base for list representation of large numbers;
BLOG10 = 9	(d)	A constant	internal use: \log_{10} BASIS
BLOG2 = 30	(d)	A constant	internal use: \log_2 BASIS
BL_NR = 0	(i)	B system	number of blocks currently allocated to the workspace;
BL_NR_MAX	(d)	B user(E)	maximum number of blocks which can be allocated to the workspace (the default value depends on your machine); can be changed by gcreinit();
BL_SIZE = $2^{14} - 1$	(d)	B user(E)	number of cells in a memory block; can be changed by gcreinit() (see on-line documentation);

Parameter, default value		owner	description
BL_START []		B system	vector containing the starting address of each block allocated to the workspace;
BSMALL = 2^{15}	(d)	A constant	internal use: multiplication of large integers;
DECBAS = 10^9	(d)	A constant	decimal base for input and output of large numbers;
DIFF [481]	(d)	A constant	differences between the units $a_i \pmod{2310}$, for $1009 \leq a_i \leq 3319$;
DUM = 0	(i)	B system	internal dummy variable: used as a parameter in some macros;
ERROR = -2^{30}		B constant	return value of SIMATH functions in case of error;
FL_EPS = 5	(d)	A user	maximal list length of the mantissa of a floating point number;
FL_JMP	(i)	A system	internal variable for handling overflow of floating point numbers;
FL_LN2 = 0	(i)	A system	internal variable for floating point arithmetic;
FL_STZ = ST_INDEX	(i)	A system	internal variable for handling overflow of floating point numbers;
GC_CC = 0	(i)	B system	counter: number of cells collected by the garbage collector;
GC_COUNT = 0	(i)	B system	counter: number of garbage collections;
GC_MESS = 0	(d)	B user	messages from the garbage collector; suppress = 0 / print = 1;
GC_QUOTE = 10	(d)	B user(E)	If the percentage of free cells with respect to the size of the current workspace is less than $1/\mathbf{GC_QUOTE}$, the workspace is enlarged; if BL_NR = BL_NR_MAX , the garbage collector will terminate the program execution;

Parameter, default value			owner	description
GC_TEST = 0	(d)	B	user	the garbage collector will perform some additional tests on the program's lists; disable = 0 / enable = 1. This can be used as a debugging tool (see on-line documentation on islist());
GC_TIME = 0	(i)	B	system	total CPU time required for all garbage collections;
_H_BOUND = 0.0	(i)	E	user	if _H_BOUND > 0.0: upper limit for the Weil height of points on an elliptic curve over Q: used during the search for points in Manin's algorithm; if _H_BOUND = 0.0, the algorithm searches for points with unlimited Weil height; for most computations _H_BOUND = 11.0 is sufficient;
ITERMAX = 500	(d)	P	user	maximal number of iteration steps: used in computation of the real and complex roots of a polynomial (udprf.S);
LIST_GSP [51]	(d)	P	constant	list of the 50 largest single precision primes: used in polynomial factorization;
LIST_SP [169]	(d)	A	constant	list of all primes smaller than 1000: used in integer factorization;
LN_SIZE = 80	(d)	B	user	line length for output;
MADUMMY = 0	(i)	M	system	internal dummy variable: used as a parameter in some macros for matrix computations;
MANUMMY = 0	(i)	M	system	internal dummy variable: used as a parameter in some macros for matrix computations;
MARGIN = 0	(d)	B	user	left margin: column number where output should start;
NUM = 0	(i)	B	system	internal dummy variable: used as a parameter in some macros;
POLDUMMY = 0	(i)	P	system	internal dummy variable: used as a parameter in some macros for polynomial computations;

Parameter, default value		owner	description
RESTORE		B system	internal dummy variable for the preprocessor: used with the return value of functions;
SEMIRAND		A user	controls the behaviour of <code>irand()</code> ;
SP_MAX = BL_SIZE * BL_NR_MAX	(d)	B user(E)	maximum number of cells which can be allocated to the workspace of a running program; can be changed by <code>gcreinit()</code> ;
STACK		B system	pointer to the SIMATH stack of references;
ST_INDEX = 0	(i)	B system	stack pointer to the last occupied position in the SIMATH stack;
ST_SIZE = 500	(d)	B user	size of the SIMATH stack; can be changed via the setstack() instruction;
_0		B constant	empty list;

CHAPTER 4

The SIMATH system

1. The base system

1.1. Overview. The base system contains mainly routines for list programming, administration, searching, and sorting.

The workspace is implemented by “singly linked lists” which are administrated by the garbage collector **gc()**. It is dynamically created during program execution; the workspace is enlarged whenever there are not enough free memory cells available. Information on the size of the workspace is displayed at the end of program execution.

The global variable **SP_MAX** limits the maximal expansion of the workspace, but does not actually reduce nor enlarge it. If you want to change the size of the workspace, call the function **setspace()** with the desired number of memory cells (see the on-line documentation on **setspace()**). You can reinitialize the complete workspace by **gcreinit()**.

1.2. Data Types. The data type **list** represents an empty list or a list of finitely many objects. An object (data type **obj**) is either a list or an atom. An atom (data type **atom**) is an integer with absolute value smaller than **BASIS**. All other SIMATH data types are built from those three basic types. General advice on the “usage” of SIMATH data types can be found in §3.3.

1.3. Representing lists. ... in input and output

Lists are represented as an opening parenthesis followed by a sequence of objects, separated by a comma or blanks, and a closing parenthesis. For example, the following representations are valid:

```
(0, 8, -15)
()
(0 (10 11) 2 3)
```

(The symbol `()` represents the empty list.)

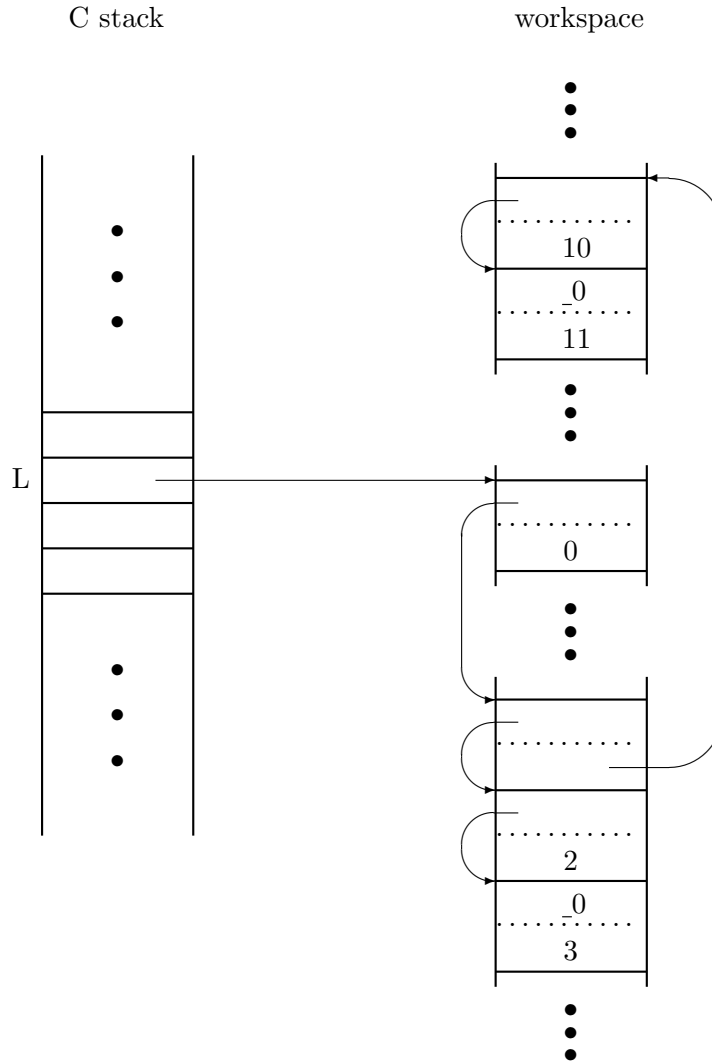
... in source programs

The functions `list1()`, `list2()`, ..., `list6()` create lists having 1, 2, ..., 6 objects respectively. For lists with more than six objects, use the function `lconc()` to concatenate two lists or `lcomp()` to add an element to a list (see §4.1.5). The empty list is represented by the SIMATH constant `_0`.

... in the memory

A variable of type **list** is implemented as a pointer to the first object in the list. Objects are implemented as cells from the workspace. A cell consists of two parts: a pointer to the next object in the list (reductum) and the actual data which is either an atom or a pointer to the first object in a sublist. The system internally differentiates between pointers and atoms.

Example. Let L be the list $(0 (10 11) 2 3)$. Pointers are represented by arrows. $_0$ marks the end of a list since the reductum, at this point, is the empty list.



1.4. The memory administration. At this point, we make some remarks to explain the role played by the global parameters in the internal memory administration. The SIMATH programmer can claim some control from the system over the memory administration by changing the value of certain global parameters (see §3.7). However, we strongly recommend to beginners to stay with the default values until they gain some programming experience.

The dynamic list administration is done automatically by the garbage collector (**gc()**); global parameters **GC_*** (see §3.7) refer to the garbage collector.

The workspace is divided into blocks of size **BL_SIZE**. The actual number of those blocks is given by **BL_NR**.

The workspace memory is enlarged if

1. the garbage collector cannot deliver the number of cells required by the running program. In this case, another block of cells is added to the workspace. At the beginning of a program execution, the workspace consists of no block, i.e. **BL_NR** = 0. If **BL_NR** exceeds the limit **BL_NR_MAX**, the program will exit and a message will be displayed on your screen.
2. the function **setspace()** is invoked; the required number of memory cells is passed as a parameter to **setspace()**. This can be done at any point within the program though **setspace()** is usually used at the beginning if you know that more than one block will be required during the computation. No more than **BL_NR_MAX** blocks will be used. (See the on-line documentation on **setspace()**.)

The default values of **BL_NR_MAX** and **BL_SIZE** depend on the size of your machine's main memory; **BL_NR_MAX** * **BL_SIZE** * **sizeof(CELL)** should not exceed that size. Use the global variable **SP_MAX** to limit the size of the workspace: at the beginning of the program, let **SP_MAX** be the maximum number of cells by which the workspace is allowed to expand. (Reasonable values are between **BL_SIZE** and **BL_NR_MAX** * **BL_SIZE**.) Since the allocation is done blockwise, the actual memory size may exceed **SP_MAX** by a small amount.

Although the functions **gc()**, **gcfree()**, **gccpr()**, **gcreinit()**, and **setspace()** allow handling of the memory management, the explicit call of these functions is not recommended in normal use.

1.5. The list functions. Here is a detailed description of the most important list functions. For each function, we also give an example and a short description of related functions.

“list of 1 object”

list list1(o)
obj o;

A list of one object is returned.

Example: **list L;**
 L = list1(7); **/* L = (7) */**
 L = list1(list1(2)); **/* L = ((2)) */**

Some related functions:

list2(), list3(), list4(), list5(), list6() for creating lists of respectively 2, 3, 4, 5, and 6 objects.

“list composition”

list lcomp(o, L)
obj o;
list L;

The list composition of the object **o** and the list **L** is returned.

Example: **list L, L1;**
 L = list2(2, 3);
 L1 = lcomp(7, L); **/* L1 = (7,2,3) */**

Some related functions:

lcomp2(), lcomp3(), lcomp4(), lcomp5(), lcomp6() for the composition of respectively 2, 3, 4, 5, and 6 objects and a list.
lsuffix() for the composition of a list and an object.

“list first”

obj lfirst(L)
list L;

The first object in the list **L** is returned.
(**L** must consist of at least one object!)

Example: **list L;**
 obj a;
 L = list3(2, 3, 5);
 a = lfirst(L); /* a = 2 */

Some related functions:

lsecond(), **lthird()**, **lfourth()**, **lfifth()**, **lsixth()** to get respectively the second, third, fourth, fifth, and sixth object in a list; the list must consist of at least as many objects as the position of the object to be retrieved.
lelt(), a general object retrieval function.

“list reductum”

list lred(L)
list L;

The reductum of the list **L** is returned, i.e. the pointer to the second object in **L**. Any subsequent change to the reductum (as **lfirst()**, **lconc()** etc.) will appear in **L**.
(**L** must consist of at least one object!)

Example: **list L, M;**
 L = list3(2, 7, 13);
 M = lred(L); /* M = (7,13) */

Some related functions:

lred2(), **lred3()**, **lred4()**, **lred5()**, **lred6()** to get respectively the second, third, fourth, fifth, and sixth reductum of a list; the list must consist of at least as many objects as the number of the reductum to be returned.
lreduct(), a general list reductum function.
llast() which returns a **list** made of the last object of the input list. **llast()** does not return the element as the functions **lfirst()**, **lsecond()** etc.

“list set first”

```
obj lsfirst(L, o)
list L;
obj o;
```

The first object in the list **L** is replaced by the object **o**; **o** is returned.
(**L** must consist of at least one object!)

```
Example:      list L;
               obj a;
               L = list4(2, 3, 5, 7);
               a = lsfirst(L, 1);      /* L = (1,3,5,7) */
                                           /* a = 1      */
```

Some related function:

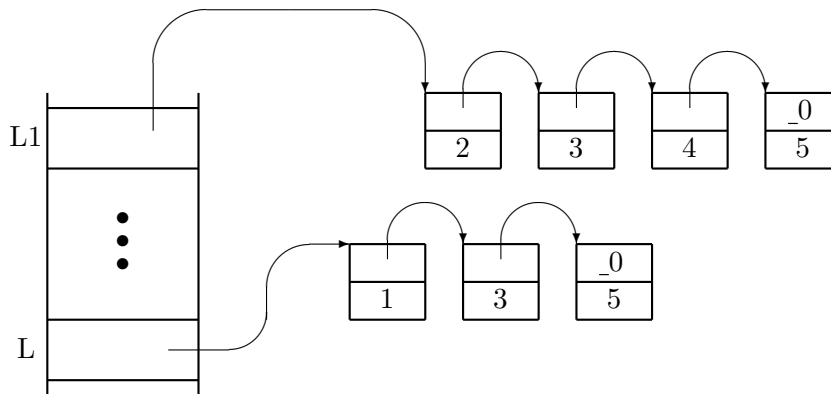
lset(), a general function to replace any element in a list.

“list set reductum”

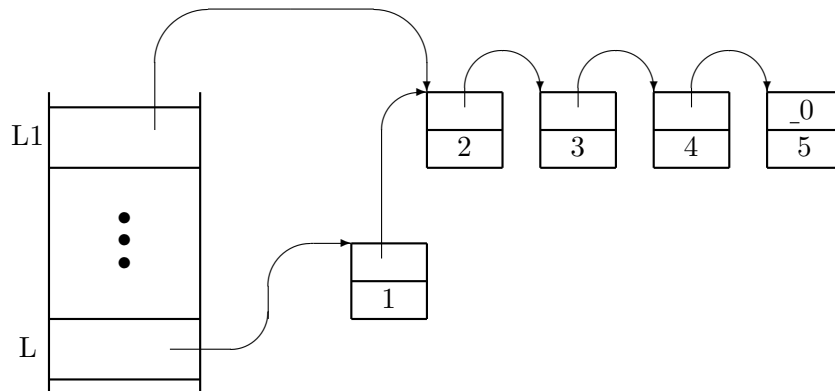
list lsred(L, L1)
list L, L1;

The reductum of the list **L** is replaced by the list **L1**; **L** is changed and **L1** is returned.
 (**L** must consist of at least one object!)

Example: **list L, L1, L2;**
 L = list3(1, 3, 5);
 L1 = list4(2, 3, 4, 5);



L2 = lsred(L, L1); **/* L = (1,2,3,4,5) */**
 /* L2 = (2,3,4,5) */

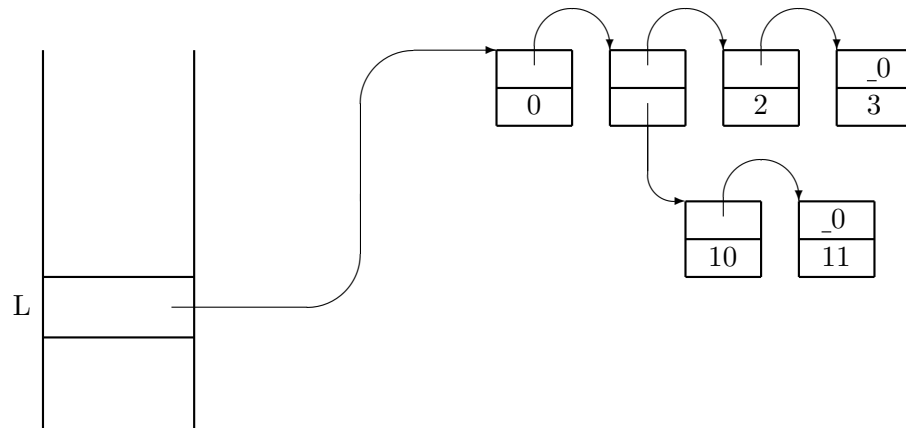


“list inverse”

list linv(L)
list L;

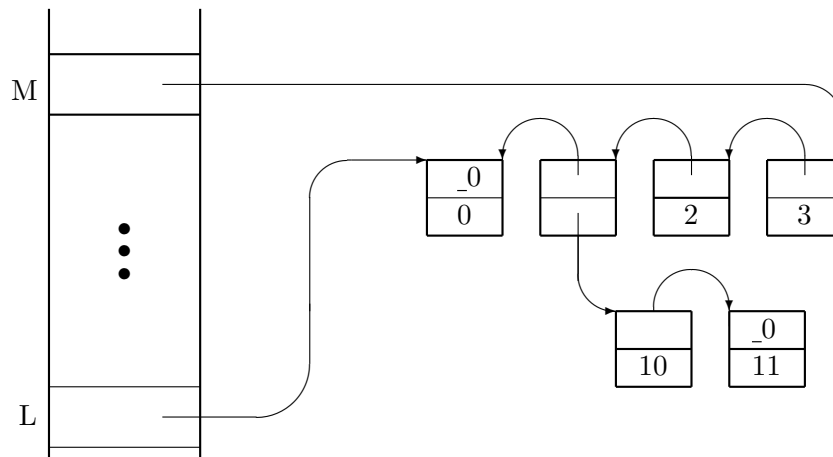
The inverse of the list **L** is returned; **L** will point to the last object of the inverted list.

Example: `list L, M;`
 `L = list4(0, (10, 11), 2, 3);`



`M = linv(L);`

`/* M = (3, 2, (10, 11), 0) */`
`/* L = (0) */`



Some related function:

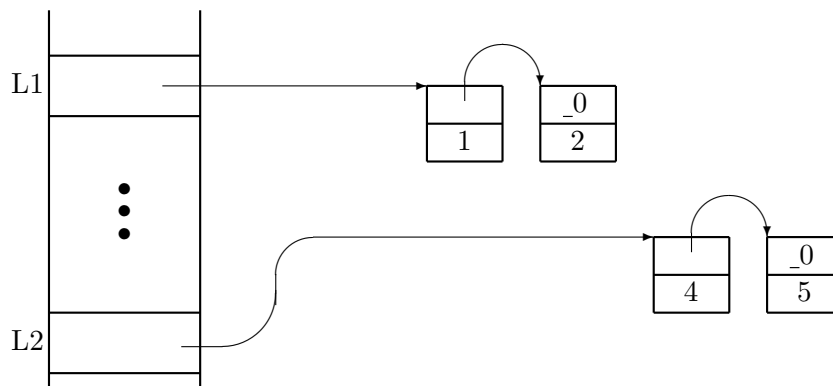
lcinv() which returns the inverse of a copy of the input list.

“list concatenation”

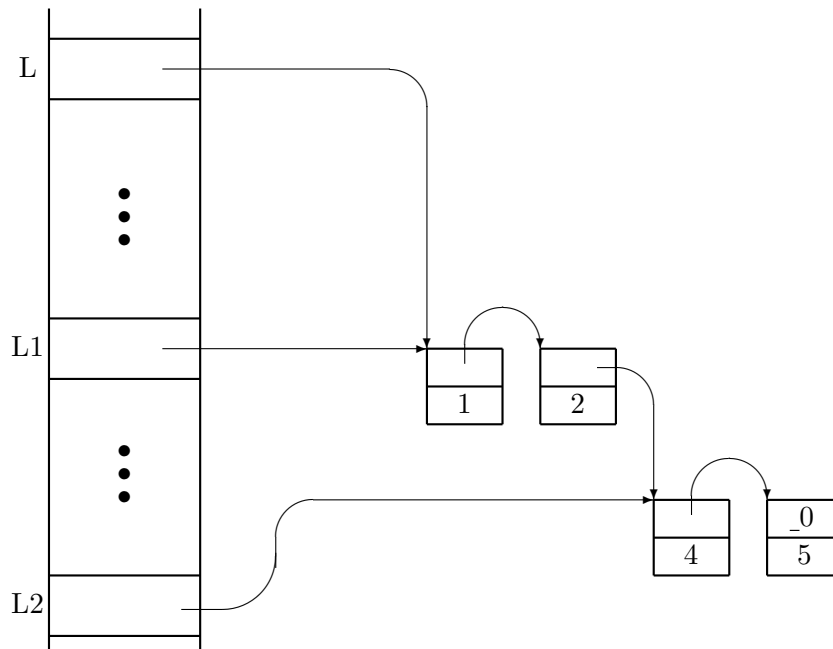
```
list lconc(L1, L2)  
list L1, L2;
```

The concatenation of the lists **L1** and **L2** is returned; **L1** is changed.

```
Example:      list L, L1, L2;  
              L1 = list2(1, 2);  
              L2 = list2(4, 5);
```



```
L = lconc(L1, L2);      /* L = (1, 2, 4, 5) */
```



Some related function:

llconc() which returns the concatenation of all the sublists in a list of lists.

“list constructive concatenation”

```
list lcconc(L1, L2)
list L1, L2;
```

The constructive concatenation of the lists **L1** and **L2** is returned, i.e. **L2** is appended to a copy of **L1** so that both **L1** and **L2** remain unchanged.

```
Example:      list L, L1, L2;
               L1 = list2(1, 2);
               L2 = list2(4, 5);
               L  = lcconc(L1, L2);      /* L  = (1,2,4,5) */
                                           /* L1 = (1,2)   */
                                           /* L2 = (4,5)   */
```

“list element delete”

```
obj ledel(*L, n)
list *L;
single n;
```

The **n**th element of the list **L** is deleted from **L** and returned.
(**L** must consist of at least **n** elements!)

```
Example:      list L;
               obj a;
               L = list4(2, 3, 5, 7);
               a = ledel(&L, 3);      /* L = (2,3,7) */
                                       /* a = 5      */
```

Some related function:

lecdel() to delete an element from a copy of the input list.

“list element insert”

```
list leins(L, k, o)
list L;
single k;
obj o;
```

The object **o** is inserted in the list **L** after the **k**th element; the changed list is returned.
(**L** must consist of at least **k** elements!)

```
Example:      list L, L1;
               obj a;
               L = list4(2, 3, 5, 7);
               a = 4;
               L1 = leins(L, 2, a);    /* L1 = (2,3,4,5,7) */
                                       /* L  = (2,3,4,5,7) */
```

Some related functions:

lecins() to insert an object in a copy of the input list.

leins2() to insert an object in the second position of a list.

lsins() to insert an object of type **single** at the appropriate position in an ordered list of elements of type **single**.

“list of single precision bubble merge sort”

```
list lsbmsort(L)
list L;
```

The ordered list of elements of type **single** is returned; **L** is changed. (This routine is to be used on large lists!)

```

Example:      list L, L1;
              L = list4(7, 3, 5, 2);
              L1 = lsbmsort(L);          /* L1 = (2,3,5,7) */

```

Some related function:

lsbsort() to sort small to medium lists of elements of type **single**.

“list search”

```

single lsrch(o, L)
obj o;
list L;

```

lsrch() returns the position number of the first encountered element of the list **L** which matches the object **o**; if **o** is not in the list, the function returns 0.
(**L** must consist of at least one object!)

```

Example:      list L;
              obj a;
              single n;
              L = list4(2, 3, 5, 7);
              a = 5;
              n = lsrch(a, L);          /* n = 3 */

```

Some related function:

lmemb() to test if an object belongs to a list or not.

“set union”

```
list sunion(L1, L2)  
list L1, L2;
```

The ordered set union of the lists **L1** and **L2** is returned; **L1** and **L2** must be ordered lists of elements of type **single**. (The input lists remain unchanged.)

```
Example:      list L, L1, L2;  
                L1 = list2(3, 4);  
                L2 = list3(2, 3, 5);  
                L  = sunion(L1, L2);      /* L = (2,3,4,5) */
```

Some related function:

usunion() which returns the unordered set union of two lists.

“set intersection”

```
list sinter(L1, L2)  
list L1, L2;
```

The ordered set intersection of the lists **L1** and **L2** is returned; **L1** and **L2** must be ordered lists of elements of type **single**. (The input lists remain unchanged.)

```
Example:      list L, L1, L2;  
                L1 = list2(3, 4);  
                L2 = list3(2, 3, 5);  
                L  = sinter(L1, L2);      /* L = (3) */
```

Some related function:

usinter() which returns the unordered set intersection of two lists.

“set difference”

```
list sdiff(L1, L2)  
list L1, L2;
```

The ordered set difference of the lists **L1** and **L2** is returned, i.e. the ordered list of elements of **L1** which are not in **L2**; **L1** and **L2** must be ordered lists of elements of type **single**. (The input lists remain unchanged.)

```
Example:      list L, L1, L2;  
                L1 = list3(3, 4, 6);  
                L2 = list3(2, 3, 5);  
                L  = sdiff(L1, L2);      /* L = (4,6) */
```

Some related functions:

usdiff() which returns the unordered set difference of two lists.

ussdiff() which returns the unordered set symmetrical difference of two lists.

Those were only the most important list functions of the base system. SIMATH contains many more functions for I/O of lists, testing lists, permuting, and rotating list elements, merging, and comparing two lists, getting the size and length of a list, manipulating objects and atoms, etc. You will find a detailed description of all the base system functions in the on-line documentation or by the SM keyword index.

Note:

The list functions try to use as few new memory cells as possible from the workspace. Whenever possible, try to obtain the desired results by manipulating cell pointers; try to understand the effects of manipulating pointers on the list contents from the following two examples.

SIMATH instructions	effect
L1 = list2(1, 2); L2 = list2(4, 5); L = lconc(L1, L2);	L = (1,2,4,5) L1 = (1,2,4,5) L2 = (4,5)
lsfirst(L1, 0);	L = (0,2,4,5) L1 = (0,2,4,5) L2 = (4,5)
lsfirst(L2, 3);	L = (0,2,3,5) L1 = (0,2,3,5) L2 = (3,5)
 L1 = list2(1, 2); L2 = list2(4, 5); L = lcconc(L1, L2);	 L = (1,2,4,5) L1 = (1,2) L2 = (4,5)
lsfirst(L1, 0);	L = (1,2,4,5) L1 = (0,2) L2 = (4,5)
lsfirst(L2, 3);	L = (1,2,3,5) L1 = (0,2) L2 = (3,5)

2. The arithmetic package

2.1. Overview. The arithmetic package contains algorithms for computing over the integers \mathbb{Z} , the rational numbers \mathbb{Q} , the finite rings $\mathbb{Z}/m\mathbb{Z}$, the finite fields (Galois fields) \mathbb{F}_q (especially finite fields of characteristic 2), p -adic number fields \mathbb{Q}_p , the real numbers \mathbb{R} , the complex numbers \mathbb{C} , algebraic number fields, and function fields. Functions for basic arithmetic as well as higher algorithms are available (e.g. gcd and lcm, chinese remainder theorem, prime number testing, factorizing in \mathbb{Z} , discriminants, norm, and trace of number fields, etc.).

2.2. Data types. The arithmetic package supports the data types **single**, **int**, **rat**, **floating**, **complex**, **gfel**, **gf2el**, **nfel**, **pfel**, **rfunc**, and **afunc** (overview in §3.3); they will be explained in detail in the next section.

Integers of type **int** are “multiple-precision” integers. They are not subject to any size limitations (as long as they do not exceed the capacity of the memory).

Integer variables which, during program execution, have absolute value smaller than **BASIS**, (e.g. index variables) should be declared as **single** to “lighten” memory administration by reducing the number of references in the SIMATH stack; this is particularly important for recursive functions.

2.3. Internal representation. Large integers

Every integer $n \in \mathbb{Z}$ has a unique representation as a sum

$$n = \sum_{i=0}^k a_i \cdot \mathbf{BASIS}^i$$

with

$$\begin{aligned} k &\geq 0, \quad a_i \in \mathbb{Z}, \quad |a_i| < \mathbf{BASIS}, \\ a_i &= 0 \quad \text{or} \quad \text{sign } a_i = \text{sign } n, \\ a_k &\neq 0 \quad \text{if } k > 0. \end{aligned}$$

Internally, the a_i ’s are implemented as **atoms**. If $k = 0$, then n is represented by the **atom** a_0 , otherwise by the **list** (a_0, a_1, \dots, a_k) .

Large integers have the type designation **int**.

Residue class rings

The elements of a residue class ring

$$\mathbb{Z}/m\mathbb{Z} \quad (m \in \mathbb{Z}, m > 0)$$

are represented by elements of the smallest positive residue system

$$\{0, 1, \dots, m-1\}.$$

The functions dealing with residue class rings are called “modular integer ...”, or “modular single ...” for single precision moduli m .

When computing in $\mathbb{Z}/m\mathbb{Z}$, the programmer must make sure that the integer representations are within the prescribed residue system.

Elements of residue class rings have no special type designation. They should be declared as **single** or **int** depending on the size of the modulus.

Rational numbers

For the rational number $r \neq 0$, let $r = m/n$ be the unique representation as a fraction, with coprime integers m and n , $n > 0$. Then r is represented by the list (m, n) .

The rational number 0 is represented by the **atom** 0.

Rational numbers have the type designation **rat**.

Real numbers

A real number $f \neq 0$ has the form

$$f = \left(\overbrace{\sum_{i=0}^k a_i \mathbf{BASIS}^i}^{\text{mantissa}} \right) \cdot \mathbf{BASIS}^{\overbrace{e}^{\text{exponent}}}$$

with

$$k \geq 0, a_i, e \in \mathbb{Z} \text{ and } |a_i|, |e| < \mathbf{BASIS}, \\ a_i = 0 \text{ or sign } a_i = \text{sign } f.$$

The uniqueness of this representation is assured by requiring that $a_k \neq 0$ and $a_0 \neq 0$. f is then represented by the list

$$\left(\underbrace{e}_{\text{exponent}}, \underbrace{a_0, a_1, \dots, a_k}_{\text{mantissa digits}} \right).$$

The maximum length of the list used to represent the mantissa of a real number is given by the global variable **FL_EPS** (see §3.7). By default, real numbers have the representation

$$\left(\underbrace{e}_{\text{exponent}}, \underbrace{a_0, a_1, \dots, a_k}_{k+1 \text{ mantissa digits}} \right),$$

with $k + 1 \leq \mathbf{FL_EPS} + 1 = 6$.

Exceptions: in some functions, the return value will be computed as precisely as possible independently of **FL_EPS**; i.e. the return value will have as many mantissa digits as the input value. This is the case, for example, for the negation of a real number or the addition of 0.

You may, of course, change the default precision of 6 mantissa digits by setting the variable **FL_EPS** to the desired value. (See also the on-line documentation on **finit()**.)

In general, the following error estimate for a real number x represented by f (i.e. approximated) is valid

$$\left| \frac{x - f}{x} \right| \leq \frac{1}{2} \mathbf{BASIS}^{1-\mathbf{FL_EPS}}.$$

During a computation, if the exponent is greater than or equal to **BASIS** (“overflow”), then program execution will be terminated with the message “floating point overflow”. You can prevent this by using the function **flerr()** (see on-line documentation on **flerr()**). If the exponent is less than or equal to $-\mathbf{BASIS}$ (“underflow”), the real number will be interpreted as 0. Otherwise, you will always obtain a value which satisfies the estimate above. If necessary, computations within a SIMATH function will be done with greater precision (i.e. with more than **FL_EPS** mantissa digits).

The real number 0 is represented by the **atom** 0.

Real numbers have the type designation **floating**.

Complex numbers

A complex number $z \neq 0$ has the form $z = f + ig$ with real numbers f and g . $i = \sqrt{-1}$ is the complex unit. Then z is represented by the list (f, g) .

Complex numbers have the type designation **complex**.

Galois fields

Let p be a single precision prime and n a positive single precision integer. Let $g(X)$ be an irreducible, monic polynomial over $\mathbb{Z}/p\mathbb{Z}$ of degree n .

For computations in the Galois field $\text{Gf}(p^n)$, we consider the isomorphic field

$$\mathbb{Z}/p\mathbb{Z}[X] / (g(X) \cdot \mathbb{Z}/p\mathbb{Z}[X]).$$

Its elements are represented by polynomials over $\mathbb{Z}/p\mathbb{Z}$ of degree smaller than n , i.e. by the elements of the reduced residue system modulo $g(X)$.

When computing in a Galois field you must create a so called arithmetic list **AL** and if necessary, an irreducible monic polynomial $g(X)$, using the function

gfsalgen()

(*Galois field with single characteristic arithmetic list generator*). Functions computing in Galois fields require **AL** as a parameter (see the on-line documentation on Galois fields functions).

The Galois field element 0 is represented by the **atom** 0.

An element of $\text{Gf}(p^n)$ has the type designation **gfel** (Galois field element).

Galois fields of characteristic 2

Let $g(X)$ be any given irreducible monic polynomial over $\mathbb{Z}/2\mathbb{Z}$ of degree n . Then there exists an isomorphism

$$\text{Gf}(2^n) \cong \mathbb{Z}/2\mathbb{Z}[X] / (g(X) \cdot \mathbb{Z}/2\mathbb{Z}[X]).$$

Elements of $\text{Gf}(2^n)$ are represented by polynomials of degree smaller than n over $\mathbb{Z}/2\mathbb{Z}$, i.e. by the elements of the reduced residue system modulo $g(X)$. Let $p(X)$ be a polynomial of degree $m < n$ representing the element $b \in \text{Gf}(2^n)$, $b \neq 0$, i.e.

$$p(X) = \sum_{i=0}^m c_i X^i$$

with $c_i = 0$ or 1, for $0 \leq i < m$, and $c_m = 1$. Since each coefficient of $p(X)$ is either 0 or 1, $p(X)$ is internally represented in bit notation, as a list of singles

$$(m, a_k, a_{k-1}, \dots, a_1, a_0)$$

with

$$1 \leq k = \lfloor \frac{m}{30} \rfloor, a_i \in \mathbb{N}_0, |a_i| < \text{BASIS}$$

and

$$\begin{aligned} a_0 &= c_{29}2^{29} + c_{28}2^{28} + \dots + c_12^1 + c_02^0 \\ a_1 &= c_{59}2^{29} + c_{58}2^{28} + \dots + c_{31}2^1 + c_{30}2^0 \\ &\vdots \\ a_k &= c_{30k+29}2^{29} + c_{30k+28}2^{28} + \dots + c_{30k+1}2^1 + c_{30k}2^0. \end{aligned}$$

Each a_j ($0 \leq j \leq k$) is a single precision number, i.e. a 32-bit word with the first 2 bits used for list administration. The last 30 bits contain 30 coefficients of the polynomial $p(X)$. The monic irreducible polynomial $g(X)$ is also represented in special bit notation¹.

There are routines that convert a polynomial into bit representation (**udpm2tosb()**) and vice versa (**gf2eltoudpm2()**). See also the on-line documentation on **getgf2el()** and **putgf2el()** for input and output.

The 0 element of $\text{Gf}(2^n)$ is represented by the **atom** 0.

Elements of $\text{Gf}(2^n)$ has the type designation **gf2el** (Galois field of characteristic 2 element).

¹Similar to the special bit notation of polynomials over $\mathbb{Z}/2\mathbb{Z}$ there exist also matrices over $\mathbb{Z}/2\mathbb{Z}$ in special bit notation. See the documentation of **mamstomam2()**

Algebraic number fields

The algebraic number field $K = \mathbb{Q}(\alpha)$ is uniquely determined (up to isomorphism) by the minimal polynomial $f(X)$ of α . In SIMATH functions, the number field K is always specified by the polynomial $f(X)$.

For computations in K , we consider the isomorphic residue class field

$$\mathbb{Q}[X] / f(X) \cdot \mathbb{Q}[X] .$$

The elements of K correspond to polynomials over \mathbb{Q} of degree smaller than the degree of f , i.e. to the elements of the reduced residue system modulo f .

Let $g(X)$ be a polynomial representing an element $b \in K^*$, n the common denominator of the coefficients of $g(X)$, and G the list representation of $n \cdot g(X)$ as a dense polynomial over the integers (see §4.3.3). Then the algebraic number b is represented by the list (n, G) .

The algebraic number 0 is represented by the **atom** 0.

Algebraic numbers have the type designation **nfel** (number field element).

p -adic numbers

Let p be a single precision prime. Each element $a \neq 0$ of the p -adic field \mathbb{Q}_p has a unique p -adic series expansion of the form

$$a = \sum_{i=v_p(a)}^{\infty} a_i p^i,$$

with $a_i \in \mathbb{Z}$, $0 \leq a_i < p$ and $a_{v_p(a)} \neq 0$. (Here, v_p stands for the additive, normalized p -adic valuation on \mathbb{Q}_p .) Since infinite (non-periodic) series cannot be represented on computers, the elements of \mathbb{Q}_p have to be approximated. For any given $d \in \mathbb{Z}$

$$a' = \sum_{i=v_p(a)}^d a_i p^i$$

is called the approximation of a to degree d . Thus

$$v_p(a - a') \geq d.$$

If $d < v_p(a)$, a' is the empty sum, thus $a' = 0$. If $d \geq v_p(a)$, a' is represented by the list

$$(d, v_p(a), \underbrace{a_{v_p(a)}, \dots, a_d}_{d+1-v_p(a) \text{ places}}).$$

The p -adic number 0 is represented by the **atom** 0.

p -adic numbers have the type designation **pfel** (p -adic field element).

Rational functions

Currently, rational functions are implemented over the prime field $\mathbb{Z}/p\mathbb{Z}$ and over the rationals.

Let $K = \mathbb{Q}(X)$ or $\mathbb{Z}/p\mathbb{Z}(X)$ (the rational function field of transcendence degree 1 over $\mathbb{Z}/p\mathbb{Z}$), and $F \in K^*$. Then F has the unique representation $F = f_1/f_2$ with coprime polynomials $f_1, f_2 \in \mathbb{Z}[X]$ or $\mathbb{Z}/p\mathbb{Z}[X]$ respectively. F is represented by the list (f_1, f_2) .

The zero function is represented by the **atom** 0.

Rational functions have the type designation **rfunc**.

2.4. The arithmetic functions. For each data type supported by the arithmetic package, there are functions to perform basic arithmetic operations, tests, and comparisons. There are also utility functions for input/output from and to a file, the terminal or a string, and functions for conversion from one data type to another if this is possible. Furthermore, there are functions to perform operations specific to the different data types such as factorization for integers, chinese remainder theorem for modular integers, etc.

It would be impossible to give an exhaustive list of the arithmetic functions here. Instead, we shall give a short description of the basic arithmetic functions for each data type described in §4.2.3, and the name of some related functions. A detailed description of every arithmetic function contained in this package can be found in the on-line documentation or by the SM keyword index.

Note: If you want to list all SIMATH functions, which do a special operation (for example all functions, which add SIMATH objects), you can search for those functions with the SM keyword index (e.g. by searching for “sum”).

Here are some important functions for basic arithmetic operations. The conditions on the input parameters depend on the data types; we refer you to the on-line documentation for more details.

“sum”

int isum(A, B) int A, B;	rat rsum(A, B) rat A, B;	floating flsum(A, B) floating A, B;
int misum(M, A, B) int M, A, B;	nfel nfsum(F, A, B) pol F; nfel A, B;	pfel pfsum(p, A, B) single p; pfel A, B;
rfunc rfrsum(r, A, B) single r; rfunc A, B;	gf2el gf2sum(G, A, B) obj G; gf2el A, B;	gfel gfssum(p, AL, A, B) single p; list AL; gfel A, B;

These functions return $A + B$.

Some related functions: **csum()**, **mssum()**, **nfssum()**, **qnfisum()**, **qnfsum()**, **rfmsp1sum()**.

“difference”

int idif(A, B) int A, B;	rat rdif(A, B) rat A, B;	floating fldif(A, B) floating A, B;
int midif(M, A, B) int M, A, B;	nfel nfdif(F, A, B) pol F; nfel A, B;	pfel pfdif(p, A, B) single p; pfel A, B;
rfunc rfrdif(r, A, B) single r; rfunc A, B;	gf2el gf2dif(G, A, B) obj G; gf2el A, B;	gfel gfsdif(p, AL, A, B) single p; list AL; gfel A, B;

These functions return $A - B$.

Some related functions: **cdif()**, **iqnfdif()**, **msdif()**, **nfsdif()**, **qnfdif()**, **qnfdif()**, **rfmsp1dif()**, **rqnfdif()**.

“additive inverse (negation)”

int ineg(A) int A;	rat rneg(A) rat A;	floating fneg(A) floating A;
int mineg(M, A) int M, A;	nfel fneg(F, A) pol F; nfel A;	pfel pfneg(p, A) single p; pfel A;
rfunc rfrneg(r, A) single r; rfunc A;	gf2el gf2neg(G, A) obj G; gf2el A;	gfel gfsneg(p, AL, A) single p; list AL; gfel A;

These functions return $-\mathbf{A}$.

Some related functions: **cneg()**, **msneg()**, **nfsneg()**, **qnfneg()**, **rfmsp1neg()**.

“product”

int iprod(A, B) int A, B;	rat rprod(A, B) rat A, B;	floating flprod(A, B) floating A, B;
int miprod(M, A, B) int M, A, B;	nfel nfprod(F, A, B) pol F; nfel A, B;	pfel pfprod(p, A, B) single p; pfel A, B;
rfunc rfrprod(r, A, B) single r; rfunc A, B;	gf2el gf2prod(G, A, B) obj G; gf2el A, B;	gfel gfsprod(p, AL, A, B) single p; list AL; gfel A, B;

These functions return $\mathbf{A} * \mathbf{B}$.

Some related functions: `cprod()`, `gf2prodAL()`, `i22prod()`, `ifelprod()`, `ip2prod()`, `iproduct_lo()`, `isprod()`, `liprod()`, `liprodoe()`, `msprod()`, `nfeliprod()`, `nfelrprod()`, `nfsprod()`, `pfeliprod()`, `pfelrprod()`, `pfpprod()`, `qnfidprod()`, `qnfiprod()`, `qnfpprod()`, `qnfrprod()`, `rfmsp1prod()`, `rp2()`, `sprod()`, `sxprod()`, `gf2squ()`, `gf2squAL()`, `qnfidsquare()`, `qnfsquare()`, `sxsqu()`.

“exponentiation”

int iexp(A, n) int A; single n;	rat rexp(A, n) rat A; single n;	floating flsexp(A, n) floating A; single n;
int miexp(M, A, n) int M, A, n:	nfel nfexp(F, A, n) pol F; nfel A; single n;	pfel pfexp(p, A, n) single p, n; pfel A;
gf2el gf2exp(G, A, n) obj G; gf2el A; single n;	gfel gfsexp(p, AL, A, n) single p, n; list AL; gfel A;	

These functions return A^n .

Some related functions: **csexp()**, **flpow()**, **miexp_lo()**, **msexp()**, **qnfexp()**, **qnfidexp()**, **sexp()**.

“quotient”

int iquot(A, B) int A, B;	rat rquot(A, B) rat A, B;	floating flquot(A, B) floating A, B;
int miquot(M, A, B) int M, A, B;	nfel nfquot(F, A, B) pol F; nfel A, B;	pfel pfquot(p, A, B) single p; pfel A, B;
rfunc rfrquot(r, A, B) single r; rfunc A, B;	gf2el gf2quot(G, A, B) obj G; gf2el A, B;	gfel gfsquot(p, AL, A, B) single p; list AL; gfel A, B;

These functions return A / B .

Some related functions: **cquot()**, **flqrem()**, **flsquot()**, **ip2quot()**, **iqnfquot()**, **iqrem()**, **iqrem_lo()**, **irem()**, **irshift()**, **isqrem()**, **isquot()**, **isrem()**, **msquot()**, **nfelmodi()**, **nfsquot()**, **qnfiquote()**, **qnfquot()**, **qnfrquot()**, **rfmspl1quot()**, **rqnfquot()**, **sqrem()**.

“inverse”

rat rinv(A) rat A;	int miinv(M, A) int M, A;	nfel nfinv(F, A) pol F; nfel A;
pfel pfinv(p, A) single p; pfel A;	rfunc rfrinv(r, A) single r; rfunc A;	
gf2el gf2inv(G, A) obj G; gf2el A;	gfel gfsinv(p, AL, A) single p; list AL; gfel A;	

These functions return \mathbf{A}^{-1} .

Some related functions:

miinv_lo(), **msinv()**, **nfsinv()**, **qnfinv()**, **rfmsp1inv()**.

“square root”

int isqrt(A)	int misqrt(n, A)	floating flsqrt(A)
int A;	int A, n;	floating A;

These functions return $\sqrt{\mathbf{A}}$.

Some related functions: **csqrt()**, **iroot()**, **isqrt_lo()**, **miproot()**, **mipsqrt()**, **misqrtas()**, **misqrtsrch()**, **mppsqrtd()**, **mpsqrtd()**, **nf3sqrt()**, **ssqrt()**.

There are many other arithmetic functions which were not described here since they are defined only for some data types. Among those are functions to get the sign of a variable or the numerator or denominator of a rational number, and functions to compute the Jacobi-symbol, least common multiple, maximum or minimum of numbers; please refer to the on-line documentation or the SM keyword index for more details. The next category of functions perform tests and comparisons on the input.

is...

single isint(A) obj A;	single israt(A) obj A;	single isfloat(A) obj A;
single ismi(m, A) obj m, A;	single isnfel(F, A) pol F; nfel A;	single ispfel(p, A) single p; obj A;
single isrfr(r, A) obj r, A;	single isgf2el(G, A) obj G, A;	single isgfsel(p, AL, A) single p; list AL; obj A;

These functions return 1 if

$\mathbf{A} \in \mathbb{Z}$	$\mathbf{A} \in \mathbb{Q}$	$\mathbf{A} \in \mathbb{R}$
$\mathbf{A} \in \mathbb{Z}/\mathbf{m}\mathbb{Z}$	$\mathbf{A} \in \mathbf{Nf}_{\mathbf{F}}$	$\mathbf{A} \in \mathbb{Q}_{\mathbf{p}}$
$\mathbf{A} \in F(\mathbb{Q})$	$\mathbf{A} \in \mathbf{Gf}_{\mathbf{G}}(2^n)$	$\mathbf{A} \in \mathbf{Gf}_{\mathbf{G}}(\mathbf{p}^{\mathbf{n}})$

respectively; 0 otherwise.

Some related functions: **isgf2impsb()**, **isgfsal()**, **isms()**, **isnfels()**, **isqnfel()**, **isqnfint()**, **isqnfirat()**, **isrfmsp1()**, **islistgf2()**, **islistgfs()**, **islisti()**, **islistmi()**, **islistms()**, **islistnns()**, **islists()**.

“comparison”

single icomp(A, B) int A, B;	single rcomp(A, B) rat A, B;	single fcomp(A, B) floating A, B;
---	---	--

These functions return 1 if $\mathbf{A} > \mathbf{B}$, 0 if $\mathbf{A} = \mathbf{B}$, and -1 if $\mathbf{A} < \mathbf{B}$.

Some related functions: **ccomp()**, **qnfelcomp()**, **qnfidcomp()**.

There are, of course, more tests and comparisons which can be performed on the different data types. Some of the properties which can be tested are even/odd, and if a variable is a square, a unit, or equivalent to the value for unity associated with its data type; as usual, we refer you to the on-line documentation or the SM keyword index for details. Here is a table of some useful functions specific to some data types; we give only the name of the functions related to a function category and data type.

Function type	data type	function names
rounding truncating	int rat floating	inearesttor, itrunc rceil, rfloor ffloor, flround
absolute value	int rat floating complex	iabs, sabs rabs flabs cabsv
greatest com- mon divisor	int	iegcd, igcd, igcdf, igcd_lo, ihgcd, segcd, sgcd
logarithm	int rat floating	ilog2, ilog10, slog2 rlog2 fllog
Trigonometric functions	floating	flcos, flsin, fltrig
factorization	int nfel	ifact, ifact60, ifactcfe, ifactlf, ifactpp, sfact qnfielpifact, qnfpifact
primality testing	int	iecpt, iftpt, igkapt, isipprime, isiprime, isiprimemsg, isispprime, ispt, issprime
chinese remaindering	$\mathbb{Z}/m\mathbb{Z}$	micra, micran, milcra, miscra, mscra, mscran, mslcra
norm	nfel	nfnorm, qnfnorm
trace	nfel	nftrace, qnftrace

Function type	data type	function names
discriminant	nfel	nffielddiscr, qnfdisc
decomposition law	nfel	nfipdeclaw, nfspdeclaw
class number	nfel	abnfrelcl, abnfrelclmp
homomorphism	$\mathbb{Z}/m\mathbb{Z}$ nfel	mihom, mihoms, mitos, mshom, mshoms, qnfpihom
valuation	int rat nfel pfel	iaval, iavalint, imp2d, intpp, intppint, raval, ravalint, qnfaval, nfextofpadi, pfaval
field embedding	gf2el gfsel	gf2efe, gf2ies, gfsefe, gfsalgenies

Finally, the arithmetic package contains functions to get random integers, prime integers or Galois-field elements, and data type conversion routines, for example:

From	to
int	rat, floating, complex, nfel, pfel
floating	rat, complex
rat	floating, complex, $\mathbb{Z}/m\mathbb{Z}$, nfel, pfel, rfunc
nfel	polynomial, matrix row
gf2el	gfsel, polynomial
gfsel	gf2el

3. The polynomial package

3.1. Overview. SIMATH contains also functions for polynomial arithmetic over \mathbb{Z} , \mathbb{Q} , \mathbb{R} , \mathbb{C} , $\mathbb{Z}/m\mathbb{Z}$, \mathbb{F}_{p^n} , \mathbb{F}_{2^n} , \mathbb{Q}_p , and algebraic number fields. Higher algorithms for factorizing and computing Gröbner bases are also included in the package.

Furthermore, utility functions for convenient input/output to and from files or the terminal, and conversion to and from the different internal polynomial representations are available.

3.2. Convenient input/output. The input routines read a polynomial as a sum of monomials; the symbol `#` indicates the end of the polynomial. (Detailed information about the input format can be found in the on-line documentation of the respective input routines. See also the example program `m_polynomials.io.S` in `/usr/local/simath/examples/basics/`.) Either symbol `^` or `**` can be used for the exponent; valid input data are

```
27 X^5 Y1 - X Y1^3 Y2^3 + 9 X Y2^2 #
20X**5Y1-X Y1**3Y2**3+9Y2**2X+7X**5Y1#
27*X^5*Y1^1*Y2^0 - 1*X^1*Y1^3*Y2^3 + 9*X^1*Y1^0*Y2^2 #
```

The output routines use the first form above to display polynomials.

3.3. Internal representation. When developing algorithms, one should always use the most appropriate internal list representation for polynomials. For this purpose, the SIMATH system offers a distributive representation and two recursive representations, namely the recursive sparse and the recursive dense representation.

In general, SIMATH programs use the recursive sparse representation; the input routines return the sparse representation of polynomials. The name of a function dealing with polynomials indicates if a different representation is required as a parameter to the function.

Polynomials in zero variables are considered as elements of the coefficient ring.

The zero polynomial is represented by the **atom** 0.

Polynomials have the type designation **pol**.

The recursive sparse representation

Let R_i be a polynomial ring in i variables, $i \geq 0$, over the commutative ring R . For $n > 0$ consider R_n as a polynomial ring in one variable over R_{n-1}

$$R_n = R[X_1, \dots, X_n] = R_{n-1}[X_n].$$

Then a polynomial $p \in R_n$, $p \neq 0$, has the form

$$p = p(X_1, \dots, X_n) = \sum_{i=1}^k p_i(X_1, \dots, X_{n-1}) X_n^{e_i}$$

with $k \leq \deg(p) + 1$, $p_i \in R_{n-1}$, $p_i \neq 0$, and decreasing exponents

$$e_1 > e_2 > \dots > e_k \geq 0.$$

Then the list

$$p^{(sp)} := (e_1, p_1^{(sp)}, \dots, e_k, p_k^{(sp)})$$

is the sparse representation of the polynomial p .

Example: the polynomial

$$p(X, Y) = 27X^5Y - XY^3 + Y^3 + 9XY^2 = (-X^1 + X^0)Y^3 + (9X^1)Y^2 + (27X^5)Y$$

has the sparse representation

$$p^{(sp)} = (3, (1, -1, 0, 1), 2, (1, 9), 1, (5, 27)).$$

The recursive dense representation

The polynomial $p \in R_n$ has the form

$$p(X_1, \dots, X_n) = \sum_{j=0}^d q_j(X_1, \dots, X_{n-1}) X_n^j,$$

where $d = \deg(p)$ and $q_j \in R_{n-1}$. Then the list

$$p^{(dns)} = (d, q_d^{(dns)}, q_{d-1}^{(dns)}, \dots, q_0^{(dns)})$$

is the dense representation of the polynomial p .

Example: the polynomial

$$\begin{aligned} p(X, Y) &= (-X^1 + 0X^0)Y^3 + (9X^1 + 0X^0)Y^2 \\ &\quad + (27X^5 + 0X^4 + 0X^3 + 0X^2 + 0X^1 + 0X^0)Y^1 + 0Y^0 \end{aligned}$$

has the dense representation

$$p^{(dns)} = (3, (1, -1, 0), (1, 9, 0), (5, 27, 0, 0, 0, 0, 0), 0).$$

The distributive representation

Let

$$p(X_1, \dots, X_n) = \sum_{i=1}^k m_i(X_1, \dots, X_n)$$

be the sum of monomials m_i over R_n , $m_i \neq 0$, with

$$m_i(X_1, \dots, X_n) = c_i X_n^{e_{i,n}} X_{n-1}^{e_{i,n-1}} \dots X_1^{e_{i,1}}, \quad c_i \in R, \quad c_i \neq 0,$$

so that the exponent-tuples

$$e_i = (e_{i,n}, e_{i,n-1}, \dots, e_{i,1}) \quad 1 \leq i \leq k$$

are lexicographically ordered, i.e.

$$e_i > e_j \text{ for } i < j.$$

Then the list

$$p^{(di)} = (c_1, e_1, c_2, e_2, \dots, c_k, e_k).$$

is the distributive representation of the polynomial p .

Example: the polynomial

$$p(X, Y) = -XY^3 + Y^3 + 9XY^2 + 27X^5Y$$

has the distributive representation

$$p^{(di)} = (-1, (3, 1), 1, (3, 0), 9, (2, 1), 27, (1, 5))$$

3.4. The polynomial functions. As with the arithmetic package, it would be impossible to give an exhaustive list of all the functions included in this package. We shall give a short description of the basic polynomial arithmetic functions, some useful and interesting functions particular to polynomials and functions to test polynomials as well as the name of some related functions. A detailed description of all the polynomial functions can be found in the on-line documentation and by the SM keyword index.

The following notation is used for the different representations

p polynomial in recursive sparse representation
dp polynomial in recursive dense representation
dip polynomial in distributive representation
up univariate polynomial
udp univariate dense polynomial

Here are some important polynomial arithmetic functions. The conditions on the input parameters depend on the polynomial representation and data type; we refer you to the on-line documentation or the SM keyword index for more details.

“polynomial sum”

pol psum(r,P,Q)	pol prsum(r,P,Q)	pol pmisum(r,m,P,Q) pol dpmisum(r,m,P,Q)
pol dipisum(r,P,Q) pol udpisum(P,Q)	pol diprsum(r,P,Q) pol udprsum(P,Q)	pol udpmisum(m,P,Q)
pol pnfsum(r,F,P,Q) pol dipnfsum(r,F,P,Q)	pol ppfsum(r,p,P,Q) pol udppfsum(p,P,Q)	pol diprfrsum(r1,r2,P,Q)

with the following parameters type **single r1, r2, r, p;**
 int m;
 pol F, P, Q;

These functions return $\mathbf{P} + \mathbf{Q}$.

Some related functions: **cdprfmssp1sum()**, **cdprsum()**, **dipgfssum()**, **dipmipsum()**, **dipmspsum()**, **dippisum()**, **dpmssum()**, **pgf2sum()**, **pgfssum()**, **pmssum()**, **prfmssp1sum()**, **udpmssum()**.

The functions for polynomial difference (***dif**) and polynomial product (***prod**) are analogous to those for polynomial sum. See the keyword index for more details.

“polynomial additive inverse (negation)”

pol pineg(r, P)	pol prneg(r, P)	pol pmineg(r, m, P) pol dpmineg(r, m, P)
pol dipineg(r, P) pol udpineg(P)	pol diprneg(r, P) pol udprneg(P)	pol udpmineg(m, P)
pol pnfneg(r, F, P) pol dipfneg(r, F, P)	pol ppfneg(r, p, P) pol udppfneg(p, P)	pol diprfrneg(r1, r2, P)

with the following parameters type **single r1, r2, r, p;**
 int m;
 pol F, P;

These functions return $-P$.

Some related functions: **dipgfsneg(), dipmipneg(), dipmspneg(), dippineg(), dpmsneg(), pgf2neg(), pgfsneg(), pmsneg(), prfmsp1neg(), udpmsneg().**

“polynomial exponentiation”

pol piexp(r, P, n)	pol prexp(r, P, n)	pol pmiexp(r, m, P, n)
pol pnfexp(r, F, P, n)	pol ppfexp(r, p, P, n) pol udppfexp(p, P, n)	

with the following parameters type **single r, p, n;**
 int m;
 pol F, P;

These functions return P^n .

Some related functions: **pgf2exp(), pgfsexp(), pmsexp(), upmimpexp().**

“polynomial quotient”

pol piquot (r, P, Q)	pol prquot (r, P, Q)	pol pmiquot (r, m, P, Q)
		pol udpmiquot (m, P, Q)

pol pnfquot(r, F, P, Q)

[illegible]

These functions return **P** / **Q**.

Some related functions: `cdprfmsplupq()`, `pgf2qrem()`, `pgf2quot()`, `pgfsqrem()`, `pgfsquot()`, `pipsrem()`, `piqrem()`, `pirem()`, `pmimidqrem()`, `pmipsrem()`, `pmiqrem()`, `pmirem()`, `pmiupmiquote()`, `pmspsrem()`, `pmsqrem()`, `pmsquot()`, `pmsrem()`, `pmsupmsquote()`, `pnfqrem()`, `pnfrem()`, `ppmvquote()`, `ppvquote()`, `prfmsplqrem()`, `prqrem()`, `prrem()`, `udpipsrem()`, `udpmiqrem()`, `udpmirem()`, `udpmsqrem()`, `udpmsquot()`, `udpmsrem()`, `udprqrem()`, `upmimprem()`, `upmirem()`, `upmsmprem()`, `upmsrem()`.

There are many other polynomial arithmetic functions which were not described here. Please refer to the on-line documentation for more details.

Next are Groebner bases functions.

“Groebner basis”

list dipigb (r, PL) list diprgb (r, PL) list dipmigb (r, p, PL)
list dipnfgb (r, F, PL) list diprfrgb (r1, r2, PL)

with the following parameters type

```
single r, r1, r2;  
int p;  
pol F;  
list PL;
```

These functions return the Groebner basis of the polynomial list **PL**.

Some related functions: `dipgfsqb()`, `dipmsqb()`, `dippiqb()`, `diprfrqb()` .

There are also functions for Groebner bases augmentation and recursion, and, for polynomials in polynomial rings, functions to compute (reduced) comprehensive Groebner bases and Groebner tests. See also the examples in `/usr/local/simath/examples/advanced/Groebner_basis/`.

What follows are functions which perform tests on their arguments.

is...

single ispi(r, P) single ispr(r, P) single ispmi(r, m, P)
single isdpi(r, P) single isdpr(r, P) single isdpmi(r, m, P)

single isdppf(r, p, P)

with the following parameters type **single r, p;**
 int m;
 pol P;

These functions return 1 if

$P \in \mathbb{Z}[X]$ $P \in \mathbb{Q}[X]$ $P \in \mathbb{Z}/m\mathbb{Z}[X]$

$P \in \mathbb{Q}_p[X]$

respectively; 0 otherwise.

Some related functions: **ispgf2(), ispgfs(), isdpms(), ispms().**

You can also test if a polynomial is equivalent to one, irreducible, squarefree, a unit, monomial, etc.; as usual, we refer you to the on-line documentation or the SM keyword index for details. Here is a table of some useful functions specific to some types of polynomials; we give only the name of the functions related to a function category.

Function type	function names
degree	diptdg, ptdegree, pdegree, pdegreesv
derivation	pderiv, pldderiv, pgf2deriv, pgfsderiv, pldderiv, pmldderiv, pmsderiv, pnfderiv, ppfderiv, prderiv, prfmplderiv, udlmldderiv, udlpmsderiv, pderivsv, pldderivsv, pgf2derivsv, pgfsderivsv, plderivsv, pmldderivsv, pmsderivsv, pnfderivsv, ppfderivsv, prderivsv

Function type	function names
content	dipicp, dippicp, pgfsucont, picont, picontpp, piicont, pmiaucont, pmsucont, pnfcont, pnfcontpp, udpicontpp
discriminant	pidiscr, pidiscrhank, pmidiscr, pmidiscrhank, pmsdiscr, upireddiscc, upprmsp1disc, upprmsp1redd
resultant	pgfsres, piresbez, pirescoll, pirescspec, piressylv, pmires, pmirescoll, pmsres, pmsrescoll, pprmsp1ress, upiresulc, upmires, upmiresulc, upmsres, upmsresulc, upprmsp1ress
greatest common divisor	pigcdcf, pmigcdcf, pmsgcdcf, pnfgcdcf, upmigcd, upmsgcd, upgf2gcd, upgfsgcd, upmigcd, upmsgcd, upnfgcd, uprfmsp1egcd
chinese remaindering	picra, picran
homomorphism	cdprfmosp1mh, cdprzmodhom, pimidhom
factorization	pifact, pisfact, pnffact, pnfsfact, spifact, spnffact, upgf2bofact, upgf2ddfact, upgf2nfact, upgf2sfact, upgfsbfact, upgfsbfzm, upgfsbofact, upgfscfact, upgfsddfact, upgfsrelpfac, upgfssfact, upifact, upm2cfact, upmibfact, upmibfzm, upmibofact, upmicfact, upmiddfact, upmirelpfact, upmisfact, upmsbfact, upmsbfzm, upmscfact, upmsddfact, upmsrelpfact, upmssfact, upnffact, upnfsfact, uprfact, uspiapf, uspifact, uspnffact, uspprmosp1apf
squarefree part	upgf2sfp, upgfssfp, upisfp, upmisfp, upmssfp, upnfsfp
root finding	udpflrf, udpirf, udprf, udprrf, upgfsrf, upmirf, upmsrf

Function type	function names
evaluation	pceval, pfeval, pgf2eval, pgfseval, pgfsevalsv, pieval, pievalsv, pigf2evalfvs, pigfsevalfvs, pinfevalfvs, pmieval, pmievalsv, pmseval, pmsevalsv, pnfeval, pnfevalsv, ppfeval, ppfevalsv, preval, prevalsv, prnfevalfvs, upgf2eval, upinfeevals, upinfeval, upinfseval, upnfeval, upprmsp1afes, uprnfeval, uprnfseval
field embedding	pgf2efe, pgfsefe
substitution	pisubst, pisubstsv, pmisubst, pmisubstsv, pmssubst, pmssubstsv, ppfsubst, ppfsubstsv, prsubst, prsubstsv
translation	pitrans, pitransav, pmitrans, pmitransav, pmstrans, pmstransav, pnfttrans, pnfttransav, ppfttrans, ppfttransav, prtrans, prtransav

The polynomial package contains also some functions, which need specific variable lists. For more information on variable lists, see the on-line documentation of **fgetv1()** or use the SM keyword index and search for “variable” and “list”.

Finally, the polynomial package contains functions to change the order of variables, transform a constant into polynomial, for converting from one representation to another and from one polynomial type to another (see table).

From	to
$\mathbb{Z}[X]$	$\mathbb{Z}/m\mathbb{Z}[X], \mathbb{Q}[X], \mathbf{Nf}[X], \mathbb{R}[X], \mathbb{C}[X],$
$\mathbb{Q}[X]$	$\mathbb{Q}_p[X], \mathbb{Q}(X)$
$\mathbf{Nf}[X]$	$\mathbb{Z}/m\mathbb{Z}[X], \mathbf{Nf}[X], \mathbb{R}[X], \mathbb{C}[X],$
$\mathbf{Gf}(2^n)[X]$	$\mathbb{Q}_p[X], \mathbb{Q}(X)$
$\mathbf{Gf}(p^n)[X]$	$\mathbb{Q}[X], \mathbf{Gf}[X]$
	$\mathbf{Gf}(p^n)[X]$
	$\mathbf{Gf}(2^n)[X]$
dp	p
dip	p
p	dp, dip

4. The matrix/vector package

4.1. Overview. This package contains functions for matrix and vector computations over \mathbb{Z} , \mathbb{Q} , $\mathbb{Z}/m\mathbb{Z}$, \mathbb{F}_q , polynomial rings, number fields, and rational function fields.

There are arithmetic functions for addition, multiplication, etc., input/output functions, functions for data type conversion and higher algorithms to compute the Hermite normal form, characteristic polynomial, determinant, etc. of a matrix; the scalar product, length, etc. of vectors.

4.2. Internal representation. Vector

A vector V of dimension $n > 0$ over a structure S is represented as a list

$$V = (a_1, \dots, a_n), \quad a_1, \dots, a_n \in S.$$

The zero vector is *not* represented by the **atom** 0; it has the form

$$\underbrace{(0, \dots, 0)}_{n \text{ times}}.$$

Vectors have the type designation **vec**.

Matrix

The rows of an $m \times n$ matrix $M = (a_{ij})$ over S are implemented as vectors

$$Z_i = (a_{i1}, \dots, a_{in}), \quad i = 1, \dots, m.$$

Then the matrix M is represented by the list of its row vectors

$$M = (Z_1, \dots, Z_m).$$

Example: the list $M = ((1 \ 2 \ 3) \ (4 \ 5 \ 6))$ represents the matrix

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

The zero matrix is *not* represented by the **atom** 0; it has the list representation

$$\underbrace{((\underbrace{0, \dots, 0}_{n \text{ times}}, \dots, \underbrace{0, \dots, 0}_{n \text{ times}}))}_{m \text{ times}}$$

There is a special bit representation for matrices over $\mathbb{Z}/2\mathbb{Z}$. Such a matrix is represented as a list

$$(n, r_1, \dots, r_m),$$

where n is the number of the columns (a positive single precision number) and the r_i 's are represented by lists of singles with $\text{llength}(r_i) = \frac{n}{30} + 1$.

For example: The matrix

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

is represented by the list

$$(3, (7), (5), (3), (1)).$$

SIMATH functions which use the special bit notation of a matrix over $\mathbb{Z}/2\mathbb{Z}$ all begin with **mam2**. There do not exist vectors over $\mathbb{Z}/2\mathbb{Z}$ in special bit representation. Use $1 \times n$ matrices instead.

Matrices have the type designation **matrix**.

4.3. The matrix/vector functions. Here again, it would be impossible to list all the functions included in this package. We shall give a short description of the basic matrix/vector arithmetic functions, some useful and interesting functions particular to matrices/vectors and functions to test matrices/vectors as well as the name of the related functions. A detailed description of all the matrix/vector functions can be found in the on-line documentation and the SM keyword index.

Here are some important matrix arithmetic functions. The conditions on the input parameters depend on the data types and the representation; we refer you to the on-line documentation or the SM keyword index for more details.

“matrix sum”

matrix maisum (M, N)	matrix marsum (M, N)
matrix mamisum (m, M, N)	matrix manfsum (F, M, N)
matrix mapisum (r, M, N)	matrix maprsum (r, M, N)
matrix mapmisum (r, m, M, N)	matrix mapnfsum (r, F, M, N)

with the following parameters type

single r;
int m;
pol F;
matrix M, N;

These functions return $M + N$.

Some related functions: **magf2sum()**, **magfssum()**, **mam2sum()**, **mamssum()**, **manfssum()**, **mapgf2sum()**, **mapgfssum()**, **mapmssum()**, **marfmsp1sum()**, **marfrsum()**, **masum()**.

The functions for matrix difference (***dif**) and matrix product (***prod**) are analogous to those for matrix sum. See the keyword index for more details.

“matrix additive inverse (negation)”

matrix maineg(M)	matrix marneg(M)
matrix mamineg(m, M)	matrix manfneg(F, M)
matrix mapineg(r, M)	matrix maprneg(r, M)
matrix mapmineg(r, m, M)	matrix mapnfneg(r, F, M)

with the following parameters type **single r;**
 int m;
 pol F;
 matrix M;

These functions return $-\mathbf{M}$.

Some related functions: **magf2neg()**, **magfsneg()**, **mamsneg()**, **manfsneg()**, **mapgf2neg()**, **mapgfsneg()**, **mapmsneg()**, **marfmsp1neg()**, **marfrneg()**, **maneg()**.

“matrix exponentiation”

matrix maiexp (M, n)	matrix marexp (M, n)
matrix mamiexp (m, M, n)	matrix manfexp (F, M, n)
matrix mapiexp (r, M, n)	matrix maprexp (r, M, n)
matrix mapmiexp (r, m, M, n)	matrix mapnfexp (r, F, M, n)

with the following parameters type **single r, n;**
 int m;
 pol F;
 matrix M;

These functions return M^n .

Some related functions: **magf2exp()**, **magfsexp()**, **mam2exp()**, **mamsexp()**,
manfsexp(), **mapgf2exp()**, **mapgfsexp()**, **mapmsexp()**, **marfmssp1exp()**,
marfrexp().

“matrix inverse”

matrix maiinv (M)	matrix marinv (M)
matrix mamiinv (m, M)	matrix manfinv (F, M)
matrix mapiinv (r, M)	matrix maprinv (r, M)
matrix mapmiinv (r, m, M)	matrix mapnfinv (r, F, M)

with the following parameters type **single r;**
 int m;
 pol F;
 matrix M;

These functions return M^{-1} .

Some related functions: **magf2inv()**, **magfsinv()**, **mam2inv()**, **mamsinv()**,
manfsinv(), **mapgf2inv()**, **mapgfsinv()**, **mapmsinv()**, **marfmssp1inv()**, **marfrinv()**.

Here are some important vector arithmetic functions. The conditions on the input parameters depend on the data types and the representation; we refer you to the on-line documentation or the SM keyword index for more details.

“vector sum”

vec vecisum(U, V)	vec vecrsum(U, V)
vec vecmisum(m, U, V)	vec vecnfsum(F, U, V)
vec vecpisum(r, U, V)	vec vecprsum(r, U, V)
vec vecpmisum(r, m, U, V)	vec vecpnfsum(r, F, U, V)

with the following parameters type **single r;**
 int m;
 pol F;
 vec U, V;

These functions return $\mathbf{U} + \mathbf{V}$.

Some related functions: **vecgf2sum()**, **vecgfssum()**, **vecmssum()**, **vecnfssum()**, **vecpgf2sum()**, **vecpgfssum()**, **vecpmssum()**, **vecrfmnsplsum()**, **vecrfrsum()**, **vecsum()**.

The functions for vector difference (***dif**) are analogous to those for vector sum. See the keyword index for more details.

“vector additive inverse (negation)”

vec vecineg(V)	vec vecrneg(V)
vec vecmineg(m, V)	vec vecnfneg(F, V)
vec vecpineg(r, V)	vec vecprneg(r, V)
vec vecpmineg(r, m, V)	vec vecpnfneg(r, F, V)

with the following parameters type **single r;**
 int m;
 pol F;
 vec V;

These functions return $-\mathbf{V}$.

Some related functions: **vecgf2neg()**, **vecgfsneg()**, **vecmsneg()**, **vecnfsneg()**, **vecpgf2neg()**, **vecpgfsneg()**, **vecpmsneg()**, **vecrfmnsplneg()**, **vecrfrneg()**, **vecneg()**.

“vector scalar product”

int vecisprod(U, V)	rat vecrsprod(U, V)
int vecmisprod(m, U, V)	nfel vecnfsprod(F, U, V)
pol vecpisprod(r, U, V)	pol vecprsprod(r, U, V)
pol vecpmisprod(r, m, U, V)	pol vpnfsprod(r, F, U, V)

with the following parameters type **single r;**
 int m;
 pol F;
 vec U, V;

These functions return the **scalar product** $\langle U, V \rangle$.

Some related functions: **vecgf2sprod()**, **vecgfssprod()**, **vecmssprod()**,
vecnfsprod(), **vecpgf2sprod()**, **vecpmssprod()**, **vecrfmsp1sprod()**,
vecfrsprod(), **vpgfssprod()**.

Here is the matrix vector multiplication function. The conditions on the input parameters depend on the data types and the representation; we refer you to the on-line documentation or the SM keyword index for more details.

“matrix vector multiplication”

vec maivecmul(M, V)	vec marvecmul(M, V)
vec mamivecmul(m, M, V)	vec manfvecmul(F, M, V)
vec mapivecmul(r, M, V)	vec maprvecmul(r, M, V)
vec mapmivecmul(r, m, M, V)	vec mapnfvecmul(r, F, M, V)

with the following parameters type **single r;**
 int m;
 pol F;
 matrix M;
 vec V;

These functions return $M * V$.

Some related functions: **magf2vecmul()**, **magfsvecmul()**, **mamsvecmul()**,
manfsvecmul(), **mapgfsvecmul()**, **mapmsvecmul()**, **marfmsp1vmul()**,
marfrvecmul(), **mavecmul()**.

The next category of functions perform tests on the input.

is...

single ismai(M)	single ismar(M)	single ismami(m, M)
single ismapi(r, M)	single ismapr(r, M)	single ismapmi(r, m, M)
single isveci(V)	single isvecr(V)	single isvecms(n, V)
single isvecpi(r, V)	single isvecpr(r, V)	single isvecpms(r, n, V)

with the following parameters type **single r, n;**
 int m;
 matrix M;
 vec V;

These functions return 1 if

$\mathbf{M} \text{ over } \mathbb{Z}$	$\mathbf{M} \text{ over } \mathbb{Q}$	$\mathbf{M} \text{ over } \mathbb{Z}/\mathbf{m}\mathbb{Z}$
$\mathbf{M} \text{ over } \mathbb{Z}[X]$	$\mathbf{M} \text{ over } \mathbb{Q}[X]$	$\mathbf{M} \text{ over } \mathbb{Z}/\mathbf{m}\mathbb{Z}[X]$
$\mathbf{V} \text{ over } \mathbb{Z}$	$\mathbf{V} \text{ over } \mathbb{Q}$	$\mathbf{V} \text{ over } \mathbb{Z}/\mathbf{n}\mathbb{Z}$
$\mathbf{V} \text{ over } \mathbb{Z}[X]$	$\mathbf{V} \text{ over } \mathbb{Q}[X]$	$\mathbf{V} \text{ over } \mathbb{Z}/\mathbf{n}\mathbb{Z}[X]$

respectively; 0 otherwise.

Some related functions: **isma()**, **isma_()**, **ismadp()**, **ismadpi()**, **ismadpms()**,
ismadpr(), **ismaeqel()**, **ismagf2()**, **ismagfs()**, **ismams()**, **ismanf()**, **ismanfs()**,
ismap(), **ismapgf2()**, **ismapgfs()**, **ismapms()**, **ismarfmsp1()**, **ismarfr()**, **ismas()**,
isvec(), **isvec_()**, **isvecdp()**, **isvecdpi()**, **isvecdpms()**, **isvecdpr()**, **isvecgf2()**,
isvecgfs(), **isvecms()**, **isvecnf()**, **isvecnfs()**, **isvecp()**, **isvecpgf2()**, **isvecpgfs()**,
isvecrfmsp1(), **isvecrfr()**.

You can also test if a matrix or a vector is the zero matrix or the zero vector; as usual, we refer you to the on-line documentation or the SM keyword index for details.

Here is a table of some useful functions specific to some types of matrices/vectors; we give only the name of the functions related to a function category.

Function type	function names
characteristic polynomial	machpol, magf2chpol, magfschpol, maichpol, mamichpol, mamschpol, manfchpol, mapgf2chpol, mapgfschpol, mapichpol, mapmichpol, mapmschpol, mapnfchpol, maprchpol, marchpol, marfrchpol
eigenvalues	magf2evifcp, magfsevfcp, maiev, maievifcp, mamiev, mamievifcp, mamsev, mamsevifcp, marev, marevifcp
determinant	mafldet, magf2det, magfsdet, maidet, mamidet, mamsdet, manfdet, manfsdet, mapgf2det, mapgfsdet, mapidet, mapmidet, mapmsdet, mapnfdet, maprdet, mardet, marfmspldet, marfrdet
rank	magf2rk, magfsrk, manfrk, manfsrk, marfmsplrk, marfrk, marrk
trace	magf2trace, magfstrace, maitrace, mamitrace, mamstrace, manftrace, mapgf2trace, mapgfstrace, mapitrace, mapmitrace, mapmstrace, mapnftrace, maprtrace, marfrtrace, martrace
transpose	mactransp, mam2transp, matransp
construction	cdmarfmsplid, cdmarid, maconsdiag, maconszero, magf2cons1, magfscons1, maicons1, mam2conszero, mam2um, mamicons1, mamscons1, manfcons1, mabfscons1, mapgf2cons1, mapgfscons1, mapicons1, mapmicons1, mapmscons1, mapnfcons1, maprcons1, marcons1, marfmsplc1, marfrcons1
delete	madellrc, madelsc, madelsr, madelsrc
embedding	magf2efe, magfsefe, mapgf2efe, mapgfsefe, vecgf2efe, vecgfsefe, vecpgf2efe, vecpgfsefe
elementary divisor form	maiedfcf, maupmipedfcf, maupmspedfcf, maupredfcf

Function type	function names
Hermite normal form	cdmarfmsp1hr , cdmarhermred , maiherm , maihermltne , maihermltpe , maihermspec , maupmshersp
LLL reduction	maillred , marllred
null space basis	magfsnsb , mam2gnsb , mam2nsb , maminsb , mamsnsb , manfnsb , manfsnsb , marfmsp1nsb , marfrnsb , marnsb
solution of a system of linear equations	magfssle , mamiclanssle , mamilanssle , mamilftssle , mamssle , manfssle , manfssle , marfmsp1ssle , marfrssle , marssle
linear combination	vecgfslc , vecilc , vecmilc , vecmslc , vecnflc , vecnfslc , vecpgfslc , vecpilc , vecpmilc , vecpmslc , vecpnflc , vecprlc , vecrfmsp1lc , vecrfrlc , vecrlc
unimodular transformation	veciunimtr , vecupmsunimt

Finally, the matrix/vector package contains functions to converting from one data type to another (see table for some examples).

From matrix over	to matrix over
\mathbb{Z} \mathbb{Q} $\mathbf{Nf}[X]$ $\mathbf{Gf}(2^n)$ $\mathbf{Gf}(p^n)$	$\mathbb{Z}/m\mathbb{Z}$, \mathbb{Q} , \mathbf{Nf} , $\mathbf{Gf}(2^n)$, $\mathbf{Gf}(p^n)$, $\mathbb{Z}[X]$ $\mathbb{Z}/m\mathbb{Z}$, \mathbf{Nf} , $\mathbb{Q}[X]$ $\mathbb{Q}[X]$, $\mathbf{Gf}(p^n)$ $\mathbf{Gf}(2^n)$
$\mathbb{Z}[X]$ $\mathbb{Q}[X]$	$\mathbb{Z}/m\mathbb{Z}[X]$, $\mathbb{Q}[X]$, $\mathbf{Nf}[X]$, $\mathbb{Q}(X)$ $\mathbb{Z}/m\mathbb{Z}[X]$, $\mathbf{Nf}[X]$

5. The elliptic curves package

5.1. Overview. SIMATH contains many programs for elliptic curves over the rational numbers, over prime fields, over finite fields of characteristic 2 and over algebraic number fields. The higher algorithms included in this package are for example the algorithms of Manin and Cremona for computing the basis of the Mordell Weil group over \mathbb{Q} . For elliptic curves over \mathbb{Q} and over quadratic number fields we have an efficient internal representation.

5.2. Internal representation. An elliptic curve over a field K can be given in the long Weierstrass normal form

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

with coefficients $a_1, \dots, a_6 \in K$. For elliptic curves over finite fields or over number fields of degree > 2 , there is no special internal representation; all functions for those curves require the coefficients a_1, \dots, a_6 as arguments.

Elliptic curves over \mathbb{Q} and quadratic number fields

For the initialization of an elliptic curve

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

over \mathbb{Q} or over a quadratic number field $K = \mathbb{Q}(\sqrt{D})$, use the function **ecrinit()** or **ecqnfinit()** with the parameters a_1, \dots, a_6 in \mathbb{Q} or in K . They return a list of lists

$$L = (L_1 \ L_2 \ L_3 \ L_4) \quad \text{for } \mathbb{Q}$$

or

$$L = (L_1 \ L_2 \ L_3 \ L_4 \ L_5) \quad \text{for } K,$$

which is the internal representation of the elliptic curve. The lists L_1, \dots, L_5 contain the following data:

For an elliptic curve E over \mathbb{Q} in long Weierstrass normal form:

List	data
L_1	contains the data of the actual model of E . Functions which use or change this data begin with ecrac for e(liptic) c(urve over) r(ational numbers), a(ctual) c(urve)
L_2	contains the data of the isomorphic minimal model of restricted type, i.e. the coefficients a_1, \dots, a_6 of the Weierstrass form are the integral coefficients of a global minimal model for E over \mathbb{Q} with $a_1, a_3 \in \{0, 1\}$ and $a_2 \in \{-1, 0, 1\}$. Functions which use or change this data begin with ecimin for e(liptic) c(urve with) i(nteger coefficients,) min(imal model)
L_3	contains the data of an isomorphic model in short Weierstrass normal form (sWNF). If the model in L_2 is already in sWNF, then we use the coefficients a_4 and a_6 from the list L_2 . Otherwise, we transform the curve in an equation of the form $Y^2 = X^3 - 27c_4X - 54c_6$ with c_4, c_6 from the list L_{13} . Functions which use or change this data begin with ecisnf for e(liptic) c(urve with) i(nteger coefficients,) s(hort) n(ormal) f(orm)
L_4	contains the invariants of E which are independent on the special model. Functions which use or change this data begin with ecrinv for e(liptic) c(urve over) r(ational numbers,) inv(ariants of the curve)

In detail, the lists L_1, \dots, L_4 are of the following form:

$$L_1 = (L_{11} \ L_{12} \ L_{13} \ L_{14} \ L_{15} \ L_{16} \ L_{17})$$

where

$L_{11} = (a_1 \ a_2 \ a_3 \ a_4 \ a_6)$ are the coefficients of the actual model of E/\mathbb{Q} ,

$L_{12} = (b_2 \ b_4 \ b_6 \ b_8)$ and

$L_{13} = (c_4 \ c_6)$ are the Tate values for the actual model,

$L_{14} = (D \ FD)$ are the discriminant D and the factorization FD of D ; $FD = (p_1 \ e_1 \ \dots)$ with $D = p_1^{e_1} p_2^{e_2} \dots$,

$L_{15} = (TP \ GT)$ contains a list with all torsion points of the actual model (TP) and a list of the generators of the torsion group of the actual model (GT),

$L_{16} = (BTm \ Bts)$ are the parameters of the birational transformation from the actual model to the minimal model ($BTm = (r_{rm} \ s_{rm} \ t_{rm} \ u_{rm})$) and from the actual model to the model in sWNF ($Bts = (r_{rs} \ s_{rs} \ t_{rs} \ u_{rs})$),

$L_{17} = Br = (P_1 \ \dots \ P_r)$ is a list of basis points of the Mordell Weil group of the actual model E/\mathbb{Q} with their Néron Tate heights.

$$L_2 = (L_{21} \ L_{22} \ L_{23} \ L_{24} \ L_{25} \ L_{26} \ L_{27})$$

where

$L_{21} = (a_1m \ a_2m \ a_3m \ a_4m \ a_6m)$ are the coefficients of the minimal model of restricted type,
 $L_{22} = (b_2m \ b_4m \ b_6m \ b_8m)$,
 $L_{23} = (c_4m \ c_6m)$,
 $L_{24} = (Dm \ FDm \ dwnm)$,
 $L_{25} = (TPm \ GTm)$,
 $L_{26} = (BTmr \ BTms)$, and
 $L_{27} = Bm$ are analogous to the lists L_{12}, \dots, L_{17} with the minimal model instead of actual model. The only difference is that the transformation parameters are from the minimal model to the actual model ($BTmr$) and from the minimal model to the model in sWNF ($BTms$). $dwnm$ in list L_{24} is the difference between the Weil height and the Néron Tate height on the minimal model of E/\mathbb{Q} .

$$L_3 = (L_{31} \ L_{32} \ L_{33} \ L_{34} \ L_{35} \ L_{36} \ L_{37} \ L_{38})$$

where

$L_{31} = (a_4s \ a_6s)$ are the coefficients of the model in sWNF of E ,
 $L_{32} = (b_2s \ b_4s \ b_6s \ b_8s)$,
 $L_{33} = (c_4s \ c_6s)$,
 $L_{34} = (Ds \ FDs \ dwns)$,
 $L_{35} = (TPs \ GTs)$,
 $L_{36} = (BTsr \ BTsm)$, and
 $L_{37} = Bs$ are analogous to the lists L_{22}, \dots, L_{27} with a model in short Weierstrass form instead of the actual model. As above, the only difference is that the transformation parameters are from the model in sWNF to the actual model ($BTsr$) and from the model in sWNF to the minimal model ($BTsm$).
 $L_{38} = (prec \ RRs)$ with the list $RRs = (e_1)$ or $RRs = (e_1 \ e_2 \ e_3)$ of the real roots of the polynomial $x^3 + a_4s \cdot x + a_6s$ and the precision (FLEPS =) $prec$, that was used for their computation. If there are three real roots, they are ordered by $e_1 < e_2 < e_3$.

$$L_4 = (L_{41} \ L_{42} \ L_{43} \ L_{44} \ L_{45} \ L_{46})$$

where

$L_{41} = (j \ Fdj)$ are the j -invariant (j) and the factorization of the denominator of j (Fdj),
 $L_{42} = (N \ FN \ Nrt \ Lcp)$ are the conductor N of E/\mathbb{Q} , the factorization of the conductor (FN), the prime numbers, where E has bad reduction together with their reduction type (Nrt) and the c_p -values Lcp ,
 $L_{43} = (oT \ ST)$ are the order (oT) and the structure (ST) of the torsion group of E/\mathbb{Q} ,
 $L_{44} = (C \ r \ Lr \ RTS \ \chi \ l_1)$ are the sign of the functional equation (C), the rank of E/\mathbb{Q} (r), the value of the r -th derivation of the L-series at 1 (Lr), the regulator of E/\mathbb{Q} (R), the order of the Tate Shafarevic group (TS), the characteristic polynomial of the regulatormatrix (χ) and the least eigenvalue of the regulatormatrix (l_1),

$L_{45} = (pw_1 \ w_1 \ pw_2 \ w_2 \ p\tau \ \tau)$ are the real period (w_1) (computed with precision $\text{FLEPS} = pw_1$), the complex period (w_2) (computed with precision $\text{FLEPS} = pw_2$) and $\tau = \pm \frac{w_1}{w_2}$ (computed with precision $\text{FLEPS} = p\tau$), such that $\text{Im}(\tau) > 0$,

$L_{46} = (Nnth \ b'_2 \ b'_4 \ b'_6 \ b'_8)$ are some constants which are used during the computation of the archimedean local Néron Tate height.

For an elliptic curve E over $K = \mathbb{Q}(\sqrt{D})$, given in long Weierstrass normal form:

List	data
L_1	contains the data of the actual model of E . Functions which use or change this data begin with ecqnfac for e(liptic) c(urve over) q(uadratic) n(umber) f(ield), a(ctual) c(urve)
L_2	<p>If a global minimal model of E exists over K, then L_2 contains the data of the global minimal model of restricted type, i.e. the coefficients a_1, \dots, a_6 of the Weierstrass form are the integral coefficients of a global minimal model for the curve over K with $a_1, a_3 \in \{0, 1\}$ if the extension of 2 in K is split or ramified, $a_1, a_3 \in \{a + b\omega \mid a, b \in \{0, 1\}\}$ if the extension of 2 in K is inert and $a_2 \in \{-1, 0, 1\}$, if the extension of 3 in K is split or ramified, $a_2 \in \{a + b\omega \mid a, b \in \{-1, 0, 1\}\}$ if the extension of 3 in K is inert. $\{1, \omega\}$ is an integral basis of the number field K, i.e. $\omega = \sqrt{D}$ if $D \equiv 2, 3 \pmod{4}$ and $\omega = \frac{1+\sqrt{D}}{2}$ if $D \equiv 1 \pmod{4}$.</p> <p>If there exists no minimal model for E over K, then L_2 contains lists (P_i, π_i, d_i, LP_i) where P_i is a list which represents a prime ideal which extends a prime number p and where the curve over K has bad reduction; π_i is a uniformizing parameter for P_i; d_i encodes the decomposition law of p and LP_i contains the data of the minimal model at P_i of restricted type.</p> <p>Functions which use or change this data begin with ecqnfmin for e(liptic) c(urve over) q(uadratic) n(umber) f(ield), min(imal) model</p>
L_3	<p>contains the data of a model in short Weierstrass normal form (sWNF). If there exists a global minimal model for E/K, then we take the sWNF as in the case of an elliptic curve over \mathbb{Q}. If there exists no global minimal model, then we use the local minimal model for the first prime ideal P_i in L_2 to find the sWNF as above.</p> <p>Functions which use or change this data begin with ecqnfsnf for e(liptic) c(urve over) q(uadratic) n(umber) f(ield), s(hort) n(ormal) f(orm)</p>
L_4	contains the invariants of E which are independent on the special model. Functions which use or change this data begin with ecqnfinv for e(liptic) c(urve over) q(uadratic) n(umber) f(ield), inv(ariants of the curve)
L_5	contains the data of the quadratic number field.

In detail, the lists L_1, \dots, L_5 are of the following form:

$$L_1 = (L_{11} \ L_{12} \ L_{13} \ L_{14} \ L_{15} \ L_{16} \ L_{17})$$

where

$L_{11} = (a_1 \ a_2 \ a_3 \ a_4 \ a_6)$ are the coefficients of the actual model of E/\mathbb{Q} ,
 $L_{12} = (b_2 \ b_4 \ b_6 \ b_8)$ and
 $L_{13} = (c_4 \ c_6)$ are the Tate values for the actual model,
 $L_{14} = (D \ FD \ ND \ FND)$ are the discriminant D , the factorization of D in prime ideals (FD), the norm of D (ND) and the factorization of the norm (FND),
 $L_{15} = (TP \ GT)$ contains a list with all torsion points of the actual model (TP) and a list of the generators of the torsion group of the actual model (GT),
 $L_{16} = (BTm \ BTs)$ or $L_{16} = ((BTm_1 \ \dots \ BTm_k) \ BTs)$ are the parameters of the birational transformation from the actual model to the minimal model (BTm) or to the P_i minimal model (BTm_i) and from the actual model to the model in sWNF (BTs),
 $L_{17} = B$ is a list of the basis of the Mordell Weil group of the actual model E/K .

If there exists a global minimal model for E/K :

$$L_2 = (L_{21} \ L_{22} \ L_{23} \ L_{24} \ L_{25} \ L_{26} \ L_{27})$$

where

$L_{21} = (a_1m \ a_2m \ a_3m \ a_4m \ a_6m)$ are the coefficients of the minimal model of restricted type,
 $L_{22} = (b_2m \ b_4m \ b_6m \ b_8m)$,
 $L_{23} = (c_4m \ c_6m)$,
 $L_{24} = (Dm \ F Dm)$,
 $L_{25} = (TPm \ GTm)$,
 $L_{26} = (BTmr \ BTms)$, and
 $L_{27} = Bm$ are analogous to the lists L_{12}, \dots, L_{17} with the minimal model instead of actual model. The only difference is that the transformation parameters are from the minimal model to the actual model ($BTmr$) and from the minimal model to the model in sWNF ($BTms$).

If there exists no global minimal model, then

$$L_2 = ((P_1 \ LP_1) \ \dots \ (P_k \ LP_k))$$

where the P_i are prime ideals and the LP_i are lists of an analogous form as the list L_2 for a global minimal model.

$$L_3 = (L_{31} \ L_{32} \ L_{33} \ L_{34} \ L_{35} \ L_{36} \ L_{37})$$

where

$L_{31} = (a_4s \ a_6s)$ are the coefficients of the model in sWNF of E ,
 $L_{32} = (b_2s \ b_4s \ b_6s \ b_8s)$,
 $L_{33} = (c_4s \ c_6s)$,
 $L_{34} = (Ds \ F Ds)$,
 $L_{35} = (TPs \ GTs)$,
 $L_{36} = (BTsr \ BTsm)$, and
 $L_{37} = Bs$ are analogous to the lists L_{22}, \dots, L_{27} with a model in short Weierstrass normal form instead of the actual model. As above, the only difference is that the transformation parameters are from the model in sWNF to the actual model ($BTsr$) and from the model in sWNF to the minimal model ($BTsm$).

$$L_4 = (L_{41} \ L_{42} \ L_{43} \ L_{44} \ L_{45})$$

where

$L_{41} = (j \ Fdj)$ are the j -invariant (j) and the factorization of the denominator of j in prime ideals (Fdj),

$L_{42} = (N \ FN \ NN \ Nrt \ Lcp)$ are the conductor N of E/K , the factorization of the conductor in prime ideals (FN), the ideal norm of N (NN), the prime ideals, where E has bad reduction together with their reduction type (Nrt) and the c_p -values Lcp ,

$L_{43} = (oT \ ST)$ are the order (oT) and the structure (ST) of the torsion group of E/K ,

$L_{44} = (C \ r \ Lr \ R \ TS)$ are the sign of the functional equation (C), the rank of E/K (r), the value of the r -th derivation of the L -series at 1 (Lr), the regulator of E/K (R) and the order of the Tate Shafarevic group (TS),

$L_{45} = (w)$ is the real period.

$$L_5 = (D \ d_4)$$

where

D is the discriminant of the field K and $d_4 = \begin{cases} 1 & \text{if } D \equiv 1 \pmod{4} \\ 0 & \text{else.} \end{cases}$

Points on elliptic curves

Points on elliptic curves are represented in coordinates

$$(x \ y \ z).$$

For $z \neq 0$, this list represents the point

$$\left(\frac{x}{z^2} \ \frac{y}{z^3} \right).$$

The point at infinity is represented by the list

$$(0 \ 1 \ 0).$$

5.3. The functions for elliptic curves. As with the packages described above, it would be impossible to give an exhaustive list of all the functions included in this package. We shall give a short description of the basic arithmetic functions for points on elliptic curves, some useful and interesting functions particular to elliptic curves and functions to test points or elliptic curves as well as the name of some related functions. A detailed description of all the elliptic curve functions can be found in the on-line documentation and by the SM keyword index.

Here are some important arithmetic functions for points on elliptic curves. The conditions on the input parameters depend on the data type; we refer you to the on-line documentation or the SM keyword index for more details.

“sum of points”

```
list eciminsum(E, P, Q)          list ecisnfsum(E, P, Q)
list ecracsum(E, P, Q)
list ecmpsum(p, a1, a2, a3, a4, a6, P, Q) list ecmpsfnfsum(p, a4, a6, P, Q)
```

with the following parameters type **int** p, a₁, a₂, a₃, a₄, a₆;
 list E, P, Q;

These functions return **P + Q**.

Some related functions: **ecgf2sum()**, **ecnfsnfsum()**, **ecnfsum()**.

“difference of points”

```
list ecimindif(E, P, Q) list ecisnfdif(E, P, Q)
list ecracdif(E, P, Q)
```

with the following parameters type **list** E, P, Q;

These functions return **P − Q**.

“inverse of points (negation)”

```
list eciminneg(E, P)          list ecisnfneg(E, P)
list ecracneg(E, P)
list ecmpneg(p, a1, a2, a3, a4, a6, P) list ecmpsfneg(p, a4, a6, P)
```

with the following parameters type **int** p, a₁, a₂, a₃, a₄, a₆;
 list E, P;

These functions return $-P$.

Some related functions: **ecgf2neg()**, **ecnfsnfneg()**, **ecnfneg()**.

“multiplication map”

```
list eciminmul(E, P, n)          list ecisnfmul(E, P, n)
list ecracmul(E, P, n)
list ecmpmul(p, a1, a2, a3, a4, a6, n, P) list ecmpsnmul(p, a4, a6, n, P)
```

with the following parameters type **single** n;
 int p, a₁, a₂, a₃, a₄, a₆;
 list E, P;

These functions return $n \cdot P$.

Some related functions: **ecgf2mul()**, **ecnfmul()**, **ecnfsnmul()**.

What follows are functions which perform tests on their arguments. There are also functions which test if a given point is a torsion point, if a list of points is linearly independent, etc. Please see the on-line documentation of the functions listed below.

is...

```
single isponecimin(E, P)  single isponecisnf(E, P)
single isponecrac(E, P)
```

with the following parameters type **list** E, P;

These functions return 1 if the point P is a point on the specific elliptic curve E ; 0 otherwise.

Some related functions: **iseciminlpld()**, **isecimintorp()**, **isineciminpl()**, **ispecrpai()**, **isppecgf2eq()**, **isppecgf2pai()**, **isppecmpeq()**, **isppecmppai()**, **isppecnfeq()**, **isppecnfpai()**.

Here is a table of some useful functions specific to elliptic curves; we give only the name of the functions related to a function category.

Function type	function names
discriminant	ecgf2disc , ecimindisc , ecisnfdisc , ecmpdisc , ecmpsnnfdisc , ecnfdisc , ecnfsnfdisc , ecqnfacdisc , ecracdisc
j-invariant	ecgf2jinv , ecmpjinv , ecmpsnnfjinv , ecnfjinv , ecnfsnnfjinv , ecqnjinv , ecrjinv
Tate's values	ecgf2tavb6 , ecgf2tavb8 , ecgf2tavc6 , ecitavalb , ecitavalc , ecmptavb6 , ecmptavb8 , ecmptavc6 , ecnftavb6 , ecnftavb8 , ecnftavc6 , ecqnfacb2 , ecqnfacb4 , ecqnfacb6 , ecqnfacb8 , ecqnfacc4 , ecqnfacc6 , ecrtavalb , ecrtavalc
Tate's algorithm	ecimintate , ecitatealg , ecqnftatealg
reduction type	eciminbrtmp , eciminmrtmp , eciminrt , ecqnflstrt , ecrrt
conductor	ecqnfcond , ecqnflcond , ecqnfncnd , ecqnfpicond , ecrcond , ecrfcond
regulator	eciminreg , ecrregulator , ecqnfreg
sign	ecrsign
L-series	ecrlser , ecrfelser , ecrlser , ecrlserfd , ecrlserhd , ecrlserrkd
rank	ecrrank , ecrrank2d , ecrrankbsd , ecrrankg2d , ecqnflrank
basis	eciminbmwg , ecisnfbmwg , ecracbmwg , ecrmaninalg
integral points	ecimeqsaSip , eciminsaip , ecisnfsaip , ecisnfsaipub , ecracsaiip

Function type	function names
torsion group	ecimintorgr, ecisnftorgr, ecractorgr
number of points	ecgf2npfe, ecmspnnp, ecmspsnfnp
height	eciminlhaav, eciminlhnaav, eciminnetahem, ecracweilhe, ecqnflhaav, ecqnflhnaav, ecqnfnetahе, ecqnfnetapa, ecqnfwehe

Finally, the elliptic curve package contains functions which perform birational transformation with given transformation parameters and birational transformation from one model to another (see table).

From	to
actual model	minimal model model in sWNF
minimal model	actual model model in sWNF
model in short Weierstrass normal form (sWNF)	actual model minimal model

6. The NON-SIMATH package

To increase the performance of basic arithmetic operations, SIMATH includes some packages, which do not use the SIMATH list system. These packages are

- The **Essen** arithmetic package for positive integers; see the documentation of **Earith()** for details)
- The **Heidelberg** arithmetic package for integers. See the on-line documentation of **HDiadd()**, **HDiutil()**, **HDimul()** etc.
- The **Papanikolaou** floating point package. See the documentation for **PAFadd()** etc. in the on-line documentation or look for Papanikolaou in the keyword index.

A comfortable way to use functions of the Heidelberg and Papanikolaou package with SIMATH variables is given by the functions **iHDFu()**, **fPAFfu()** and **fltrig()**.

CHAPTER 5

The calculator **simcalc**

1. What is **simcalc**?

The SIMath CALCulator is an interactive system which evaluates simple to highly complex mathematical expressions using operators and SIMATH functions. The user can also edit input lines, store computational results in variables, write programs using the various **simcalc** program control statements, etc. We will describe **simcalc**'s utility and mathematical functions in the following sections.

The command **simcalc** opens a session with **simcalc**. After the prompt symbol **>**, **simcalc** expects an input of the form

> expression ↵

or

> variable = expression ↵ .

Several commands separated by “_” or “;” can be entered on one line (for “;” see section 5.16); the input line can be longer than the length of a screen line.

The input is passed to **simcalc** by pressing RETURN (↵).

The command

>LINES = *n* ↵


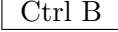
where *n* is a nonnegative single-precision number, changes the line length of the output to *n* characters. The default value of **LINES** is the global SIMATH parameter **LN_SIZE**.

The commands **exit**, **quit**, or **CTRL/D** terminate a **simcalc** session.

simcalc uses the **GNU** readline library to enable emacs style command line editing. Please refer to the appropriate manuals for a description on how to edit input lines. We describe some commands in §5.2.

2. Editing command lines

2.1. Moving the cursor. The following keys are used to move the cursor:

 , 

 , 

The cursor moves one space to the left or to the right. It can be moved only within the current command line.

2.2. Inserting and deleting characters. Characters are inserted simply by typing them at the position of the cursor; characters to the right of the cursor will be moved to the right.

The following keys are used to delete characters:



The character to the left of the cursor is erased and the cursor is moved one space backward. All the characters to the right of the erased character are moved one space backward.



The character at the cursor is erased and all the characters to the right of this character are moved one space backward.

2.3. The Ctrl P command. The **Ctrl P** command lets you edit previous input lines. If you enter

>**Ctrl P** ↵

the last instruction appears on the screen. This instruction can also be edited.

3. Example session

We now give an example session with **simcalc**; you might want to do it yourself to get an idea of **simcalc**. For this example session, the output of the computer will be shown in typewriter style and your input in bold face; we use % as the shell prompt.

After your shell's prompt symbol, type in

```
% simcalc ↵
```

The introductory screen of **simcalc** will appear on your screen. You can type in

```
>?help ↵
```

to get general information about **simcalc**. Use the space key to scroll until the **simcalc** prompt symbol > appears or press **q** to quit the description.

We now want to “log” the session into a log file named *test*

```
>logon(test) ↵  
Logging to file test.  
>
```

All the following computations will be written into the log file. In the course of the session, you will be able to look up what you have computed since opening the log file. Proceed as follows:

```
>k=1+2 ↵  
k = 3  
>k-7 ↵  
@ = -4  
>
```

If you do not explicitly assign an expression to a variable (as above), its value will automatically be assigned to the system variable @. Check its current contents:

```
>@ ↵  
@ = -4  
>
```

You can enter two expressions on one line separated by `_` :

```
>y=1+2_z=k-7 ↵
  y = 1+2
    = 3
  z = k-7
    = -4
>
```

If you want to assign a new value to a variable that is already used (e.g. `z := 9`), type in

```
>z=9 ↵

z is already used.
overwrite? (y/n) y ↵
  z = 9
>k=100!/(10!*90!) ↵

k is already used.
overwrite? (y/n) n ↵

new variable name: h ↵
  h = 17310309456440
>
```

You can look at the contents of the log file by typing in

```
>loglist ↵
```


log file of Fri Sep 22 11:04:46 1992
=====

```
in : k=1+2
out: k = 3

in : k-7
out: @ = -4

in : @
out: @ = -4

in : y=1+2_z=k-7
out: y = 1+2
      = 3

out: z = k-7
      = -4

in : z=9
out: z is already used.
      overwrite? (y/n)
in : y
out: z = 9

in : k=100!/(10!*90!)
out: k is already used.
      overwrite? (y/n)
in : n
out: new variable name :
in : h
out: h = 17310309456440

in : loglist
>
```

You can close the log file *test* as follows

```
>logoff ↵
Log file test closed.
```

Now use a **simcalc** function, e.g. the greatest common divisor

```
>gcd(12, 15, 18) ↵
      @ = 3
>gcd(5)
```

```
***** Error:  Illegal number of arguments!
>
```

If you do not know why an error was reported, **simcalc** will give you a detailed description of the function **gcd()** when you type in

```
>?gcd ↵
```

```

* * * * *
*   gcd   *
* * * * *

```

```
"greatest common divisor"
SYNTAX: X = gcd(A1, A2, ..., An)
```

```
A1, ..., An are expressions whose values are elements in Z,
polynomials over Z, Z/pZ, GF(p^m), n>=2, (where p is a prime),
or over number fields.
```

```
X is assigned the greatest common divisor of A1, ..., An.
```

```
Warning: If p > 2^30, the primality of p is not tested.
```

```
Example 1: (correct)
```

```
gcd(a1, (2*5-75), 234 + 23)
```

```
Example 2: (incorrect)
```

```
gcd(2/3,14)
```

```
Error message:
Illegal parameter!
```

```
Example 3: (correct)
```

```
gcd(x^2 + 2*x + 1, x + 1)
```

```
Example 4: (correct)
```

```
gcd(MOD(x^2 + 2*x + 1), x + MOD(1))
```

An error was reported since the function requires at least two arguments.

The function **tofl()** (“to floating points”) returns the floating point representation of integer, rational, or floating point numbers or polynomials over these structures rounded to **DIGITS** digits. The system variable **DIGITS** gives the number of decimal places and internal computation precision for floating point and complex numbers.

```
>DIGITS = 40 ↵
      DIGITS = 40
>tofl(1/7) ↵
      @ = 0.1428571428571428571428571428571428571429e0
>
```

simcalc will list the variables currently used and their current contents when you type in

```
>?? ↵
variables over Z:
      h = 17310309456440
      k = 3
      y = 3
      z = 9

variables over R:
      @ = 0.1428571428571428571428571428571428571429e0

global variable:
      DEPTH = 30
      DIGITS = 40
      HEIGHTBOUND = 0.11e2
      LINELN = 79
      OUTPUTDIG = 0
>
```

End the example session by entering the command

```
>exit ↵
```

B Y E

```
*** # GCs: 0          GC time: 0.00 s      # collected cells: 0      ***
*** # blocks: 1       block size: 16383    # free cells:      15055   ***
*** total CPU time: 15.33 s                ***
```

4. Operators and functions

In **simcalc**, an *expression* is any combination of constant, variable, and operator. **simcalc** can evaluate single expressions, a sequence of expressions connected by `_` or `;` (see §5.3 or §5.15), and functions which can be passed expressions and functions as arguments. It is also possible to program in **simcalc** by using control statements or user defined functions (see §5.4.4).

From this point on, we will use the following notation

p	prime number of \mathbb{Z}
$\mathbb{Z}/m\mathbb{Z}$	residue class ring modulo m
Nf	number field
Gf	Galois field
$P(\mathbb{Z}/m\mathbb{Z})$	polynomial ring $\mathbb{Z}/m\mathbb{Z}[X_1, \dots, X_n]$
$P(\mathbb{Z})$	polynomial ring $\mathbb{Z}[X_1, \dots, X_n]$
$P(\mathbb{Q})$	polynomial ring $\mathbb{Q}[X_1, \dots, X_n]$
$P(\mathbb{R})$	polynomial ring $\mathbb{R}[X_1, \dots, X_n]$
$P(\mathbb{C})$	polynomial ring $\mathbb{C}[X_1, \dots, X_n]$
$P(\text{Nf})$	polynomial ring $\text{Nf}[X_1, \dots, X_n]$
$P(\text{Gf})$	polynomial ring $\text{Gf}[X_1, \dots, X_n]$
$F(\mathbb{Q})$	function field $\mathbb{Q}(X_1, \dots, X_n)$

Note: for any variable A , we have

$$A \in \mathbb{C} \Rightarrow A \in \mathbb{R} \Rightarrow A \in \mathbb{Q} \Rightarrow A \in \mathbb{Z}$$

$$A \in P(\mathbb{C}) \Rightarrow A \in P(\mathbb{R}) \Rightarrow A \in P(\mathbb{Q}) \Rightarrow A \in P(\mathbb{Z})$$

“polynomial” (denoted $P()$) refers to polynomials over any of $\mathbb{Z}/m\mathbb{Z}$, \mathbb{C} , the current number field or the current Galois field.

“matrix” and “vector” refer to matrices and vectors (respectively) over any of $\mathbb{Z}/m\mathbb{Z}$, \mathbb{Q} , Nf, Gf, $P()$, or $F(\mathbb{Q})$.

Polynomials, matrices, and vectors are also called “structures”; e.g. a structure over \mathbb{Z} is either a polynomial over \mathbb{Z} , a matrix, or a vector over \mathbb{Z} , or a matrix or vector over $P(\mathbb{Z})$.

Remark: **simcalc** reads vectors as

$$\{\mathbf{a}_1, \dots, \mathbf{a}_n\},$$

and $m \times n$ -matrices as

$$\{\{\mathbf{a}_{11}, \dots, \mathbf{a}_{1m}\}\{\mathbf{a}_{21}, \dots, \mathbf{a}_{2m}\} \dots \{\mathbf{a}_{n1}, \dots, \mathbf{a}_{nm}\}\}.$$

4.1. Operators. The following table lists the precedence of the **simcalc** operators ; operators with equal priority are left associative.

hierarchy
[]
^
– (negation) !
* / : mod
+ –
< <= > >=
== !=
&&

op.	form	arguments	result
+	$A + B$	$A, B \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf}, P(), F(\mathbb{Q})$ A, B points on an elliptic curve; A, B matrices; A, B vectors	sum of A and B
–	$A - B$	$A, B \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf}, P(), F(\mathbb{Q})$ A, B points on an elliptic curve; A, B matrices; A, B vectors	difference of A and B
–	$-A$	$A \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf}, P(), F(\mathbb{Q})$ A point on an elliptic curve; A matrix; A vector	additive inverse of A

op.	form	arguments	result
*	$A * B$	$A, B \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf}, P(), F(\mathbb{Q})$	product of A and B
		A, B matrices	
		A matrix; B vector	matrix-vector multiplication of A by B
		A matrix or vector; B scalar	scalar multiplication of A by B
/	A/B	$A, B \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf}, P(\mathbb{Q}), P(\mathbb{Z}/m\mathbb{Z}), P(\text{Nf}), P(\text{Gf}), F(\mathbb{Q});$ $B \neq 0;$	quotient of A by B
		A, B matrices; B invertible	
^	A^B	$A \in \mathbb{R}; B \in \mathbb{R}$	A to the power B
		$A \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C};$ $B \in \mathbb{Q}, B$ of the form $S/2^t, S \in \mathbb{Z}$	
		$A \in \text{Nf}, \text{Gf}, P(\mathbb{Q}), F(\mathbb{Q});$ $B \in \mathbb{Z}, B < \mathbf{BASIS}$	
		$A \in P(\mathbb{Z}/m\mathbb{Z}), P(\text{Nf}), P(\text{Gf}), P(\mathbb{C});$ $B \in \mathbb{N}_0, B < \mathbf{BASIS}$	
		A square matrix; $B \in \mathbb{Z};$ $ B < \mathbf{BASIS};$ if $B < 0$, A invertible	
[]	[A]	$A \in \mathbb{C}$	absolute value of A
:	$A:B$	$A, B \in \mathbb{Z}, P(\mathbb{Q}), P(\mathbb{Z}/m\mathbb{Z}), P(\text{Nf}), P(\text{Gf});$ $B \neq 0$	the integer part of the quotient
mod	$A \bmod B$	$A \in \mathbb{Q}, P(\mathbb{Q}); B \in \mathbb{N}, P(\mathbb{Q});$	A modulo B
		$A, B \in P(\mathbb{Z}/p\mathbb{Z}), P(\text{Nf}), P(\text{Gf});$	
		A matrix or vector over $\mathbb{Q}, P(\mathbb{Q});$ $B \in \mathbb{N}, P(\mathbb{Q});$	A modulo B (componentwise)
		A matrix or vector over $P(\mathbb{Z}/p\mathbb{Z}), P(\text{Nf}), P(\text{Gf});$ $B \in P(\mathbb{Z}/p\mathbb{Z}), P(\text{Nf}), P(\text{Gf});$	
!	$A!$	$A \in \mathbb{N}_0, A < \mathbf{BASIS}$	factorial of A
<	$A < B$	$A, B \in \mathbb{R}, \text{curModulus}$	comparison of A and B
<=	$A \leq B$		
>	$A > B$		
>=	$A \geq B$		
==	$A == B$	$A, B \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf}, P(), F(\mathbb{Q})$ matrices, vectors, elliptic curves, points;	comparison of A and B
	$A \neq B$		
&&	$A \&\& B$	$A, B \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf};$	A and B
	$A \parallel B$	$A, B \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}, \text{Nf}, \text{Gf};$	A or B

For details see ‘‘?op’’.

4.2. Standard functions.

$\arccos(A)$	$A \in \mathbb{R}$ arccos returns the arc cosine of A
$\operatorname{arccot}(A)$	$A \in \mathbb{R}$ arccot returns the arc cotangent of A
$\operatorname{arcosh}(A)$	$A \in \mathbb{R}$ arcosh returns the arc hyperbolic cosine of A
$\operatorname{arcoth}(A)$	$A \in \mathbb{R}$ arcoth returns the arc hyperbolic cotangent of A
$\arcsin(A)$	$A \in \mathbb{R}$ arcsin returns the arc sine of A
$\arctan(A)$	$A \in \mathbb{R}$ arctan returns the arc tangent of A
$\operatorname{arsinh}(A)$	$A \in \mathbb{R}$ arsinh returns the arc hyperbolic sine of A
$\operatorname{artanh}(A)$	$A \in \mathbb{R}$ artanh returns the arc hyperbolic tangent of A
$\operatorname{aval}(m, A)$	$m \in \mathbb{N}$; $A \in \mathbb{Q}$ with $A \neq 0$. aval returns the additive m -adic value of A .
$\operatorname{binom}(A, B)$	$A, B \in \mathbb{N}_0$ with $0 \leq B \leq A$. binom returns the binomial coefficient of A over B .
$\operatorname{chcoef}(P, X, e, Q)$	P is a polynomial; X is a variable of P ; $e \in \mathbb{Z}$ with $ e < \mathbf{BASIS}$; Q is a number or a polynomial. chcoef replaces the coefficient of X^e of P by Q .
$\operatorname{chinrem}(A_1, M_1, \dots, A_n, M_n)$	$A_1, \dots, A_n \in \mathbb{Z}, P(\mathbb{Z})$; $M_1, \dots, M_n \in \mathbb{N}$ and pairwise coprime. chinrem returns the solution of the simultaneous congruences $B \equiv A_i \pmod{M_i}$, $1 \leq i \leq n$, using the Chinese remainder theorem.
$\operatorname{chpol}(A, X)$	A is a square matrix; X is a variable (not in A). chpol returns the characteristic polynomial of A in X .
$\operatorname{coef}(P, X, e)$	P is a polynomial; X is a variable of P ; $e \in \mathbb{Z}$ with $ e < \mathbf{BASIS}$. coef returns the coefficient of X^e of P .
$\operatorname{conjug}(A)$	$A \in \mathbb{C}$ or a quadratic number field. conjug returns the conjugate of A .
$\operatorname{cont}(P, X)$	$P \in P(\mathbb{Z})$, X is a variable which occurs in P . cont returns the content of P with respect to the variable X .
$\cos(A)$	$A \in \mathbb{R}$ cos returns the cosine of A .
$\cosh(A)$	$A \in \mathbb{R}$ cosh returns the hyperbolic cosine of A .

cot (A)	$A \in \mathbb{R}$ cot returns the cotangent of A .
coth (A)	$A \in \mathbb{R}$ coth returns the hyperbolic cotangent of A .
curgf (p, Y, Q)	p is a single-precision prime number; Y is a variable; Q is a univariate irreducible polynomial in Y over $\mathbb{Z}/p\mathbb{Z}$ or \mathbb{Z} of degree > 1 or an integer > 1 curgf specifies the current Galois field $\text{Gf}(p^n)$ where n is the degree of Q or Q respectively; the polynomial associated with $\text{GF}(p^n)$ is returned. The current Galois field remains valid until the next call of curgf or curgf2bit .
curgf2bit (Y, S)	Y is a variable; S is a sequence of 0 and 1, which represents a univariate irreducible polynomial Q in Y over $\mathbb{Z}/2\mathbb{Z}$ of degree > 1 curgf2bit specifies the current Galois field $\text{Gf}(2^n)$ where n is the degree of Q ; the polynomial associated with $\text{GF}(2^n)$ is returned. The current Galois field remains valid until the next call of curgf or curgf2bit .
curmod (m)	m is an integer > 1 . curmod specifies the current modulus m ; it remains valid until the next call of curmod .
curnf (Y, P)	Y is a variable, P is a univariate irreducible polynomial in Y over $\mathbb{Z}/m\mathbb{Z}$ or \mathbb{Z} of degree > 1 . curnf specifies the current number field $K = \mathbb{Q}(\alpha)$ where α is a root of P ; it remains valid until the next call of curnf .
declaw (p)	p is a prime number. declaw computes the decomposition law of p in the current number field.
deg (P, Y) deg (P)	P is a polynomial. Y is a variable. deg computes the degree of P with respect to the variable Y or the main variable.
denom (A)	$A \in \mathbb{Q}, F(\mathbb{Q})$, or structure over \mathbb{Q} If $A \in \mathbb{Z}$ or structure over \mathbb{Z} , denom returns 1. If $A \in \mathbb{Q}, F(\mathbb{Q})$, denom returns the denominator of A ; otherwise denom returns the least common denominator of the coefficients or entries of A .
deriv (A, X_1, \dots, X_r)	$A \in P(), F(\mathbb{Q})$; X_1, \dots, X_r are variables. deriv returns the derivatives of A with respect to X_1, \dots, X_r successively.
det (A)	A is a square matrix. det returns the determinant of A .
disc (A, X, N)	$A \in P(\mathbb{Z}/p\mathbb{Z}), P(\mathbb{Z})$; X is a variable of P ; N is 1, 2, 3, or 4 and specifies the algorithm. disc returns the discriminant of A with respect to X .

discnf ()	discnf returns the discriminant of the current number field and displays its factorization.
dmconstr (n, A)	$n \in \mathbb{N}$; $A \in \mathbb{Q}, \mathbb{Z}/m\mathbb{Z}, \text{Nf}, \text{Gf}$, or a polynomial or a rational function. dmconstr returns a $n \times n$ -diagonal matrix with A on the diagonal.
egcd (A_1, \dots, A_n)	$A_1, \dots, A_n \in \mathbb{Z}, P(\mathbb{Z}/p\mathbb{Z})$, or $P(\text{Gf})$. egcd returns the greatest common divisor of A_1, \dots, A_n and computes the cofactors.
eival (A, X)	A is a square matrix over $\mathbb{Z}/p\mathbb{Z}, \mathbb{Q}$, or Gf ; X is a variable. eival computes the eigenvalues of A and the irreducible factors of degree > 1 of the characteristic polynomial; the characteristic polynomial divided by its linear factors is returned.
elform (A, col, cor)	A is a nonzero matrix over $\mathbb{Z}, P(\mathbb{Z}/p\mathbb{Z})$, or $P(\mathbb{Q})$; cor and col are 0 or 1. elform returns the elementary divisor form of A and displays the left cofactor if col is 1, and the right cofactor if cor is 1.
elsubst (M, el, pz, ps) elsubst (V, el, pos)	M is a matrix; V is a vector; $el \in \mathbb{Z}/m\mathbb{Z}, \mathbb{Q}, \text{Nf}, \text{Gf}, P(), F(\mathbb{Q})$; pz, ps , and pos are positive integers. elsubst replaces the element in position (pz, ps) in M (the element in position pos in V) by el .
exp (A)	$A \in \mathbb{C}$. exp returns e^A .
fact (A)	$A \in \mathbb{Q}, P(\mathbb{Z}/p\mathbb{Z}), P(\mathbb{Q}), P(\text{Nf}), P(\text{Gf}), F(\mathbb{Q})$, or quadratic number field; $A \neq 0$. fact computes the list of prime factors of A , displays the prime factorization and returns the largest prime factor. If A is a rational number or a rational function, the numerator and denominator are factorized and displayed separately and the largest prime factor of the numerator is returned. If A is an algebraic number, fact computes and displays the prime ideal factorization of the principal ideal.
floor (A)	$A \in \mathbb{R}$. floor returns the largest integer $\leq A$.
ftoi (A)	$A \in \mathbb{R}, P(\mathbb{R})$. ftoi returns the value of A over \mathbb{Z} , if possible.
froot (p, A) froot (B)	$A \in P(\mathbb{Z})$; $B \in P(\mathbb{Z}/p\mathbb{Z}), P(\text{Gf}), P(\mathbb{R})$. froot returns a root of $A \bmod p$ or of B and computes all roots.
gcd (A_1, \dots, A_n)	$A_1, \dots, A_n \in \mathbb{Z}, P(\mathbb{Z}/p\mathbb{Z}), P(\mathbb{Z}), P(\text{Nf})$, or $P(\text{Gf})$. gcd returns the greatest common divisor of A_1, \dots, A_n .

getb (S)	S is a sequence of 0 and 1, which is the bit representation of an integer. getb returns the integer represented by S .
gethex (S)	S is a sequence of 0, 1, ..., 9, $A, B, \dots, F, a, b, \dots, f$, which is the hexadecimal representation of an integer. gethex returns the integer represented by S .
getoct (S)	S is a sequence of 0, 1, ..., 7, which is the octal representation of an integer. getoct returns the integer represented by S .
gftopol (A)	A is an element or a structure over Gf . gftopol transforms A into an element or structure over $P(\mathbb{Z})$ in the variable specified by curgf .
herm (A, np)	A is a regular square matrix over \mathbb{Z} ; np is 1 or -1 . herm returns the Hermite normal form of A . If np is 1, the entries below the main diagonal are ≥ 0 ; otherwise, they are ≤ 0 .
imag (A)	$A \in \mathbb{C}$ imag returns the imaginary part of A .
intbas ()	intbas computes the integral basis of the current number field and returns the index of the polynomial order in the maximal order. The factorization of the index is displayed on the screen.
integr ($P, X_1, L_1, U_1, \dots, X_n, L_n, U_n$)	$P \in P(\mathbb{Q})$; X_1, \dots, X_n are variables; L_1, \dots, L_n and $U_1, \dots, U_n \in \mathbb{Q}, P(\mathbb{Q})$, or $F(\mathbb{Q})$. integr returns the integral of P with respect to the variables X_1, \dots, X_n with corresponding lower bounds L_1, \dots, L_n and upper bounds U_1, \dots, U_n .
irpgen (p, Y, D, N) irpgen (Y, D, N)	Y is a variable; $D \in \mathbb{N}$; $N \in \mathbb{N}_0$. irpgen returns a univariate irreducible polynomial in Y of degree D over $\mathbb{Z}/p\mathbb{Z}$, where in the second case p is the current modulus. N specifies the number of coefficients $\neq 0$.
isprime (A)	$A \in \mathbb{N}$ isprime test A for primality.
jacsym (A, M)	$A, M \in \mathbb{Z}$; A and M coprime, $M > 0$ and odd. jacsym returns the Jacobi symbol (A / M) .
lcm (A_1, \dots, A_n)	$A_1, \dots, A_n \in \mathbb{Z}, P(\mathbb{Z}/p\mathbb{Z}), P(\mathbb{Z}), P(\text{Nf}),$ or $P(\text{Gf})$. lcm returns the least common multiple of A_1, \dots, A_n .
linequ (A, b)	A is an $(m \times n)$ -matrix over $\mathbb{Z}/p\mathbb{Z}, \mathbb{Q}, \text{Nf}, \text{Gf}, P(\mathbb{Q}), F(\mathbb{Q})$; b is a vector of length m over $\mathbb{Z}/p\mathbb{Z}, \mathbb{Q}, \text{Nf}, \text{Gf}, P(\mathbb{Q}), F(\mathbb{Q})$; linequ returns the solution to the system of linear equations $A \cdot X = b$.

ln (A)	$A \in \mathbb{R}, A > 0$. ln returns the natural logarithm of A .
log2 (A)	$A \in \mathbb{R}, A > 0$. log2 returns the integer part of the base 2 logarithm of A .
log10 (A)	$A \in \mathbb{Z}$ log10 returns 1+ the integer part of the base 10 logarithm of A , i.e. the number of digits of A .
mam2 (n, l_1, \dots, l_m)	n is a positive single-precision integer, l_1, \dots, l_m are integers with $0 \leq l_i \leq 2^n - 1$. mam2 returns the matrix which is represented by these numbers in special bit representation (see §5.6).
mcoef (A, l, k)	A is a matrix; $l, k \in \mathbb{N}$. mcoef extracts the $l \times k$ -th coefficient of A .
mex (M, A, E)	$M \in \mathbb{N}, A \in \mathbb{Q}, E \in \mathbb{Z}$; the denominator of A is relatively prime to M . mex returns $(A^E) \bmod M$.
minor (A, l, k) minor ($A, R, C, r_1, \dots, \dots, r_R, c_1, \dots, c_C$)	A is a matrix; $l, k, R, C, r_1, \dots, r_R, c_1, \dots, c_C \in \mathbb{N}_0$. minor returns the matrix which is derived from A by deleting the l -th row and the k -th column or the r_k -th rows and the c_l -th columns.
mtoi (A)	A is an element, a structure, an elliptic curve or a point over $\mathbb{Z}/m\mathbb{Z}$. mtoi transforms A into an element or structure over \mathbb{Z} .
mval (m, A)	$m \in \mathbb{N}; A \in \mathbb{Q}, A \neq 0$. mval returns the multiplicative m -adic value of A .
nftopol (A)	A is an element or a structure over \mathbb{Nf} . nftopol transforms A into an element or structure over $P(\mathbb{Z})$ in the variable specified by curnf .
norm (A)	$A \in \mathbb{Nf}$. norm returns the norm of A over \mathbb{Q} .
num (A)	$A \in \mathbb{Q}, F(\mathbb{Q})$, or structure over \mathbb{Q} num returns $A * \mathbf{denom}(A)$, i.e. the numerator of A .
ord (m, A) ord (B)	$m \in \mathbb{N}; A \in \mathbb{Z}; B \in \mathbb{Z}/m\mathbb{Z}$. ord returns the order of A or B in the multiplicative group of $\mathbb{Z}/m\mathbb{Z}$.
pconstr (A, n)	$A \in \mathbb{N}; n$ is 0, 1, or 2 and specifies the algorithm. pconstr returns a prime $p > A$ and displays the factorization of $p - 1$.
pfact (p, A)	p is a prime number; $A \in P(\mathbb{Q})$ (univariate). pfact factorizes A as a polynomial over $\mathbb{Z}/p\mathbb{Z}$, displays the factorization and returns the prime factor of highest degree.
pgen (U, O)	$U, O \in \mathbb{N}_0, U \leq O$. pgen computes the list of primes between U and O and returns the smallest one.

phi (A)	$A \in \mathbb{N}$. phi returns the value of the Euler φ -function of A .
prfunc (A, X_1, \dots, X_r)	$A \in P(\mathbb{Q})$; X_1, \dots, X_r are variables. prfunc returns the primitive function of A with respect to X_1, X_2, \dots, X_r .
prim (P, X)	$P \in P(\mathbb{Z})$, X is a variable which occurs in P . prim returns the primitive part of P with respect to the variable X .
pripgen (X, n, j)	X is a variable, n and j are single-precision integers with $n > 1$ and $n + 2 > j > 2$. pripgen returns a monic univariate primitive polynomial P of degree n in the variable X over a specified Galoisfield of characteristic 2. P has j non vanishing coefficients and $P(0) \neq 0$.
print (A_1, \dots, A_n)	A_1, \dots, A_n are strings or simcalc expressions. If the input is a string, print displays it as it is. If the input is a simcalc expression, print displays its value.
proot (A) proot ()	$A \in \mathbb{N}$ or the current modulus. proot returns a primitive root of $\mathbb{Z}/A\mathbb{Z}$.
putb (A)	$A \in \mathbb{N}$. putb displays the bit representation of A and returns the value of A .
putbits (A)	A may be any SIMATH type variable . If $A \in \mathbb{Z}$, $ A < \mathbf{BASIS}$, putbits displays the bit representation of A . Otherwise, putbits displays the bit representation of the address of A .
putGF2bit (A)	A is an element of $\text{Gf}(2^n)$. putGF2bit displays the bit representation of A , interpreted as a polynomial over $\mathbb{Z}/2\mathbb{Z}$.
puthex (A)	$A \in \mathbb{N}$. puthex displays the hexadecimal representation of A and returns the value of A .
puto (A)	A may be any SIMATH type variable . puto displays the internal representation of A .
putoct (A)	$A \in \mathbb{N}$. putoct displays the octal representation of A and returns the value of A .
real (A)	$A \in \mathbb{C}$ real returns the real part of A .
relcln (p^n, A)	$p \neq 2$; $A \mid \varphi(p^n)$. relcln returns the relative class number of the abelian number field of conductor p^n and degree A .

resul (A, B, X, N)	$A, B \in P(\mathbb{Z}/p\mathbb{Z}), P(\mathbb{Z}), P(\text{Gf})$; X is a variable; N is 1, 2, 3, or 4 and specifies the algorithm. resul returns the resultant of A and B with respect to X .
rk (A)	A is a matrix. rk returns the rank of A .
root (A, n)	$A \in \mathbb{Z}/m\mathbb{Z}, \mathbb{R}, \mathbb{C}$; $n \in \mathbb{Z}, n < \mathbf{BASIS}$. If $A \in \mathbb{Z}/m\mathbb{Z}, \mathbb{C}$, then n must be a power of 2. root returns the n^{th} root of A .
scalar (A, B)	A and B are vectors of the same length. scalar returns the euclidian scalar product of A and B .
sfp (A)	$A \in \mathbb{Q}, P(\mathbb{Q}), P(\mathbb{Z}/p\mathbb{Z}), P(\text{Nf}), P(\text{Gf}),$ or $F(\mathbb{Q})$. sfp returns the square free part of A .
sgn (A)	$A \in \mathbb{R}$. sgn returns the sign of A .
sin (A)	$A \in \mathbb{R}$. sin returns the sine of A .
sinh (A)	$A \in \mathbb{R}$. sinh returns the hyperbolic sine of A .
sort ($A, X_{p(1)}, \dots, X_{p(r)}$)	A is a polynomial in the variables $X_{p(1)}, \dots, X_{p(r)}$. sort returns A sorted with respect to the specified order of variables; A is unchanged.
sqrt (A)	$A \in \mathbb{R}, \mathbb{Z}/m\mathbb{Z}, \mathbb{C}$. sqrt returns the square root of A .
tan (A)	$A \in \mathbb{R}$. tan returns the tangent of A .
tanh (A)	$A \in \mathbb{R}$. tanh returns the hyperbolic tangent of A .
tofl (A)	$A \in \mathbb{R}, P(\mathbb{R})$. tofl returns the floating point representation of A rounded to DIGITS digits (see §5.4.4).
trace (A) trace (M)	$A \in \text{Nf}$; M is a square matrix. trace returns the trace of A over \mathbb{Q} or the trace of M .
transp (A)	A is a matrix. transp returns the transpose of A .
unit ()	The current number field must be quadratic. unit computes the unit group of the current number field. For imaginary quadratic fields, the generator is returned; in the real quadratic case, the fundamental unit is returned.
vcoef (A, l)	A is a vector; $l \in \mathbb{N}$. vcoef extracts the l -th coefficient of A .

zmconstr (m, n)	$m, n \in \mathbb{N}$. zmconstr constructs the $m \times n$ -zeromatrix.
zvconstr (n)	$n \in \mathbb{N}$. zvconstr constructs the zerovector of length n .

For details see ‘‘?functionname’’.

4.3. Functions for elliptic curves and their points. In the following, “elliptic curves” and their “points” refer to elliptic curves and points over $\mathbb{Z}/p\mathbb{Z}$, \mathbb{Q} , $\text{Gf}(2^n)$, or the current number field. Otherwise, the field over which the elliptic curve is defined is mentioned explicitly.

For any points on the current elliptic curve, addition is done with the “+” operator, subtraction and negation with the “−” operator (see §5.4.1 and §5.9).

aftopr (P)	P is an affine point on an elliptic curve. aftopr returns the corresponding projective representation of P .
areptsli (P_1, \dots, P_n)	P_i are points on the current elliptic curve over \mathbb{Q} or over quadratic number fields. areptsli computes the determinant of the matrix of Néron-Tate pairings and decides if the points are linear independent or not.
basismwg (E)	E is an elliptic curve over \mathbb{Q} . basismwg computes the basis of the Mordell-Weil group and returns the rank of the Mordell-Weil group of E .
bitrans (E, r, s, t, u) bitrans (P, r, s, t, u)	E is an elliptic curve; P is a point on an elliptic curve; r, s, t , and u are elements of the field over which the elliptic curve is defined, $u \neq 0$. bitrans computes the birational transformation of E or P by r, s, t , and u and returns the transformed curve or point.
cond (E)	E is an elliptic curve over \mathbb{Q} or a quadratic number field. cond returns the conductor of the global minimal model of E .
cper (E)	E is an elliptic curve over \mathbb{Q} . cper returns the complex period of E .
curec (E)	E is an elliptic curve. curec specifies the current elliptic curve. The group law is defined with respect to this curve. The current elliptic curve remains valid until the next call to curec .
derivL (n, E)	$n \in \mathbb{N}_0$; E is an elliptic curve over \mathbb{Q} . derivL returns the n^{th} derivative of the L -series $L(E, s)$ at $s = 1$.
discec (E)	E is an elliptic curve. discec returns the discriminant of E .
ecccoef (E, n)	E is an elliptic curve; $n = 1, 2, 3, 4$, or 6 . ecccoef returns the n -th coefficient of E .

ecgnp (p, n, D) ecgnp (n, D)	p is a prime > 3 ; in the second case, a prime $p > 3$ must be specified by curmod . If $n \in \mathbb{N}_0$, $D = 0$ or $D = (p + 1 - n)^2 - 4 \cdot p$; n and D must not be both 0. ecgnp returns an elliptic curve of n points over $\mathbb{Z}/p\mathbb{Z}$ or \mathbb{Q} respectively.
ecinf (E)	E is an elliptic curve. Depending on which field E is defined, ecinf displays Tate's values, discriminant (and its factorization), j -invariant (and the factorization of its denominator), a global minimal model and the parameters of the transformation, the conductor and its factorization, the Kodaira and Néron symbols for the primes of bad reduction.
faintp (E)	E is an elliptic curve over \mathbb{Q} . faintp computes all integral points of E .
fp (E)	E is an elliptic curve over $\mathbb{Z}/p\mathbb{Z}$ or $\text{GF}(2^n)$. fp returns any projective point of E which is not the point at infinity.
intcoef (E)	E is an elliptic curve over \mathbb{Q} or a quadratic number field. intcoef returns a birational isomorphic elliptic curve with coefficients in the integral domain.
isptec (E, P)	E is an elliptic curve; P is a point on an elliptic curve. isptec tests whether P lies on E or not.
jinv (E)	E is an elliptic curve. jinv returns the j -invariant of E
minim (E, s)	E is an elliptic curve over \mathbb{Q} ; s is 0 or 1. minim returns the birational isomorphic global minimal elliptic curve of restricted type. If $s = 1$, the transformation parameters r, s, t , and u are displayed.
np (p, A) np (E)	p is a prime number; A is an elliptic curve over \mathbb{Q} with good reduction at p , p and the denominator of the coefficients of A coprime. E is an elliptic curve over $\mathbb{Z}/p\mathbb{Z}$ or $\text{GF}(2^n)$. np returns the number of points on the reduced elliptic curve A modulo p or on E .
npfe (E, N_p, P)	E is an elliptic curve over $\text{Gf}(2^m)$. N_p is the number of points of E over $\text{Gf}(2^m)$. P is a univariate polynomial of degree n over $\mathbb{Z}/2\mathbb{Z}$ or \mathbb{Z} or P is the degree n ; m must divide n ($m < n$). npfe returns the number of points of E over $\text{Gf}(2^n)$.
nthei (P)	P is a point on the current elliptic curve over \mathbb{Q} or over quadratic number fields. nthei returns the Néron–Tate height of P .
ntpair (P, Q)	P, Q are points on the current elliptic curve over \mathbb{Q} or over quadratic number fields. ntpair returns the Néron–Tate pairing of P and Q .

ordtsg (E)	E is an elliptic curve over \mathbb{Q} . ordtsg returns the order of the Tate-Shafarevic group of E .
prtoaf (P)	P is a point on an elliptic curve in projective representation. prtoaf returns the affine representation of P .
ptcoef (P, n)	P is a point on an elliptic curve; $n = 1, 2, 3$. ptcoef returns the n -th coefficient of P .
red (p, E)	p is a prime number; E is an elliptic curve over \mathbb{Q} . E must be minimal and have good reduction at p , p and the denominator of the coefficients of E coprime. red returns the reduced elliptic curve E modulo p .
reg (E)	E is an elliptic curve over \mathbb{Q} or over quadratic number fields. reg returns the regulator of E .
rk (E) rk (E, s) rk (E, s, b, v)	E is an elliptic curve over \mathbb{Q} or over a real quadratic number field with class number 1. s is 0, 1, 2, or 3. b is a single precision positive number. v is 0 or 1. rk returns the rank of the Mordell-Weil group of E . s determines the algorithm for computing the rank. For real quadratic fields, b is the upper bound for the search for points on the homogeneous spaces. v causes brief output or detailed output.
rk2d (E)	E is an elliptic curve over \mathbb{Q} or over a real quadratic number field with class number 1 which has a torsion group of order divisible by 2. rk2d returns the rank of the Mordell-Weil group of E , computed via 2-descent.
rkbsd (E)	E is an elliptic curve over \mathbb{Q} or over a real quadratic number field with class number 1. rkbsd returns the rank of the Mordell-Weil group of E , computed via the Birch and Swinnerton-Dyer conjecture.
rkg2d (E) rkg2d (E, b) rkg2d (E, b, v)	E is an elliptic curve over \mathbb{Q} or over a real quadratic number field with class number 1. b is a single precision positive number. v is 0 or 1. rkg2d returns the rank of the Mordell-Weil group of E , computed via general 2-descent. For real quadratic fields, b is the upper bound for the search for points on the homogeneous spaces. v causes brief output or detailed output.
rper (E)	E is an elliptic curve over \mathbb{Q} . rper returns the real period of E .
sgnfeq (E)	E is an elliptic curve over \mathbb{Q} . sgnfeq returns the sign of the functional equation of the modified Hasse-Weil L -function Γ .

swnf (E, s)	E is an elliptic curve over $\mathbb{Z}/p\mathbb{Z}(p \neq 2)$, \mathbb{Q} , \mathbb{Nf} ; s is 0 or 1. swnf returns a birational isomorphic model of E in short Weierstrass normal form. If $s = 1$, the transformation parameters r, s, t , and u are displayed.
taalg (p, E)	p is a prime number; E is an elliptic curve over a quadratic number field or with coefficients in \mathbb{Z} . If E is an elliptic curve over \mathbb{Q} and not minimal at p , taalg returns a birational isomorphic elliptic curve minimal at p . Otherwise, taalg displays the reduction type of E modulo p ; in the case of bad reduction at p , taalg also displays the Kodaira and Néron symbols and returns the index of the Néron symbol.
tavb2 (E)	E is an elliptic curve. tavb2 returns Tate's value b_2 of E .
tavb4 (E)	E is an elliptic curve. tavb4 returns Tate's value b_4 of E .
tavb6 (E)	E is an elliptic curve. tavb6 returns Tate's value b_6 of E .
tavb8 (E)	E is an elliptic curve. tavb8 returns Tate's value b_8 of E .
tavc4 (E)	E is an elliptic curve. tavc4 returns Tate's value c_4 of E .
tavc6 (E)	E is an elliptic curve. tavc6 returns Tate's value c_6 of E .
tors (E, s)	E is an elliptic curve with coefficients in \mathbb{Z} ; s is 1, 2, or 3. tors returns the order of the torsion group of E over \mathbb{Q} . Depending on the value of s , tors displays the structure, the generators and/or all elements of the torsion group.
whai (P)	P is a point on the current elliptic curve over \mathbb{Q} or over quadratic number fields. whai returns the Weil height of P .

For details see ‘‘?functionname’’.

4.4. simcalc's program control statements and user defined functions. The variable **DEPTH** determines the recursion depth of user-defined functions. By default, **DEPTH** = 30 (see “?progfunc”). You can change the value of **DEPTH** to n by typing in

>**DEPTH** = n ↵

In the following, B is a boolean expression which may contain any of the operators

== != < <= > >= && || .

Sequence is any number of expressions separated by _ or ; (for ; see §5.16).

break	break terminates the current loop.
continue	continue forces the next iteration of the current loop to take place. Before the next iteration, for <i>dowhile</i> and <i>while</i> loops, the boolean expression is evaluated, for <i>for</i> , <i>forprime</i> , and <i>forstep</i> loops, reinitialization is done.
decl (F_1, \dots, F_n) F_i : fct (p_1, \dots, p_m) = <i>seq</i> fct () = <i>seq</i>	F_1, \dots, F_n are user defined functions; fct is the function name; p_1, \dots, p_m are the function parameters; <i>seq</i> is a sequence. decl assigns the function F_i to the function name fct .
dowhile (B, seq)	<i>seq</i> is a sequence. <i>seq</i> is executed as long as the value of B is nonzero; <i>seq</i> is executed at least once.
for ($x = l, u, seq$) for ($x = l; , u, seq$) for (l, u, seq) for ($l; , u, seq$)	$l, u \in \mathbb{R}, l \leq u$. <i>seq</i> is a sequence. <i>seq</i> is executed while x (or the system variable @ in the third and fourth cases) runs from l to u in increments of 1.
forprime ($x = l, u, seq$) forprime ($x = l; , u, seq$) forprime (l, u, seq) forprime ($l; , u, seq$)	$l, u \in \mathbb{R}, l \leq u, u \geq 2$. <i>seq</i> is a sequence. <i>seq</i> is executed while x (or the system variable @ in the third and fourth cases) runs through all prime numbers in the range $[l, u]$.
forstep ($x = l, u, step, seq$) forstep ($x = l; , u, step, seq$) forstep ($l, u, step, seq$) forstep ($l; , u, step, seq$)	$l, u, step \in \mathbb{R}, step \neq 0$. If $step > 0, l \leq u$; if $step < 0, l \geq u$. <i>seq</i> is a sequence. <i>seq</i> is executed while x (or the system variable @ in the third and fourth cases) runs from l to u in increments of $step$.

if ($B, seq1, seq2$)	$seq1$ and $seq2$ are sequences. If the value of B is nonzero, $seq1$ is executed; otherwise $seq2$ is executed.
local (x_1, \dots, x_n)	x_1, \dots, x_n are variables. At the beginning of user defined functions, local creates the local variables x_1, \dots, x_n .
prod ($x = l, u, \text{exp}$) prod (l, u, exp)	$l, u \in \mathbb{N}$; exp is an expression. prod returns the product of the expression exp for $x = l$ to u or $@ = l$ to u .
return (A)	A is a simcalc expression. return terminates a user defined function and returns the value of A as the value of that function.
sum ($x = l, u, \text{exp}$) sum (l, u, exp)	$l, u \in \mathbb{N}$; exp is an expression. sum returns the sum of the expression exp for $x = l$ to u or $@ = l$ to u .
while (B, seq)	seq is a sequence. seq is executed while the value of B is nonzero.

The user defined functions are evaluated by calling **fct**(A_1, \dots, A_m) or **fct**(), where A_1, \dots, A_m are any valid **simcalc** expressions.

For details see ‘‘?cname’’.

5. Variable administration

Variable names

A variable name of at most 20 characters. The first character must be a letter and subsequent characters must be alphanumeric. A variable name cannot be the same as any of the following **simcalc** keyword.

Note: **simcalc** distinguishes between lower- and uppercase.

AV	continue	fclose	linequ	pconstr	sfp
EC	cos	file	ln	pfact	sgn
GF	cosh	flfunc	local	pfunc	sgnfeq
GF2bit	cot	fload	log	pgen	sin
I	coth	floor	log2	phi	sinh
MOD	cper	ftoi	log10	prfunc	sort
NF	curec	fopen	loglist	prim	sqrt
O	curgf	for	logoff	pripgen	stat
PT	curgf2bit	forprime	logon	print	statoff
Pi	curmod	forstep	mam2	prod	staton
aftopr	curnf	fp	mcoef	profunc	subst
arccos	decl	fread	mex	proot	sum
arccot	declaw	froot	mfunc	prtoaf	swnf
arcosh	deg	func	minim	ptcoef	sysvar
arcoth	denom	gcd	minor	putb	taalg
arcsin	deriv	getb	mod	putbits	tan
arctan	derivL	gethex	modfunc	putGF2bit	tanh
areptsli	det	getoct	mtoi	puthex	tavb2
arsinh	disc	gffunc	mval	puto	tavb4
artanh	discec	gftopol	nffunc	putoct	tavb6
aval	discnf	help	nffoff	quit	tavb8
avoff	dmconstr	herm	nfon	real	tavc4
avon	dowhile	i	nftopol	red	tavc6
basismwg	eccoef	if	norm	reg	tofl
binom	ecfunc	ifunc	np	relcln	tors
bitrans	ecgnp	imag	npfe	resul	trace
break	ecinf	intbas	nthei	return	transp
cfunc	egcd	intcoef	ntpair	rfunc	unit
chcoef	eival	integr	num	rk	vardel
chinrem	elform	irpgen	openf	rk2d	vcoef
chpol	elsubst	isprime	ord	rkbsd	vfunc
coef	exit	isptec	ordtsg	rkg2d	whei
cond	exp	jacsym	ow	root	while
conjug	fact	jinv	owoff	rper	zmconstr
cont	faintp	lcm	owon	scalar	zvconstr

Creating variables

If you want to assign the result of a computation to a variable **VAR**, enter

```
>VAR = expression ↵ or VAR[i] = expression ↵
```

If the variable **VAR** does not exist, it will be created and the value of **expression** assigned to it. For array variables, the index must be a nonnegative integer < 100000 .

The system variables @ and AV

If an expression is not explicitly assigned to a variable, it will be assigned automatically to the system variable @. Type in

```
>?sysvar ↵
```

to obtain further information on how to use @.

If besides the return value further results are computed by a function, these results are stored in the auxiliary array **AV**. The values in **AV** can be recalled for other computations (see ‘‘?avfunc’’).

The command **avon** enables the storage of additional results in **AV** and the command **avoff** disables it. By default, it is enabled.

The system variables DIGITS and OUTPUTDIG

The variable **DIGITS** gives the internal computation precision when performing operations with floating point and complex numbers and polynomials over these structures. The default value of **DIGITS** is 37. **OUTPUTDIG** determines the output format of numbers or polynomials. A value of 0 means output in scientific notation; any other value means output in fixpoint notation with **OUTPUTDIG** digits precision. You can change the value of **DIGITS** or **OUTPUTDIG** to n by typing in

```
>DIGITS =  $n$  ↵ or OUTPUTDIG =  $n$  ↵
```

Using predefined variables

Predefined and system variables can be used in any expression.

Assigned variables (variables store)

If you want a list of all the variables which have been assigned a value, enter

```
>?? ↵
```

The variables, ordered by type, and their current value are listed in lexicographical order.

If you type in

```
>? VAR1, ..., VARn ↵
```

these variables and their current value will be displayed. You can use the character `*` as a wildcard for any alphanumeric string.

Overwriting a variable

Whenever a variable is assigned a value, **simcalc** automatically checks whether the variable name has already been used. If so, the message “overwrite? (y/n)” will appear on the screen. If you enter **y**, the old value of the variable is replaced by the value of the expression; otherwise, you will be asked for another variable name.

At any point during the session, you can check the value of a variable with the **?** command (see §5.17).

Note: the command **CTRL/C** does not work when a variable is being overwritten (see §5.16).

The command **owoff** turns off the overwrite protection and **owon** turns it on (for details see ‘‘?ow’’).

The overwrite protection mechanism is automatically disabled within a loop. It is restored at the end of the loop. In user defined functions, there is no overwrite protection for local variables created by the **local** command.

Deleting variables

You can delete variables with

```
>vardel ↵
```

simcalc will ask for variable names; you can use the character `*` as a wildcard for any alphanumeric string. (Deleting all variables using `*` is ensured against possible misuse.) Enter **CTRL/D** to cancel the delete command.

CTRL/C does not work here (see §5.17) (for details see ‘‘?vardel’’).

Variable store overflow

The variable store can hold up to 300 variables. If it overflows, a message will appear on your screen and you will be asked to delete some variables. (The command **CTRL/C** does not work here (see §5.17).)

If you do not want to delete any variable, the value of the current expression will be assigned to the system variable **@**.

6. Computing in $\mathbb{Z}/m\mathbb{Z}$

Before doing any computation in $\mathbb{Z}/m\mathbb{Z}$, the user must specify the current modulus m with the function **curmod**(m). m is stored in the variable *curModulus* (for details see ‘‘?curmod’’).

The symbol **MOD** is used for input/output of elements from the residue class ring $\mathbb{Z}/m\mathbb{Z}$. For $A \in \mathbb{Q}$, structure over \mathbb{Q} , elliptic curve or point on an elliptic curve over \mathbb{Q} ,

$$X = \mathbf{MOD}(A)$$

assigns to X the corresponding value of A over $\mathbb{Z}/m\mathbb{Z}$.

Vectors are entered as

$$(1) \quad X = \mathbf{MOD}(\{a_1, \dots, a_n\})$$

or

$$(2) \quad X = \{a_1, \dots, \mathbf{MOD}(a_i), \dots, a_n\} ;$$

similarly for matrices. In case (2), the function **MOD** must be used for at least one parameter.

There is a special (internal) bit representation for matrices over $\mathbb{Z}/2\mathbb{Z}$ (see §4.4.2). You can enter a matrix in special bit notation by using the function **mam2**.

Remark: on the screen or in a file, e.g.

$$\mathbf{MOD}(100 * \mathbf{x} + 10 * \mathbf{x} * \mathbf{y} + 3)$$

is displayed as

$$\mathbf{MOD}(100) * \mathbf{x} + \mathbf{MOD}(10) * \mathbf{x} * \mathbf{y} + \mathbf{MOD}(3).$$

(For details see ‘‘?MOD’’.)

Remark: If $m_1, m_2 \in \mathbb{Z}/m\mathbb{Z}$, m defined by **curmod**, then for example the sum of m_1 and m_2 is given by **m1** + **m2** (see §5.4).

If the current modulus m is changed to m' using the function **curmod**, all the elements over $\mathbb{Z}/m\mathbb{Z}$ are transformed into elements over $\mathbb{Z}/m'\mathbb{Z}$ automatically.

7. Computing in Galois fields

Before doing any computation in a Galois field, the user must specify the current Galois field $\text{Gf}(p^n)$ with the function **curg** $\mathbf{f}(p, Y, Q)$, where p is a single-precision prime, Y is a variable and Q is a univariate irreducible polynomial in Y over $\mathbb{Z}/p\mathbb{Z}$ or \mathbb{Z} of degree n or the degree n ($n > 1$). In the second case, **curg** \mathbf{f} generates randomly an irreducible monic polynomial in Y over $\mathbb{Z}/p\mathbb{Z}$ of degree $Q = n$. In characteristic 2, the second possibility to specify the current Galois field is to use the function **curg** $\mathbf{f2bit}(Y, S)$, where Y is a variable and S is a sequence of 0 and 1. The sequence is interpreted as the coefficients of a polynomial in Y over $\mathbb{Z}/2\mathbb{Z}$ (For example, the sequence 100101 represents the polynomial $Y^5 + Y^2 + 1$).

The polynomial is stored in the variable *curGaloisField* (for details see “**?curg** \mathbf{f} ” or “**?curg** $\mathbf{f2bit}$ ”).

The symbol **GF** is used for input/output of elements from the Galois field $\text{Gf}(p^n)$. For $A \in \mathbb{Z}/p\mathbb{Z}, \mathbb{Q}$, structure over $\mathbb{Z}/p\mathbb{Z}$ or \mathbb{Q} , elliptic curve in long Weierstrass normal form or point on an elliptic curve over $\mathbb{Z}/2\mathbb{Z}$ or \mathbb{Q} ,

$$X = \mathbf{GF}(A)$$

assigns to X the expression corresponding to A over $\text{Gf}(p^n)$.

Vectors are entered as

$$\begin{aligned} (1) \quad X &= \mathbf{GF}(\{a_1, \dots, a_n\}) \\ \text{or} \\ (2) \quad X &= \{a_1, \dots, \mathbf{GF}(a_i), \dots, a_n\} ; \end{aligned}$$

similarly for matrices. In case (2), the function **GF** must be used for at least one parameter.

Remark: on the screen or in a file, e.g.

$$\mathbf{GF}(4 * x + 4 * x * y + Y)$$

is displayed as

$$\mathbf{GF}(4) * x + \mathbf{GF}(4) * x * y + \mathbf{GF}(Y) .$$

(For details see “**?GF**”).

In characteristic 2, you can enter Galois-field elements with the function

$$X = \mathbf{GF2bit}(S).$$

S is a sequence of 0 and 1. They are interpreted as coefficients of the Galois-field element (For example, $X = \mathbf{GF2bit}(1101)$ represents the Galois-field element $X = \mathbf{GF}(Y^3 + Y^2 + 1)$). If you want to see the bit representation of an element of $\text{Gf}(2^n)$, please use **putGF2bit** (see “**?putGF2bit**”).

Remark: if $e_1, e_2 \in \text{Gf}(p^n)$, defined by **curg** \mathbf{f} or **curg** $\mathbf{f2bit}$, then for example the sum of e_1 and e_2 is given by $e_1 + e_2$ (see §5.4).

If the current Galois field $\text{Gf}(p^n)$ is changed to $\text{Gf}(q^m)$ using the function **curgf**, all the elements of $\text{Gf}(p^n)$ are transformed into elements of $\text{Gf}(q^m)$ automatically. If $p = q$ and $n \mid m$, the user is asked if the variables over the old Galois field should be embedded into the new one (for details see ‘‘?curgf’’ or ‘‘?curgf2bit’’).

8. Computing in number fields

Before doing any computations with algebraic numbers, the user must specify the current number field K with the function **curnf**(Y, P), where Y is a variable and P is a univariate irreducible polynomial in Y ; then $K = \mathbb{Q}(\alpha)$ where $P(\alpha) = 0$, i.e. $K \cong \mathbb{Q}[Y]/(P(Y))$. All algebraic numbers are considered to be elements of this field K . P is stored in the variable *curNumberField* (for details see ‘‘?curnf’’).

The symbol **NF** is used for input/output of elements from the number field K . For $A \in \mathbb{Q}$, structure over \mathbb{Q} , elliptic curve or point on an elliptic curve over \mathbb{Q} ,

$$X = \mathbf{NF}(A)$$

assigns to X the expression corresponding to A over K .

Vectors are entered as

$$(1) \quad X = \mathbf{NF}(\{a_1, \dots, a_n\})$$

or

$$(2) \quad X = \{a_1, \dots, \mathbf{NF}(a_i), \dots, a_n\} ;$$

similarly for matrices. In case (2), the function **NF** must be used for at least one parameter.

Remark: on the screen or in a file, e.g.

$$\mathbf{NF}(100 * x + 10 * x * y + Y)$$

is displayed as

$$100 * x + 10 * x * y + \mathbf{NF}(Y) .$$

(For details see ‘‘?NF’’.)

Remark: if $e_1, e_2 \in K$, defined by **curnf**, then for example the sum of e_1 and e_2 is given by $e_1 + e_2$ (see §5.4).

Structures over number fields, which may be considered as structures over \mathbb{Q} , are usually treated as structures over the rationals. If you want to lift them to the number field specified by **curnf**, you can use the switch **nfon** / **nfoff**. After **nfon**, in the following functions - **cond**, **ecinf**, **fact**, **sfp**, **taalg** - the algorithms for structures over number fields are used for the structure over \mathbb{Q} . This is valid until **nfoff**.

If the current number field K is changed to K' using the function **curnf**, all the elements of K are transformed into elements over K' automatically.

9. Elliptic curves

Before doing any computations with elliptic curves, the user must specify the current elliptic curve with the function **curec**(E), where E is an elliptic curve over $\mathbb{Z}/p\mathbb{Z}$, \mathbb{Q} , Nf, or $\text{Gf}(2^n)$. The group law on the set of points is defined with respect to this curve until the next call to **curec**. The current elliptic curve is stored in the variable *curEllCurve* (for details see ‘‘?curec’’).

The symbol **EC** is used for input/output of elliptic curves. For a_1, \dots, a_6 and $a, b \in \mathbb{Z}/p\mathbb{Z}$, \mathbb{Q} , Nf, or $\text{Gf}(2^n)$,

$$X = \mathbf{EC}(a_1, a_2, a_3, a_4, a_6) \quad \text{or} \quad X = \mathbf{EC}(a, b)$$

assigns to X the elliptic curve in long or short Weierstrass normal form (respectively):

$$\begin{aligned} E : \quad y^2 + a_1xy + a_3y &= x^3 + a_2x^2 + a_4x + a_6 \quad \text{or} \\ E : \quad y^2 &= x^3 + ax + b. \end{aligned}$$

Remark: on the screen or in a file, **EC**(a_1, a_2, a_3, a_4, a_6) or **EC**(a, b) is displayed.

(For details see ‘‘?EC’’.)

The symbol **PT** is used for input/output of points on an elliptic curve. For $x, y, z \in \mathbb{Z}/p\mathbb{Z}$, \mathbb{Q} , Nf, or $\text{Gf}(2^n)$,

$$X = \mathbf{PT}(x, y, z) \quad \text{or} \quad X = \mathbf{PT}(x, y)$$

assigns to X a point on any elliptic curve in projective or affine coordinates (respectively).

Remark: on the screen or in a file, **PT**(x, y, z) or **PT**(x, y) is displayed.

(For details see ‘‘?PT’’.)

Elliptic curves over $\mathbb{Z}/p\mathbb{Z}$, Nf, or $\text{Gf}(2^n)$ are entered and displayed as

$$\begin{aligned} (1) \quad X &= \mathbf{MOD}(\mathbf{EC}(a_1, a_2, a_3, a_4, a_6)) \\ X &= \mathbf{NF}(\mathbf{EC}(a_1, a_2, a_3, a_4, a_6)) \\ X &= \mathbf{GF}(\mathbf{EC}(a_1, a_2, a_3, a_4, a_6)) \end{aligned}$$

or

$$\begin{aligned} (2) \quad X &= \mathbf{EC}(a_1, \mathbf{MOD}(a_2), a_3, a_4, a_6) \\ X &= \mathbf{EC}(a_1, \mathbf{NF}(a_2), a_3, a_4, a_6) \\ X &= \mathbf{EC}(a_1, \mathbf{GF}(a_2), a_3, a_4, a_6) \end{aligned}$$

respectively. In case (2), the function **MOD**, **NF**, or **GF** must be used for at least one parameter. Similarly for elliptic curves in short Weierstrass normal form and for points on elliptic curves.

Warning: form (2) must be used for elliptic curves with coefficients $\in K \setminus \mathbb{Q}$ or $\text{Gf}(2^n) \setminus \mathbb{Z}/2\mathbb{Z}$. **NF**(a_i) and **GF**(a_i) must be used for all coefficients $a_i \in K \setminus \mathbb{Q}$ and $\text{Gf}(2^n) \setminus \mathbb{Z}/2\mathbb{Z}$ respectively. a_i or **NF**(a_i) can be used for coefficients $\in \mathbb{Q}$, and a_i or **GF**(a_i) for coefficients $\in \mathbb{Z}/2\mathbb{Z}$. Similarly for points on elliptic curves.

Remark: the sum of two points on the current elliptic curve is given by $P_1 + P_2$ (see §5.4).

If the current modulus p is changed using the function **curmod**(), curves over $\mathbb{Z}/p\mathbb{Z}$ which would be singular over the new modulus are transformed into elliptic curves over \mathbb{Q} . If the current modulus p is changed to q , where q is not a prime, elliptic curves and points on elliptic curves over $\mathbb{Z}/p\mathbb{Z}$ are transformed into elliptic curves and points over \mathbb{Q} . Similarly for **curgf**().

O denotes the point at infinity in affine representation (for details see ‘‘?0’’).

The variable **HEIGHTBOUND** determines the upper bound of the height of points for the search in the algorithm of finding basis points of an elliptic curve (i.e. in `basismwg` and `faintp`). By default, **HEIGHTBOUND** = 11 (see “?ecfunc”). You can change the value of **HEIGHTBOUND** to n by typing in

`>HEIGHTBOUND = n ↵`

where n is an integer, a rational or a floating point number with $0 \leq n \leq 2^{999}$. If **HEIGHTBOUND** is zero, then the height of points is unlimited.

10. Computing over \mathbb{R} and \mathbb{C}

Input of floating point numbers and polynomials over \mathbb{R} can be done in fix point or scientific notation. Output is done depending on the value of **OUTPUTDIG** (see §5.5) (for details see ‘‘?flfunc’’).

simcalc computes floating point and complex numbers and polynomials over these structures with a precision of **DIGITS** digits; the default value is **DIGITS** = 37 (see §5.5).

Input of complex numbers can be done in two ways

- (1) $a + b * i$,
- (2) $a + b * I$;

they are always displayed in the first form. The variables i and I are reserved for $\sqrt{-1}$.

11. Variable substitution

In any computation with variables, it is possible to substitute values for the variables. The substitution is done by

$$X = f(x_1 = A_1, \dots, x_n = A_n).$$

f is either a polynomial, a rational function, a matrix or a vector; f must contain all the variables x_1, \dots, x_n . A_1, \dots, A_n are expressions which evaluate to numbers, polynomials or rational functions.

Recursive substitution and substitutions of the form

$$X = f(x_1 = x_2 = \dots = x_m = A)$$

or

$$X = f(x_1 = \dots = A = \dots = x_m), \quad m \leq n$$

are also allowed (for details see ‘‘?subst’’).

Warning: if A_i contains the variable x_j ($i < j$), then this x_j will also be substituted by A_j .

12. Reading from files

In **simcalc** it is possible to read strings from files; each string is displayed on the screen

fload, fload()	opens the default file <code>__INP</code> (it must exist in the current working directory), reads all lines and closes the file.
fload(NAME)	opens the file <code>NAME</code> (it must exist in the current working directory), reads all lines and closes the file.
fopen, fopen()	opens the default file <code>__INP</code> ; it must exist in the current working directory.
fopen(NAME)	opens the file <code>NAME</code> ; it must exist in the current working directory.

Note: no more than 10 files can be opened at the same time (for details see ‘‘?fopen’’).

fread, fread()	reads from the file <code>__INP</code> the line marked by the file pointer.
fread n, fread(n)	starting at the line marked by the file pointer, reads n lines from the file <code>__INP</code> .
fread(NAME)	reads from the file <code>NAME</code> the line marked by the file pointer.
fread(NAME, n)	starting at the line marked by the file pointer, reads n lines from the file <code>NAME</code> .

Before calling **fread**, the file must have been previously opened by **fopen** (for details see ‘‘?fread’’).

fclose, fclose()	closes the file <code>__INP</code> .
fclose(NAME)	closes the file <code>NAME</code> .

The file must have been previously opened before calling **fclose**.

openf	lists all files currently opened.
--------------	-----------------------------------

13. Predefinitions

Predefinitions for **simcalc** are read from the file **.simcalcrc** in the current or home directory.

14. Log files

As seen in §5.3, it is possible to “log” all or parts of a session with **simcalc** into a file.

Opening a log file

A log file with the name *FILE* is opened with the command

```
>logon(FILE) ↵
```

The command

```
>logon ↵ or >logon() ↵
```

opens the log file *__LOG*. If the file *FILE* or *__LOG* does not exist, it will be created; otherwise, the new protocol will be added to the end of the log file. **simcalc** will automatically close the current log file before opening a new one.

Immediately after opening a log file, all interaction with **simcalc** (input and output) will additionally be written to the file. (For details see ‘‘?log’’.)

Listing the contents of a log file

The command

```
>loglist ↵
```

displays the contents of the current log file on the screen.

Closing a log file

There are two ways to close the current log file:

- create a new log file
- use the command

```
>logoff ↵ or >logoff() ↵
```

(For details see ‘‘?log’’.)

15. Statistical functions

After receiving the command

```
>staton ↵
```

simcalc will list the following information after each computation:

computation time (in seconds) needed for the current computation
total computation time (in seconds) needed so far

If you no longer want the statistical information, enter the command

```
>statoff ↵
```

By default **simcalc** does not list statistical information. (For details see ‘‘?stat’’.)

16. Suppressing the output

It is sometimes convenient to be able to suppress the output of computational results, e.g. during loop execution. If you end the expression to be evaluated with a semicolon “;”, the result of the computation will be assigned internally to a variable (see §5.5), but will not be shown on your screen.

17. Interrupting output and computations

Interrupting the output

You can interrupt the output of a computational result with **CTRL/C**. **simcalc** confirms with the message

```
*****      Output is interrupted.      *****
```

When listing the contents of the variables store, only the output of the variable that is currently being displayed is interrupted.

Interrupting a computation

The command **CTRL/C** can also be used to interrupt a computation. **simcalc** confirms with the message

```
*****      Calculation is interrupted.      *****
```

Whenever the result of a computation is known *internally*, the equality symbol will appear on your screen. Then, **CTRL/C** interrupts the output of the result and the following message appears:

```
*****      Result is computed.      *****
*****      Output is interrupted.    *****
```

18. Special characters as commands

The ? command

The ? command provides detailed information on

- the contents of the variables store (see §5.5),
- deleting variables (see §5.5),
- working with log files (see §5.14),
- reading from files (see §5.12),
- the overwrite protection (see §5.5),
- using statistical functions (see §5.15),
- variable substitution (see §5.11),
- operators and functions (see §5.4).

For details on the ? command enter

```
>?help ↵
```

Detailed information is obtained with

?func	for general functions
?profunc	for program control statements
?modfunc	for functions over $\mathbb{Z}/m\mathbb{Z}$
?ifunc	for functions over \mathbb{Z}
?rfunc	for functions over \mathbb{Q}
?flfunc	for functions over \mathbb{R}
?cfunc	for functions over \mathbb{C}
?nffunc	for functions over number fields
?gffunc	for functions over $\text{Gf}(p^n)$
?pfunc	for functions over polynomials
?mfunc	for functions over matrices
?vfunc	for functions over vectors
?ecfunc	for functions over elliptic curves

The &-command

The command

```
>& ↵
```

displays **simcalc**'s introductory screen.

Using shell commands

During a **simcalc** session you can execute a shell command by typing

```
>$ command ↵
```

The system command **command** will be executed. Concatenating commands using `_` or `;` is not allowed (for details see ‘‘?\$\$’’).

Branching into a shell

The command

```
>! ↵
```

lets you branch into a subshell. Enter **CTRL/D** or **exit** to return to the current **simcalc** session (for details see ‘‘?\$\$’’).

19. Error messages

All **simcalc** error messages are self-explanatory. **simcalc** reports only the first error found.

You can use the `%` command (see §5.2) (**CTRL/P** or **CTRL/N** for the GNU readline version) to correct erroneous input.