LiE MANUAL DRAFT

describing LiE version 2.0

Marc A. A. van Leeuwen
Arjeh M. Cohen
Bert Lisser

LiE is a software package for Lie group theoretical computations

developed by the

Computer Algebra Group
of the
CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

```
LiE LiE     LiE LiE     LiE LiE LiE LiE
LiE LiE     LiE LiE     LiE LiE LiE LiE
LiE LiE     LiE LiE     LiE LiE
LiE LiE     LiE LiE     LiE LiE
LiE LiE     LiE LiE     LiE LiE LiE LiE
LiE LiE     LiE LiE     LiE LiE LiE LiE
LiE LiE     LiE LiE     LiE LiE
LiE LiE                 LiE LiE
LiE LiE LiE LiE LiE     LiE LiE LiE LiE
LiE LiE LiE LiE LiE     LiE LiE LiE LiE
```

LiE MANUAL DRAFT

describing LiE version 2.0

Marc A. A. van Leeuwen
Arjeh M. Cohen
Bert Lisser

LiE is a software package for Lie group theoretical computations

developed by the

Computer Algebra Group
of the
CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

```
LiE LiE      LiE LiE      LiE LiE LiE LiE
LiE LiE      LiE LiE      LiE LiE LiE LiE
LiE LiE      LiE LiE      LiE LiE
LiE LiE      LiE LiE      LiE LiE
LiE LiE      LiE LiE      LiE LiE LiE LiE
LiE LiE      LiE LiE      LiE LiE LiE LiE
LiE LiE      LiE LiE      LiE LiE
LiE LiE                   LiE LiE
LiE LiE LiE LiE LiE       LiE LiE LiE LiE
LiE LiE LiE LiE LiE       LiE LiE LiE LiE
```

LiE Manual
Chapter 1. INTRODUCTION

LiE is the name of a software package under development at CWI since January 1988. Its purpose is to enable mathematicians and physicists to obtain on-line information as well as to interactively perform computations of a Lie group theoretic nature. It focuses on the representation theory of complex semisimple (reductive) Lie groups and algebras, and on the structure of their Weyl groups and root systems.

The basic objects of computation are vectors and matrices with integer entries, and polynomials with integral coefficients. These objects are used to represent weights, (sets of) roots, characters and similar objects relating to Lie groups and algebras. LiE does not compute directly with elements of the Lie groups and algebras themselves, but the computations may be parametrised by the type of the Lie group or algebra for which they should be performed. Our primary goal in realising the present version has been to cover (on-line) the mathematical content of the following three books:

[Tits 1967]	J. Tits, *Tabellen zu den einfachen Lie Gruppen und ihren Darstellungen*, Lecture Notes in Math. 40, Springer, Berlin, 1967.
[Brem ea 1985]	M. R. Bremner, R. V. Moody, J. Patera, *Tables of dominant weight multiplicities for representations of simple Lie algebras*, Monographs and Textbooks in Pure and Appl. Math. 90, Dekker, New York, 1985.
[McKay ea 1981]	W. G. McKay & J. Patera, *Tables of dimensions, indices and branching rules for representations of simple Lie algebras*, Lecture Notes in Pure and Appl. Math. 69, Dekker, New York, 1981.

The package establishes an interactive environment from which commands can be given, involving basic programming primitives as well as powerful built-in mathematical functions (the package can be run in batch mode as well.) These commands are read by an interpreter built into the package and passed through to the core of the system: a collection of programs representing the various available mathematical functions. Furthermore, the interpreter offers online facilities which explain the operations and functions available, give background information about Lie group theoretical concepts, and give information about currently valid definitions and values.

LiE is written in C, and can be made available on any system running UNIX or comparable operating systems, and (with a little more effort) probably on many other machines with a C-compiler. The interpreter has been set up with the help of the UNIX program "yacc". The present version is available for the following computers: SUN 3, SUN 4 and SparcStation (under SunOS 4.0), VAX (under BSD UNIX 4.3 *and* under VMS), IBM RT (under AIX), DEC3100 (Ultrix) and Apple Macintosh. Should you want to order the package, please contact: Computer Algebra Group, c/o Marc van Leeuwen, CWI, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, email: maavl@cwi.nl.

1.1. About the content of this manual

Chapter 2 explains the environment offered by the LiE interpreter. It explains how to evaluate expressions, call built-in functions, and invoke the online help facilities. It also defines a programming language in which users may define their own algorithms, making use of the built-in operations and functions. The interpreter recognises the following types of objects.

type	name	example	comment
integer	**int**	`-12344321267`	arbitrary size
vector	**vec**	`[1,2,-7,6,9,8]`	machine size integer entries
matrix	**mat**	`[[1,2],[3,-4]]`	row length should not vary
polynomial	**pol**	`X[1,0]-7X[3,-5]`	multivariate Laurent polynomials
group	**grp**	`A6A6E8F4T4`	T_4 is a 4-dimensional torus
text	**tex**	`"any string"`	quotes are required
void	**vid**		to unify functions and procedures

The about 100 mathematical functions which form the heart of the LiE package are described in detail in Chapter 4; these involve amongst others root systems, Weyl groups, multiplicities and degrees of highest weight modules, tensor product decompositions, branching (i.e., restriction of modules) to reductive subgroups, centralisers of a semisimple elements, and the spectrum of such elements on a module. In order to describe these functions, it is necessary to introduce the relevant mathematical terms and concepts, and the way in which these are represented in LiE; these matters are described in Chapter 3.

The LiE programming language makes it possible to customise and extend the package with more mathematical functions; examples of this are given in Chapter 5.

1.2. Theoretical aspects

The package is mainly intended for computations concerning semisimple Lie groups and algebras. Since reductive groups provide a more general and at the same time more convenient setting, they form the class of groups we have chosen to work with. For notational convenience, we adapt names only for groups whose semisimple part is simply connected. Since all other reductive groups are quotients of these by finite central subgroups, we feel that this is not a major limitation.

Most mathematical functions implemented in LiE have a Lie group as argument. No multiplication of Lie algebra or Lie group elements is available. The notion of group we use is hardly more than an indication of its isomorphism class. The computations are mainly done on the level of vectors, matrices and polynomials corresponding to various relevant objects in Lie group theory. For instance, representations are parametrised by vectors via the so-called *highest weights*, and the elements of the Weyl group of a Lie group appear in different guises (they can be represented both as vectors, indicating a product of fundamental reflections, and as matrices, indicating the image in the reflection representation).

The emphasis has been on the development of basic routines that perform the mathematical operations in the greatest generality. Therefore, it is quite likely that greater speed could have been achieved in specific cases with more specialised programs. In one instance we have also realised algorithms specific for certain types of groups, namely the Young tableau techniques, giving fast implementations for certain computations in the special linear groups (notably the Littlewood-Richardson rule).

1.3. The authors

Arjeh M. Cohen developed the idea, wrote some mathematical functions and a first version of this manual and is the project leader. Bert Lisser made the interpreter and provided the information for Chapter 2 of this manual. Marc van Leeuwen is the author of the current version of this manual, and implemented the Young tableau algorithms. An earlier version of the package was constructed with aid of Ron Sommeling, Bart de Smit and Bert Ruitenburg, who are no longer involved in the project; we hope that they still appreciate what we have done to LiE.

For more information beyond what this manual has to offer, bug reports, interesting algorithms you may want us to know, or any other helpful comments, contact: Arjeh M. Cohen, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, email: marc@cwi.nl.

LiE Manual

Chapter 2. THE INTERPRETER

In this chapter, the facts needed to run a successful LiE session are described. We discuss the features of the interactive shell, that interprets the commands entered during a session. After an introductory session, we give more details of the types of objects the interpreter recognises. Then, in Section 2.3, the operators defined in the package are listed, and Section 2.4 similarly treats functions. Section 2.5 discusses the ingredients needed to construct larger programs, and Section 2.6 shows how to define your own functions. Finally Section 2.7 describes some features that allow the user additional control over LiE. A note about the typography of this chapter: in the introductory section, all commands as typed by the user and the responses of LiE are reproduced in typewriter type style, to indicate the exact appearance on the screen, but in the further sections a more aesthetically pleasing form of rendering expressions is chosen, distinguishing identifiers (*italic* type), keywords (**bold** type) and direct commands to LiE (`typewriter` type).

2.1. A first look

An interactive session of LiE starts by executing the command `LiE` on your machine (provided LiE has been installed; the leaflet accompanying the software package explains how to do that). You will then enter the *Lie shell*, a sign-on message will be printed, followed by the prompt '`>`'. In this mode, you can enter commands. A command will be executed upon completion of the line by hitting ⟨Return⟩. The command will be read by an *interpreter* built into LiE and, if necessary, will invoke some of the mathematical functions. The system will respond to the command by returning an answer if relevant. In the examples given below, the lines starting with the prompt character '`>`' are the commands as typed by the user, the other lines are LiE's response.

Your first concern after entering LiE is of course if it is possible to get out again; the answer is *yes*, it suffices to type

```
> quit
```

and LiE will sign off with '`end program`' and stop (synonyms for `quit` are `exit` and `@`). Should you at any moment find that LiE has embarked on a (seemingly) endless computation, then you can always force it to abort the computation and prompt for a new command by typing ⟨control⟩`C`, i.e., press the control key and the `c` simultaneously.

The simplest commands are to perform arithmetic computations; the interpreter then behaves like a pocket calculator, evaluating the expression typed in and printing the result.

```
> 19+68
     87
> 1111111111*1111111111
     1234567900987654321
> $/1111111111
     1111111111
> -$ % 1000003
     892225
>
```

Here $ means "the previous result", and % means "modulo". Variables may be used to save values in a more permanent way than in $.

```
> a=345
> a^2+3*a-5
     120055
> $/7*a
     5916750
```

Besides integer arithmetic, LiE can also calculate with vectors and matices with integer entries. Here are a few operations with vectors, from these the reader may guess which computations are performed.

```
> v=[3,2,6873,-38]
> v
     [3,2,6873,-38]
> v[3]
     6873
> v[5]
Index (= 5) out of range
(in _select)
> v+v
     [6,4,13746,-76]
> v*v
     47239586
> v+234786
     [3,2,6873,-38,234786]
> v-3
     [3,2,-38]
> v^v
     [3,2,6873,-38,3,2,6873,-38]
```

We can play similarly with matrices.

```
> [[1,0,3,3], [12,4,-4,7], [-1,9,8,0], [3,-5,-2,9]]
     [
     [ 1, 0, 3,3],
     [12, 4,-4,7],
     [-1, 9, 8,0],
     [ 3,-5,-2,9]
     ]
```

```
> m=$
> *m
     [
     [1,12,-1, 3],
     [0, 4, 9,-5],
     [3,-4, 8,-2],
     [3, 7, 0, 9]
     ]

> m^3
     [
     [ 220,   87, 81, 375],
     [-168,-1089, 13,1013],
     [1550,  357,-55,1593],
     [-854, -652, 98,-170]
     ]

> v*m
     [-6960,62055,55061,-319]
> m*v
     [20508,-27714,54999,-14089]
> v*m*v
     378549605
> m+v
     [
     [ 1, 0,   3,  3],
     [12, 4,  -4,  7],
     [-1, 9,   8,  0],
     [ 3,-5,  -2,  9],
     [ 3, 2,6873,-38]
     ]

> m-2
     [
     [ 1, 0, 3,3],
     [-1, 9, 8,0],
     [ 3,-5,-2,9]
     ]
```

Apart from integers, vectors and matrices, LiE can also calculate with (multivariate) polynomials. Because of the specific intended applications to Lie group theory, polynomials are represented in a way that may seem a bit unusual. First of all, there are no formal names such as X, Y, ..., for the polynomial indeterminates: the indeterminates are simply discriminated by their position in a fixed ordering, and monomials are represented by the symbol 'X' followed by a vector as "exponent", where the first number gives the exponent of the first indeterminate, etc. Moreover, LiE will not mix terms with different numbers of indeterminates, so zeros should be added as necessary to make all exponents the same size. Finally negative integer exponents are allowed, so we are in fact dealing with Laurent polynomials; the ground ring is **Z**. Here is a session with some simple polynomial calculations.

```
> X[1,2]
     1X[1,2]
> -3*$
     -3X[1,2]
> $+4X[-1,4]
      4X[-1,4] - 3X[ 1,2]
> $+X[6,7,8]
Number of variables in polynomials unequal
( 2 <-> 3 variables).
(in +)
> $*(X[2,0]-X[0,-4])
     -4X[-1, 0] + 3X[ 1,-2] + 4X[ 1, 4] - 3X[ 3, 2]
> $-$
     0X[0,0]
```

The core of LiE is a batch of built-in functions which can be called by the interpreter. We give two simple examples of such calls:

```
> partitions(6)
     [
     [6,0,0,0,0,0],
     [5,1,0,0,0,0],
     [4,2,0,0,0,0],
     [4,1,1,0,0,0],
     [3,3,0,0,0,0],
     [3,2,1,0,0,0],
     [3,1,1,1,0,0],
     [2,2,2,0,0,0],
     [2,2,1,1,0,0],
     [2,1,1,1,1,0],
     [1,1,1,1,1,1]
     ]


> diagram(E8)
          O 2
          |
          |
O---O---O---O---O---O---O
1   3   4   5   6   7   8
E8
```

The former call returns the matrix whose rows represent partitions of 6; the latter command prints the diagram shown, but does not deliver a resulting value, so some might wish to call *diagram* a procedure rather than a function.

The user may also define in a natural way functions that are not built into LiE, for example

```
> f(int x)=2*x
> f(984)
     1968
```

Instead of giving the resulting value at once, as in this example, one may also specify a sequence of statements to be executed first (separated by semicolons), followed by the expression giving the result.

```
> f(int n)= a=3*n-7; if a<0 then a=-a fi; 7^a+a^3-4*a-57
> f(2)
     -53
> f(5)
     5765224
```

For conditional statements (and expressions) as in the above example, logical expressions are useful; there is a number of relational and logical operators, which are represented in the same style as in the programming language "C". Some examples of logical expressions are

```
i<=n
n==8
p>10 && p!=13
f(3)<=7 || k+1 >= 5
```

Some commands describe an action to be performed rather than a value to be computed, and are called statements; examples are

```
a=[2,3]; b=7; v[2]=7
for i=1 to n do print(i*i) od
```

Statements do not yield a value, so unless the specified action explicitly produces output (as in the case of `print`), nothing will appear on the screen. In the last example we showed a loop entered directly to the interpreter; here is an example of the use of a loop within a function

```
> sum_sq(vec v)= s=0; for i=1 to size(v) do s=s+v[i]^2 od; s
> sum_sq([1,-3,5,2,7])
     88
```

There are commands for global control of LiE, such as '`quit`' above, and to control the input and output flow; some examples are

```
on monitor
edit script
```

of which the first starts recording the session on a file "`monfil`", and the second invokes an editor on the file "`script`", which presumably contains commands to LiE that will be read by LiE upon completion of the edit session.

Finally, there are some features to help you out, such as

```
listvars
learn lie group
```

The former lists the variables that have been given a value, while the latter prints a text indicating what the authors of LiE think a Lie group is.

The objects that LiE can manipulate are of the following types (in each case the indication LiE uses to designate the type is added in parentheses): integer (**int**), vector (**vec**), matrix (**mat**), polynomial (**pol**), group (**grp**), or text (**tex**); there is also the indication **vid** that stands for void, which is not really a type since there is no void value that could be assigned to a variable or passed to a function, but is used to indicate the result type of a function that does not yield any value. Variables do not have a declared type: they simply assume the type of any value that is assigned to them. However, once created variables cannot change their type during a computation: their type can only change by an assignment typed directly by the user (which of course can only happen to global variables).

We end this section with a few essential details.

2.1.1. Command prolongation

As mentioned above, a command normally ends at the end of a line. We have implemented this rule because, usually, one line suffices for a command. However, if the line ends with one of the characters '+', '-', '*', ';', ',' or '\' (none of which can be the last character of a valid command) then the command will be considered to continue onto the next line. When used in this way the character '\' is equivalent to a space (and it can therefore be inserted at almost any convenient place), while the other characters stand for themselves. A command is also assumed to continue beyond the end of a line when there are still unclosed left parentheses, brackets, braces, or unfinished conditional or loop clauses, which means that in most cases you need not bother to type any backslashes. To indicate that the remainder of a command is awaited, the prompt changes from '>' to '\'. This command prolongation cannot be used after `?`, `help`, or `:`, or within a string or comment.

2.1.2. Getting help

Use `?`, `help`, or `?help` to make enquiries. Other text following `?` can be used to get more detailed information about a particular topic. For example, `?functions` returns the list of built-in functions. The command ?⟨name⟩ returns information about the variable, function(s) or operator(s) with the specified name. So, for instance `?lierank` will return information on the built-in function *lierank*. For built-in functions, similar information can also be found in Chapter 4 of this manual.

The commands `listvars`, `listfuns` and `listops` print lists of the variables, or functions defined in the session, respectively of the operators known to LiE (cf. Section 2.3).

2.1.3. Variables

Variable names are strings of letters, digits and underscores; the first character must be a lower case letter (this requirement is necessary because for instance `A68` denotes a group, and therefore cannot be a variable).

The special variable **$** is given the value returned by the last command that did in fact deliver a value (so assignments and calls for help etc. do not alter the value of **$**). Note that only *commands* set the value of **$**; this implies that

```
> 10
     10
> 13; $
```

returns 10 rather than 13.

2.1.4. File management

Commands contained in a file named ⟨name⟩ can be read with command `read` ⟨name⟩. The same file can be edited by issuing the command `edit` ⟨name⟩. When the editing session is completed, the file will automatically be read in.

2.1.5. Comments

Comments are enclosed between a pair of characters '#' (and accordingly comments cannot contain the character '#'). If the closing '#' is missing, the comment will be closed at the end of the line.

2.1.6. Escape to the shell

The character '`:`' appearing as first character of a command line means that the remainder of the line is passed to the shell (this feature applies to UNIX implementations of LiE only). This is a newly created subshell, not the (login) shell from which LiE was called, so for instance it makes little sense to invoke a `cd` command in this manner.

2.2. Values

As mentioned above, LiE handles values of the types integer, vector, matrix, polynomial, group and text. We now treat these kinds of values in some more detail.

2.2.1. Integer

Integers are represented by LiE by values of type **int**; as usual, they may be denoted by a sequence of digits, optionally preceded by a minus sign.

Integers and coefficients of polynomials effectively have unlimited length. The integer entries of vector, matrices and the exponents in polynomials are restricted however by the word size of the machine (usually this allows values up to 2^{31} in magnitude). This restriction is made for efficiency reasons; for most purposes it is hardly a limitation since the running time of most Lie group theoretic functions becomes excessively large long before the entries of the vectors and matrices occurring as parameters or results of these functions reach their limits. Note that whereas a warning is issued if one tries to insert too big an integer into a vector, matrix, or polynimial exponent, no such warning is generated when overflow occurs within an operation on vectors, matrices and (very unlikely) polynomials themselves, e.g., when calculating a huge power of a matrix.

2.2.2. Vector

An object of type **vec** is a vector, which consists of a sequence of integers: it has a size s (which may be 0), and entries indexed by the numbers $1, \ldots, s$. A vector may be formed by a comma separated list of integer expressions enclosed in square brackets, such as `[1,9,6,8]`, `[32*13*9497,30-9*101*677]` and `[]`. It is also possible to form vectors whose size is determined at run time by calling *null*(n) or *all_one*(n), where in either case n stands for an arbitrary integer expression whose value determines the size of the vector created; in the case of *null* all entries are set to 0, while in case of *all_one* they are all set to 1. If v is a vector of size n, then its individual entries may be referred to as $v[i]$ for $1 \leq i \leq n$. The built-in function *size* allows the size of the vector v to be obtained as *size*(v).

2.2.3. Matrix

An object of type **mat** is a matrix, which consists of a rectangular array of integers: it has a number of rows r and a number of columns c, and integer entries indexed by pairs i, j of integers with $1 \leq i \leq r$ and $1 \leq j \leq c$. A matrix may be formed by a comma separated list of vector expressions enclosed in square brackets, such as `[[5,-4],[-6,5]]`, and `[[4-7,11],v]` after assigning `v=[6,9]`. Since matrices are always rectangular, it is required that all vectors occurring have the same size; they will be taken in order to form the successive rows of the matrix (note that it is possible to denote matrices with 0 columns in this way, but not with 0 rows; the latter can be created with the call *null*$(0, n)$). This notation is in accordance with the general convention in LiE that whenever a matrix is considered as a sequence of vectors, these correspond to the rows (rather than the columns) of the matrix. The functions *null* and *all_one* can also be used to create matrices, by supplying them with two integer arguments: the first argument determines the number of rows and the second the number of columns of the matrix, while the entries are all 0 or all 1 as in the case of vectors.

A matrix is printed in the same way as it is entered, with the vectors representing the rows on separate lines, and the opening and closing brackets of the whole matrix on lines by themselves (note however that it is possible to alter the style of printing such that a matrix appears just as a rectangular block of numbers enclosed in vertical bars, by means of the system parameter `lprint`, see Section 2.7.4). This method of printing is slightly ambiguous (an not in agreement with the input format) when matrices with 0 rows are concerned.

Since a matrix is often viewed as a sequence of its rows, the rows of is a matrix m with r rows, may be referred to as $m[i]$ for $1 \leq i \leq r$; the individual entries of the matrix may be referred to as $m[i, j]$ or as $m[i][j]$, both denoting the same entry. Similarly to the function *size* for vectors, there are functions *n_rows* and *n_cols* to determine the number of rows and columns of matrices.

2.2.4. Polynomials

An object of type **pol** is a Laurent polynomial in a fixed number n of indeterminates. It consists of a set of *terms* (which are automatically sorted by LiE) where each term has an integer *coefficient*, and an *exponent*, which is a vector of n integers, the i-th of which represents the power in which the i-th indeterminate occurs. Whenever terms with equal exponents would occur, they are automatically combined by LiE, whence it is guaranteed that all terms occurring have distinct exponents. There is always at least one term: the zero polynomial is represented by a term with coefficient 0 and a zero vector of the appropriate size as exponent; apart from this case coefficients are always non-zero. Terms are denoted as an optional integer coefficient (the default is 1) followed by the symbol `X` followed by a vector as exponent; polynomials with multiple terms can be formed by addition and subtraction of terms. For polynomials in 1 indeterminate one may also write an integer exponent, which is automatically converted into a vector of size 1. Polynomials are printed in the same format as they are entered (assuming the default setting of the `lprint` parameter), with coefficients always explicitly represented (even if equal to 1) and exponents always rendered as vectors. Polynomials in n indeterminates corresponding to the integer numbers 0 and 1 can be formed by *poly_null*(n) an*poly_one*(n) respectively; these calls are equivalent to

$0X\,null(n)$ and $1X\,null(n)$.

The order in which the terms of a polynomial are sorted depends on the setting of system parameters; the default is *lexicographic ordering* of the exponents, but the user may also select *total degree ordering* (in which case the sum of the exponents entries takes precedence over the lexicographic ordering) and the reverse ordering of either of these two possibilities. This ordering influences the selection of terms: the i-th term of a polynomial p can be referred to as $p[i]$ (which is a polynomial of one term). The coefficient of the i-th term can be obtained as $coef(p,i)$, and the exponent of that term as $monom(p,i)$ (which is a vector). Further functions to obtain information about polynomials are *n_vars*, giving the number of indeterminates, *length*, giving the number of terms, *degree*, giving the total degree of p, i.e., the largest integer obtainable as sum of entries of some exponent. It is not only possible to select coefficients by their position, they may also be selected by exponent: $p|v$ denotes the coefficient of the term with exponent v, or zero if no such term exists. One may also assign to $p|v$ in order to alter the coefficient of the term with exponent v; this may cause a term to be created or deleted as appropriate, as the following example shows.

```
> p = X[1,5]
> p
      1X[1,5]
> p|[3,7]=-5
> p
      1X[1,5] - 5X[3,7]
> p|[1,5]=8; p
      8X[1,5] - 5X[3,7]
> p|[1,5]=0; p
     -5X[3,7]
```

It is also possible to supersede an entire term $p[i]$ of a polynomial by another one by assigning to it, but note that because the polynomial is normalised afterwards by possibly rearranging and merging of terms, it is not generally true that after assigning $p[i] = term$ we have that $p[i] == term$ holds.

2.2.5. Group

A value of type **grp** specifies an isomorphism class of reductive complex Lie groups with simply connected semisimple part. As will be explained in Section 3.1, such groups are a direct product of simple groups (its simple components) and a central torus, where simple groups are characterised by their type and the central torus by its dimension. Therefore, LiE represents groups by a sequence of types of simple groups together with the dimension of the central torus. Types of simple groups are of the form L_n where L is an upper case letter from the set $\{A,B,C,D,E,F,G\}$ and n is a positive number, subject to the restrictions $n \geq 2$ if $L \in \{B,C\}$, $n \geq 3$ if $L = D$, $n \in \{6,7,8\}$ if $L = E$, $n = 4$ if $L = F$ and $n = 2$ if $L = G$. The letters *A–D* correspond to the classical groups (cf. [Bourb 1968]), which groups are also known by proper names as follows:

$$A_n\colon SL(n+1,\mathbf{C}),\quad B_n\colon Spin(2n+1,\mathbf{C}),\quad C_n\colon Sp(2n,\mathbf{C}),\quad D_n\colon Spin(2n,\mathbf{C})$$

The type of the n-dimensional torus $(C^*)^n$ is T_n. To denote a group in LiE one simply concatenates the types of the simple components and the central torus. The order of the simple components is retained by LiE, but each term T_n simply increases the dimension of the central torus by n; when a group is printed by LiE, the central torus appears at the end. For instance, if you enter `C3T4B12A4T6A1E7` then LiE will print `C3B12A4A1E7T10` as a result, which specifies the group

$$Sp(6,\mathbf{C}) \times Spin(25,\mathbf{C}) \times SL(5,\mathbf{C}) \times SL(2,\mathbf{C}) \times E_7(\mathbf{C}) \times (C^*)^{10}$$

For a group g, the simple group that is its i-th component may be referred to as $g[i]$, while its central torus may be referred to as $g[0]$, so for g as in the above example, we have $g[0] = T_{10}$ and $g[2] = B_{12}$. In some cases the mathematical specification would require that a function returns a group whose semisimple part is not simply connected (e.g., the function *centr*). Since such groups are not representable in LiE, the group of which it is a central quotient with finite kernel, and whose semisimple part *is* simply connected, is returned instead in such cases.

2.2.6. Text

LiE has some basic means to manipulate character strings for output, in the form of values of type **tex**. Strings are denoted by enclosing them in double quote characters, and they should be given on a single line, for instance `"this is a string"`; it follows that strings cannot contain the double quote and newline characters.

2.3. Operators

We describe the operators defined in LiE. Contrary to functions, it is not possible to define new operators, or additional instances of existing operators. The meaning of an operator and the type of its result depend on the types of its operands (this holds for functions as well). Each operator has a priority, which determines how expression are parsed: as usual, the implicit parentheses fit more closely around operators of higher priority. At each priority level association is to the left, i.e., among operators of equal priority implicit parentheses group towards the left.

There is no type 'Boolean', so that truth values are represented by integers: relational and logical operators yield 1 when true and 0 when false. When an arbitrary integer is interpreted as a truth value, all values except 0 are considered as representing **true**. There are however syntactic restrictions that prevent perfoming arithmetic with truth values: expressions such as $100+(3<4)$ are forbidden. The result of a relational or logical operator may *only* be used between '**if**' and '**then**' or **while** and **do**, as operand of a logical operator, in an assignment to a variable, or in a list of function parameters or of vector entries.

We give the operators, their priorities and their various meanings by a table. In each case the first operand is called a, the second b; there might be only one argument, in which case the operator is used monadically, written before its operand. In the case of vector, matrix and polynomial operands, some restriction is often imposed on the size, respectively on the number of rows, columns, or indeterminates; we use the notation σ_a for the size of a vector a, ρ_a and κ_a for number of rows and columns respectively of a matrix a, and ν_a for the number of indeterminates of a polynomial a.

operator	priority	type of a	type of b	type of result	meaning, comments
$+$	6	**int**	**int**	**int**	$a+b$
		vec	**vec**	**vec**	$a+b$ (vector addition) $\{\sigma_a=\sigma_b\}$

		mat	**mat**	**mat**	$a+b$ (matrix addition) $\{\rho_a=\rho_b,\kappa_a=\kappa_b\}$
		pol	**pol**	**pol**	$a+b$ (polynomial addition) $\{\nu_a=\nu_b\}$
		int	**vec**	**vec**	$[a,b[1],b[2],\ldots,b[\sigma_b]]$
		vec	**int**	**vec**	$[a[1],a[2],\ldots,a[\sigma_a],b]$
		mat	**vec**	**mat**	$[a[1],a[2],\ldots,a[\rho_a],b]$ $\{\kappa_a=\sigma_b\}$
		tex	**tex**	**tex**	concatenation
		tex	**int**	**tex**	$a+t$ where t is textual representation of b
		int	**tex**	**tex**	$t+b$ where t is textual representation of a
		tex	**grp**	**tex**	$a+t$ where t is textual representation of b
		grp	**tex**	**tex**	$t+b$ where t is textual representation of a
$-$	6	**int**	**int**	**int**	$a-b$
		vec	**vec**	**vec**	$a-b$ $\{\sigma_a=\sigma_b\}$
		mat	**mat**	**mat**	$a-b$ $\{\rho_a=\rho_b,\kappa_a=\kappa_b\}$
		pol	**pol**	**pol**	$a-b$ $\{\nu_a=\nu_b\}$
		vec	**int**	**vec**	make a one shorter by removing $a[b]$
		mat	**int**	**mat**	make a one row shorter by removing row $a[b]$
$-$	10	**int**		**int**	$-a$
		vec		**vec**	$-a$
		mat		**mat**	$-a$
		pol		**pol**	$-a$
$*$	7	**int**	**int**	**int**	ab
		int	**vec**	**vec**	$a\cdot b$ (scalar multiplication by a)
		vec	**vec**	**int**	$ab^\top=\sum_{i=1}^{\sigma_a}a[i]b[i]$ (standard inner product of a and b) $\{\sigma_a=\sigma_b\}$
		int	**mat**	**mat**	$a\cdot b$ (scalar multiplication by a)
		vec	**mat**	**vec**	ab (right multiplication by matrix b) $\{\sigma_a=\rho_b\}$
		mat	**mat**	**mat**	ab (matrix multiplication) $\{\kappa_a=\rho_b\}$
		mat	**vec**	**vec**	$ba^\top=(ab^\top)^\top$ (left multiplication of column vector b by matrix a) $\{\kappa_a=\sigma_b\}$
		int	**pol**	**pol**	$a\cdot b$ (scalar multiplication by a)
		pol	**pol**	**pol**	$a*b$ (polynomial multiplication) $\{\nu_a=\nu_b\}$
		pol	**mat**	**pol**	multiply all exponents of terms of a on the right by b and normalise result $\{\nu_a=\rho_b\}$
		pol	**int**	**pol**	$a*(b\cdot id(\nu_a))$ (see previous line)
		grp	**grp**	**grp**	$a\times b$ (concatenation of simple factors, addition of dimensions of central torus)
		int	**tex**	**tex**	the string b repeated a times
		tex	**int**	**tex**	the string a repeated b times
$*$	10	**mat**		**mat**	$a^\top$ (matrix transposition)
/	7	**int**	**int**	**int**	a/b rounded towards 0
		vec	**int**	**vec**	$[a[1]/b,\ldots,a[\sigma_a]/b]$
		mat	**int**	**mat**	$[a[1]/b,\ldots,a[\rho_a]/b]$ (see previous line)
%	7	**int**	**int**	**int**	$a \bmod b$ $\{b>0; 0\le a \bmod b<b\}$
		vec	**int**	**int**	$[a[1] \bmod b,\ldots,a[\sigma_a] \bmod b]$
		mat	**int**	**int**	$[a[1]\ \%\ b,\ldots,a[\rho_a]\ \%\ b]$ (see previous line)

^	8	**int**	**int**	**int**	a^b
		mat	**int**	**mat**	$a^b \quad \{\rho_a = \kappa_a\}$
		pol	**int**	**pol**	a^b
		vec	**vec**	**vec**	$[a[1], \ldots, a[\sigma_a], b[1], \ldots, b[\sigma_b]]$ (concatenation)
		mat	**mat**	**mat**	$[a[1], \ldots, a[\rho_a], b[1], \ldots, b[\rho_b]]$ (vertical concatenation) $\{\kappa_a = \kappa_b\}$
X	9	**int**	**int**	**pol**	$aX^{[b]}$ (standing for aX_1^b)
		int	**vec**	**pol**	the term aX^b (standing for $aX_1^{b[1]} \cdots X_{\sigma_b}^{b[\sigma_b]}$)
		int	**mat**	**pol**	$\sum_{i=1}^{\rho_b} aX^{b[i]}$
X	10	**int**		**pol**	$1X^{[a]}$
		vec		**pol**	$1X^a$
		mat		**pol**	$\sum_{i=1}^{\rho_a} X^{a[i]}$
$<$	5	**int**	**int**	**int**	$a < b$
$<=$	5	**int**	**int**	**int**	$a \leq b$
$>$	5	**int**	**int**	**int**	$a > b$
$>=$	5	**int**	**int**	**int**	$a \geq b$
$==$	4	**int**	**int**	**int**	$a = b$
		vec	**vec**	**int**	$a = b$ (componentwise equality)
		mat	**mat**	**int**	$a = b$ (componentwise equality)
		pol	**pol**	**int**	$a = b$ (termwise equality)
$!=$	4	**int**	**int**	**int**	$a \neq b$
		vec	**vec**	**int**	$a \neq b$
		mat	**mat**	**int**	$a \neq b$
		pol	**pol**	**int**	$a \neq b$
||	1	**int**	**int**	**int**	if $a \neq 0$ then 1 else $b \neq 0$ (logical or)
&&	2	**int**	**int**	**int**	if $a = 0$ then 0 else $b \neq 0$ (logical and)
!	3	**int**		**int**	if $a = 0$ then 1 else 0 (logical not)

2.4. Using functions

2.4.1. Function call

A function call has the form

$\langle \text{name} \rangle (\langle \text{arg}_1 \rangle, \ldots, \langle \text{arg}_n \rangle)$

where $\langle \text{name} \rangle$ is the name of the function to be called, and $\langle \text{arg}_1 \rangle, \ldots, \langle \text{arg}_n \rangle$ are arbitrary expressions giving the actual arguments of the function; among the possibly numerous definitions for the given function name, the one is selected for which the types of the formal parameters match those of the actual arguments. To call a parameterless function, the name of the function may or may not be followed by an empty pair of parentheses; the former possibility looks like a variable, but is really different, since the function body will be executed only at the time of the call. (In fact it is also allowable to write an empty parentheses after a name that refers to a variable, but this appears to be needlessly misleading.) Whenever a function is called, its arguments are evaluated first.

2.4.2. Basic functions

There are a few built-in functions of non mathematical nature that supplement the built-in operators. These built-in functions cannot be redefined for the given argument types, although one may add user defined meanings for other types; the same is true for the built-in mathematical functions listed in Chapter 4. Again, we give these functions by means of a table.

function	parameter(s)	result type	meaning, comments
abs	**int** x	**int**	$\|x\|$; the absolute value
factor	**int** n	**vid**	prints a tentative factorisation of n; only prime factors up to 2^{15} are found.
size	**vec** v	**int**	the number of entries of v
null	**int** n	**vec**	a vector of length n with all entries 0
all_one	**int** n	**vec**	a vector of length n with all entries 1
n_rows	**mat** m	**int**	the number ρ_m of rows of m
n_cols	**mat** m	**int**	the number κ_m of columns of m
id	**int** n	**mat**	the $n \times n$ identity matrix
null	**int** n, m	**mat**	the $n \times m$ matrix with all entries 0
all_one	**int** n, m	**mat**	the $n \times m$ matrix with all entries 1
diag	**mat** m	**vec**	the main diagonal of m
vecmat	**mat** m	**vec**	concatenation of rows of m: $m[1]\hat{}m[2]\hat{}\cdots$
matvec	**vec** v; **int** n	**mat**	matrix with column size n and rows $[v[1], \ldots, v[n]]$, $[v[n+1], \ldots, v[2n]], \ldots$ $\{n$ divides $\sigma_v\}$
blockmat	**mat** a, b	**mat**	the block matrix $\left(\begin{smallmatrix} a & 0 \\ 0 & b \end{smallmatrix}\right)$
sort	**vec** v	**vec**	vector with same entries as v, but sorted into decreasing order
sort	**mat** m	**mat**	matrix with same rows as m, but sorted into the same order as polynomial exponents
redsetmat	**mat** m	**mat**	A reduced matrix, representing the same set of rows as m, but without duplicates. The rows are also reordered as in *sort*(m).
n_vars	**pol** p	**int**	the number ν_p of indeterminates of p
length	**pol** p	**int**	the number of terms of p
coef	**pol** p; **int** n	**int**	the coefficient of the n-th term of p
monom	**pol** p; **int** n	**vec**	the exponent of n-th term of p
poly_null	**int** n	**pol**	the zero polynomial in n indetermines
poly_one	**int** n	**pol**	the unit polynomial in n indetermines
compsize	**grp** g	**int**	the number of simple components of g
void	any x	**vid**	no result, useful to force void type, for instance to make types match between branches of conditional clause
print	any x	**vid**	print the value of x
error	**tex** t	**tex**	print text t and terminate; prompt for new command

2.5. Statements and clauses

We have treated the main ways of forming expressions; however, expressions usually do not suffice to perform complicated calculations, so we need basic actions and ways to combine them into larger programs. The basic actions are performed by *statements*, the larger structures built from them are called *clauses*. The distinction between expressions, statements and clauses is not absolute, however, since on one hand expressions are considered to be statements as well, and on the other hand clauses (which may very well yield values) are themselves expressions (and hence *a* *fortiori* statements). If a clause yields no value, then the clause is said to return void, and is of type **vid**.

We first treat assignment statements, which are the most important kind of statements, apart from expressions. Then we treat the clauses, of which there are three kinds: blocks, conditional clauses and loops. Finally we treat the remaining kinds of statements, namely the **break**, **return** and **setdefault** statements.

2.5.1. Assignment statements

Assignment statements have the effect of altering the value of a variable, and return void. They come in five forms.

$\langle \text{identifier} \rangle = \langle \text{expression} \rangle$

The execution of this statement consists of evaluating the expression (which may be of any type), and assigning its value to the variable denoted by the identifier. This statement may optionally be preceded by **loc**, in which case a new local variable is created at the current level, which will be denoted by the identifier, and which is initialised to the value of the $\langle \text{expression} \rangle$. For more details see Section 2.6.2.

$\langle \text{identifier} \rangle\, [\, \langle \text{expression}_1 \rangle \,] = \langle \text{expression}_2 \rangle$

Here $\langle \text{identifier} \rangle$ must denote a vector, matrix or polynomial variable, and correspondingly $\langle \text{expression}_2 \rangle$ must be of type integer, vector, or polynomial respectively, while $\langle \text{expression}_1 \rangle$ must be of type integer in all cases. Both expressions are evaluated, and the value of $\langle \text{expression}_2 \rangle$ replaces the entry of the vector variable respectively the row of the matrix variable or the term of the polynomial variable, whose index is the value of $\langle \text{expression}_1 \rangle$. In the case of a matrix or polynomial variable it is required that the yield of $\langle \text{expression}_2 \rangle$ has the same size as the the row or term replaced by it; in particular it may not be a polynomial of length > 1.

$\langle \text{identifier} \rangle\, [\, \langle \text{expression}_1 \rangle, \langle \text{expression}_2 \rangle \,] = \langle \text{expression}_3 \rangle$

Here $\langle \text{identifier} \rangle$ must denote a matrix variable, and all expressions must be of type integer; the value yielded by $\langle \text{expression}_3 \rangle$ replaces the entry of the matrix variable whose indices are the values yielded by $\langle \text{expression}_1 \rangle$ and $\langle \text{expression}_2 \rangle$.

$\langle \text{identifier} \rangle \mid \langle \text{expression}_1 \rangle = \langle \text{expression}_2 \rangle$

Here $\langle \text{identifier} \rangle$ must denote a polynomial variable, $\langle \text{expresion}_1 \rangle$ must be of type vector and $\langle \text{expression}_2 \rangle$ of type integer. The term of the polynomial is searched whose exponent coincides with the value of $\langle \text{expression}_1 \rangle$ (if none exists, a new such term with coefficient 0 is created), and its its coefficient is replaced by the value of $\langle \text{expression}_2 \rangle$.

$\langle \text{identifier} \rangle + = \langle \text{expression} \rangle$

This statement is equivalent to

$\langle \text{identifier} \rangle = \langle \text{identifier} \rangle + \langle \text{expression} \rangle$

but it is easier to write and in most cases more efficiently executed.

2.5.2. Series

Before we treat clauses, we must briefly mention *series*, which form part of all forms of clauses. A series is nothing more than a sequence of statements, separated by semicolons:

$\langle$ statement$_1$ $\rangle$; $\langle$ statement$_2$ $\rangle$; $\cdots$; $\langle$ statement$_n$ $\rangle$

When the series is executed, its statements are executed in order from left to right, and the value of $\langle$ statement$_n$ $\rangle$ (if any) becomes the value of the whole series (any values yielded by any of the other statements are cast away).

2.5.3. Blocks

A block is formed by enclosing a series in braces:

{ $\langle$ series $\rangle$ }

Since a block is an expression, this allows the value of a series to enter into larger expresssions. Furthermore, a block establishes a range for the definition of local variables, see also Section 2.6.2. Here is a rather silly example that shows both aspects of blocks:

$a = 2$; {**loc** $a = [6, 19, 10, 1, 14, 10]$; $a/2$} $+ a$

which returns the value $[3, 9, 5, 0, 7, 5, 2]$.

2.5.4. Conditional clauses

There are two forms of conditional clauses:

if $\langle$ expression $\rangle$ **then** $\langle$ series$_1$ $\rangle$ **else** $\langle$ series$_2$ $\rangle$ **fi**

and

if $\langle$ expression $\rangle$ **then** $\langle$ series$_1$ $\rangle$ **fi**

In each case $\langle$ expression $\rangle$ is evaluated first; if the (integer) value yielded is unequal to 0 then $\langle$ series$_1$ $\rangle$ is evaluated and its value becomes the value of the conditional clause, and otherwise $\langle$ series$_2$ $\rangle$ is evaluated if present and its value becomes that of the conditional clause. In the second form of the conditional expression, where $\langle$ series$_2$ $\rangle$ is absent, it is required that $\langle$ series$_1$ $\rangle$ has void type, so that no value is yielded either way.

2.5.5. Loop clauses

There are two main kinds of loop clauses: **while** loops and **for** loops, of which the latter kind has a few variants; all loop clauses are recognisable by the keywords **do**, and **od**. A while loop has the form

while $\langle$ expression $\rangle$ **do** $\langle$ series $\rangle$ **od**

When a while loop is executed, the $\langle$ expression $\rangle$ is first evaluated; if it yields 0 then the execution of the loop terminates, and otherwise the $\langle$ series $\rangle$ is executed, after which execution of the while loop resumes from the beginning. When the loop terminates, it returns the value of the last execution of its $\langle$ series $\rangle$, or void if the $\langle$ expression $\rangle$ had value 0 the first time it was evaluated.

There are three variants of the for loop, namely for looping over an interval of the integers, over the entries of a vector, and over the rows of a matrix. The first form is

for $\langle$ identifier $\rangle$ = $\langle$ expression$_1$ $\rangle$ **to** $\langle$ expression$_2$ $\rangle$ **do** $\langle$ series $\rangle$ **od**

The identifier denotes a fresh variable, local to this loop, which will disappear when the loop is terminated; call this the loop variable. First both expressions, which should be of type integer, are evaluated. The value of $\langle$ expression$_1$ $\rangle$ is assigned to the loop variable, and the value of $\langle$ expression$_2$ $\rangle$ is stored away for comparison; call it *limit*. Then the following sequence of operations is performed until the loop is terminated:

the value of the loop variable is compared with *limit*, and if it exceeds that value, the loop terminates; otherwise the ⟨ series ⟩ is evaluated and finally the loop variable is incremented by 1. Having terminated, the loop returns the value of the most recent evaluation of ⟨ series ⟩, or void if it was not evaluated even once (i.e., if the value of ⟨ expression_1 ⟩ exceeds *limit*). It is permitted—but not recommended—to assign to the loop variable within ⟨ series ⟩.

The second form of the loop clause is

for ⟨ identifier ⟩ **in** ⟨ expression ⟩ **do** ⟨ series ⟩ **od**

Here ⟨ expression ⟩ should yield a vector v, and again ⟨ identifier ⟩ denotes a loop variable local to this loop. The execution of this kind of loop is similar to that of the first kind, but rather than initialising, testing and incrementing the loop variable, the ⟨ series ⟩ is evaluated as many times as the size of v, and prior to the i-th evaluation, the value $v[i]$ is assigned to the loop variable. Again the value of the last execution of ⟨ series ⟩ determines the value of the loop clause itself. As an example, the sum of the entries of a vector can be computed as follows:

sum(**vec** *v*) = **loc** *s* = 0; **for** *entry* **in** *v* **do** *s* = *s* + *entry* **od**; *s*

The third form is analogous to the second, looping over the rows of a matrix rather than over the entries of a vector. Its form is

for ⟨ identifier ⟩ **row** ⟨ expression ⟩ **do** ⟨ series ⟩ **od**

Here ⟨ expression ⟩ should yield a matrix m, and again ⟨ identifier ⟩ denotes a loop variable local to this loop; in this case it is a vector variable. The only further difference with the previous form of the loop clause is that the number of times ⟨ series ⟩ is evaluated equals the row size of m, and prior to the i-th evaluation, the value $m[i]$, i.e., the i-th row of m, is assigned to the loop variable.

2.5.6. Break, return and setdefault

It is possible to exit a **while** or **for** loop before the termination conditions given in Section 2.5.5 are satisfied by executing a statement **break** contained somewhere in the ⟨ series ⟩ of the loop (but not in any loop contained in that ⟨ series ⟩). This is a statement of the form

break or **break** ⟨ expression ⟩

Executing **break** forces termination of the smallest enclosing loop; the value of ⟨ expression ⟩ if present becomes the value of the loop*. The following example defines a primality test using this feature.

* This is true for any ordinary use of **break**, but in fact the rule is a bit more complicated, since LiE completes the evaluation of any statement in the loop that is being evaluated at that point; this can only happen if some clause containing the **break** is being used as a proper subexpression of some statement (or expression). For instance in 'a = {**break** 5}' the value 5 is assigned to the variable a, instead of forming the result of the loop. The rule is that the value of **break** becomes that of the enclosing clause, and may be used to complete evaluation of the statement containing that clause; the value of that statement then moves outward to the enclosing clause, etc., until the value of the loop itself is determined.

prime(**int** *n*) = **loc** *v* = [2]; \
  for *i* = 3 **to** *n* **do if** *primetest*(*i*) **then** *v*+ = *i* **fi od**; *v*
primetest(**int** *k*) = **for** *n* **in** *v* **do if** *k* % *n* == 0 **then break** 0 **else** 1 **fi od**
prime(68)

which returns [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]. Note how *primetest* uses the local variable *v* of *prime*, which is possible according to the (dynamic) binding rules for variables; see Section 2.6.2.

The statement **return** is analogous to **break**, but it terminates the function currently being executed rather than the smallest enclosing loop; this may in fact also force termination of any loops within that function (but the converse is not true: **break** can only teminiate a loop within the current function). Its form is

return or **return** ⟨ expression ⟩

In the same fashoin as for **break**, the expression after **return** will determine the result of the function. The function *primetest* in the previous example could therefore also be written as

primetest(**int** *k*) = **for** *n* **in** *v* **do if** *k* % *n* == 0 **then return** 0 **fi od**; 1

The statement **setdefault** is not related to **break** or **return**; it is simply used to set or inspect an important system parameter, the *default group*. Its form is

setdefault or **setdefault** ⟨ expression ⟩

Many of the mathematical functions, which are described in Chapter 4 involve computation within some Lie group, or its root system or representation theory etc. These functions need to be told for which group they should do their computation, and by convention this group is passed as the final argument. For convenience however, since one often does a number of computations for the same group, one may define a default group, in which case it is allowable to omit this final argument; the default group will be implicitly assumed. To set the default group execute **setdefault** ⟨ expression ⟩ with ⟨ expression ⟩ yielding the desired group; to find out what the default group is currently, execute **setdefault** without parameters. For example, the commands

setdefault A_3; *worbit*([1, 1, 1])

will produce the same result as *worbit*([1, 1, 1], A_3), but it will also have set the default group to A_3, so that it can be omitted in further function calls.

2.6. User defined functions

We have already seen some simple examples of functions defined by the user; in this section we treat this subject in more detail.

Functions can only be defined on top level, i.e., not within function bodies. At the moment of definition of a function, it is only checked for syntactic correctness, and then effectively stored textually. Only at the time of the function call does the interpreter determine the types and values of the contained symbols (this makes it possible, for instance, to define a function that calls upon other functions that are yet to be specified, as long as these functions are defined before the first function is actually called). At the time the function is called, the interpreter checks that all variables and functions are used with consistent types, and only after this has been successfully done does the real execution start. Before the function is invoked, all of its arguments are computed; thereafter the function itself is executed.

Like for operators, there can be more than one meaning for a function, as long as they can be distinguished by the number and types of their parameters. It is for instance possible for the user to extend functions that are built into LiE to other types of values, as is demonstrated in Section 5.7. The name of a function can even be simultaneously used as a variable, but the uses of a name for a *parameterless* instance of a function and as a variable are mutually exclusive.

2.6.1. Function definition

A function definition consists of the function identifier followed by a list of formal parameters, an equals sign and the (possibly compound) statement that computes the result of the function (the latter may be as simple as a single expression). Function definitions can take two similar forms:

$\langle\,\text{name}\,\rangle = (\,\langle\,\text{type}\,\rangle\langle\,\text{variables}\,\rangle;\ \ \ldots;\ \ \langle\,\text{type}\,\rangle\langle\,\text{variables}\,\rangle\,) = \langle\,\text{series}\,\rangle$
$\langle\,\text{name}\,\rangle = (\,\langle\,\text{type}\,\rangle\langle\,\text{variables}\,\rangle;\ \ \ldots;\ \ \langle\,\text{type}\,\rangle\langle\,\text{variables}\,\rangle\,)\ \ \{\,\langle\,\text{series}\,\rangle\,\}$

where each ⟨ type ⟩ is one of **int**, **vec**, **mat**, **pol**, **grp** and **tex**, and each ⟨ variables ⟩ consists of one or more identifiers, separated by commas. The first form of the function definition is most convenient for simple functions, for instance when the function body consists of a single expression; the second form on the other hand is more suitable for large functions, especially since command prolongation up to the closing brace is guaranteed. The identifiers denote the parameters of the function, in order; each identifier in ⟨ variables ⟩ has the type specified by the preceding ⟨ type ⟩. The function parameters are considered as local variables, which are initialised during a call to the values of the arguments. Therefore they can be changed, but this has no effect on the values of variables outside the function (call by value). A parameterless function may be defined by writing an empty pair of parentheses, but unlike in calls the parentheses may not be omitted altogether, for then one would obtain an assignment rather than a function definition. Examples of function definitions are:

$f(\textbf{int}\ x) = 2 * x$
$f(\textbf{tex}\ a;\ \textbf{int}\ x, y;\ \textbf{tex}\ b) = print(a);\ print(x\hat{\ }y);\ print(b)$
$gcd(\textbf{int}\ x, y) = \textbf{if}\ y == 0\ \textbf{then}\ x\ \textbf{else}\ gcd(y, x\,\%\,y)\ \textbf{fi}$
hi() {*print*(`"How do you do?"`)}

Now the call $f(3)$ yields 6, while *f*(`"7^51 ="`, 7, 51, `"That's 44 digits"`) prints the three lines

```
7^51 =
1258925529853188502634196238398754544475874 3
That's 44 digits
```

and yields no value, *gcd*(51566870, 2371954630) yields 1990, and finally the response to *hi* is `How do you do?`. As an example of a slightly less trivial function definition, we present the following function that extends *gcd* above in the sense that it not only computes the value $d = gcd(x, y)$, but also determines integers k, l such that $d = kx + ly$, by means of the so-called "extended Euclidian algorithm". The result is encoded as a vector $[d, k, l]$.

```
extgcd(int x, y) \
{  loc m = [[x,1,0],[y,0,1]];
   # invariant: m[i,1] = xm[i,2] + ym[i,3] for i ∈ {1,2} #
   for i = 1 to 2 do if m[i,1] < 0 then m[i] = -m[i] fi od;
   while m[1,1]  # stop when smaller number becomes 0 #
   do loc q = m[2,1]/m[1,1];  m = [m[2] - q * m[1], m[1]] od;
   m[2]
}
```

2.6.2. Local variables and blocks

We have already encountered local variables when discussing assignments, block and function parameters. We now discuss these in more detail.

During execution, LiE maintains a hierarchy of levels for defining the scope of variables. Command execution always starts at the top level; variables defined on this level are global variables. Lower levels are created whenever the execution of a new series starts, and remain in existence until the execution of that series is completed. Here is a complete list of the series that correspond to separate levels:

- The series of a block, which is enclosed in curly braces '{' and '}'.
- The series between **then** and **else** (or **fi**) or between **else** and **fi**,
- The series between **do** and **od**,
- The body of a function,

An assignment of the form **loc** ⟨variable⟩ = ⟨expression⟩ introduces a new (initialised) local variable at the current level. The variable will cease to exist when this level disappears and LiE returns to a higher level. The range in which such a local variable can be accessed, extends from the statement following its **loc** assignment to the end of the series defining the current level; this is almost obvious from the fact that nothing can be accessed before it is created, but note that for instance

$a = 3$; **for** $i = 1$ **to** a **do** $print(a)$; **loc** $a = a + 1$; $print(a)$ **od**

will print the values 3, 4, 3, 4, 3, 4, since the first call of *print* always prints the global a (in fact no local a exists whenever this statement is executed).

When a variable is assigned to in an assignment without **loc**, or when it is used in an expression, it is first checked whether a variable of that name exists at the current or any higher level (in that order), which ends with checking if a global variable of that name exists. As soon as a matching variable is found, that variable is used; if no variable of that name is found at all, then if the variable is being assigned to, a new variable is created at the current level (as if the assignment were preceded by **loc**), and otherwise an error message is generated. As a consequence, it is not possible to create new global variables except from the top level. Furthermore, it is not allowed at lower levels to change the type of any variable: it is only allowed to change the value to another value of the same type.

Note that the variable identified by an identifier used non-locally within a function depends on the chain of active functions at the point of reference; situations in which such an identifier denotes different variables during the execution of a single command are even possible. Therefore use of **loc** is always recommended for intermediate results within functions. Note also that although the call-by-value rule excludes the possibility that a function when called by another one modifies values in the calling function by assignment to its own parameters, it *can* modify the local variables of the calling function by means of direct assignments to them that are not shielded by any **loc**.

2.6.3. Make and apply

To LiE, functions are not values in the sense that they could be assigned to variables, or passed to or returned from (other) functions. However, there is a number of built in operations, under the names **make** and a few variants of **apply** that do accept a function as one of their parameters, and that yield values computed using this parameter function.

The function that appears as an argument to **make** or **apply** should be user defined, and it is treated as a mathematical function, so it should not have side effects (i.e., external changes obtained by calling the function, other than the value yielded), as it is not defined in what way exactly the function is called.

There are a number of meanings for each of the operations, depending on the number and type of arguments supplied. To facilitate specification of these meanings we use the letter f throughout to denote the function parameter, and for the other parameters we use n, n' for integers, v, v', v'' for vectors and m for matrices.

The operation **make** is useful to tabulate a function f on certain sample values. The simplest case is to tabulate a function on the numbers $1, \ldots, n$. For a function $f\colon \mathbf{int} \to \mathbf{int}$, we have

$$\mathbf{make}(f, n) = [f(1), \ldots, f(n)],$$

in other words $\mathbf{make}(f, n)$ is a vector v of size n, with $v[i] = f(i)$ for each i. For example, with the definitions given in Section 2.6.1, $\mathbf{make}(f, 4)$ returns $[2, 4, 6, 8]$. It is also possible to tabulate the same function on explicitly given values, so again for a function $f\colon \mathbf{int} \to \mathbf{int}$, we have

$$\mathbf{make}(f, v) = [f(v[1]), \ldots, f(v[n]),$$

where n is the size of v, in other words $\mathbf{make}(f, v)$ is a vector v' of the same size as v, with $v'[i] = f(v[i])$ for each i. We give an example with the same f defined above: $\mathbf{make}(f, [47, 11, 30, -531, 425])$ returns $[94, 22, 60, -1062, 950]$.

Similar operation are available for functions of two integer arguments. So let $f\colon (\mathbf{int}, \mathbf{int}) \to \mathbf{int}$, then we have

$$\mathbf{make}(f, n, n') = \begin{pmatrix} f(1,1) & \cdots & f(1,n') \\ \vdots & & \vdots \\ f(n,1) & \cdots & f(n,n') \end{pmatrix},$$

in other words $\mathbf{make}(f, n, n')$ is an $n \times n'$ matrix m that satisfies $m[i, j] = f(i, j)$ for all applicable i, j. As an example

$$\mathbf{make}(gcd, 3, 7) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 & 1 & 1 & 2 \\ 1 & 1 & 3 & 1 & 1 & 3 & 1 \end{pmatrix}.$$

Again there is a variant to present arbitrary sample data to f, namely by providing two equal length vectors, where the first argument to f is taken from the first, and the second argument from the second vector. In this case only pairs of entries at matching positions are selected, so the result is a vector rather than a matrix. We have

$$\mathbf{make}(f, v, v') = [f(v[1], v'[1]), \ldots, f(v[n], v'[n])]$$

where n is the size of v and of v', in other words $\mathbf{make}(f, v, v')$ is a vector v'' of the same size as v and v', with $v''[i] = f(v[i], v'[i])$ for each i. As an example $\mathbf{make}(gcd, [3, 5, 8, 21, 91], [8, 10, 12, 14, 39])$ yields $[1, 5, 4, 7, 13]$.

The operations **iapply**, **vapply** and **mapply** are used to compute iterates (or powers) of the specified function. For convenience, define the notation $f^n(x)$ by

$$f^n(x) = \begin{cases} x & \text{if } n = 0 \\ f(f^{n-1}(x)) & \text{if } n > 0 \end{cases}$$

Here x can be an integer, vector or matrix as applicable for f. The corresponding cases have different names in LiE, however:

$$\begin{array}{lll} \textbf{iapply}(f,n,n') = f^n(n') & \text{where } f\colon \textbf{int} \to \textbf{int} \\ \textbf{vapply}(f,n,v) = f^n(v) & \text{where } f\colon \textbf{vec} \to \textbf{vec} \\ \textbf{mapply}(f,n,m) = f^n(m) & \text{where } f\colon \textbf{mat} \to \textbf{mat} \end{array}$$

As a simple example we have $\textbf{iapply}(f,4,3) = 48$ for the function f given above. For the case of $f\colon \textbf{int} \to \textbf{int}$ there is also a variant that accumulates all the intermediate values into a vector; we have

$$\textbf{vapply}(f,n,n') = [n', f(n'), f^2(n'), \ldots, f^n(n')],$$

in other words, $\textbf{vapply}(f,n,n')$ is a vector v of length $n+1$, with $v[1] = n'$ and $v[i] = f(v[i-1])$ for $2 \le i \le n+1$. For example, still using the doubling function f from above, we have $\textbf{vapply}(f,4,3) = [3,6,12,24,48]$. A final variant of **vapply** uses a function $f\colon \textbf{vec} \to \textbf{int}$ to incrementally build up a vector; it can be formulated in terms of the first instance of **vapply**:

$$\textbf{vapply}(f,n,v) = \textbf{vapply}(F,n,v) \qquad \text{where } F(v) = v + f(v)$$

Here F is a function that extends a vector with a new entry computed by f from that vector. A typical example is the following procedure to compute Fibonacci numbers. First a function f is defined to compute the next Fibonacci number from a vector of preceding ones:

$$f(\textbf{vec}\ v) = \ \textbf{loc}\ s = \mathit{size}(v);\ \ v[s-1] + v[s]$$

With this function we compute the first 12 Fibonacci numbers in the sequence starting with $[1,1]$ by calling $\textbf{vapply}(f,10,[1,1]) = [1,1,2,3,5,8,13,21,34,55,89,144]$. Note that LiE decides whether to take the second or third instance of **vapply** depending on the result type of f when applied to a vector.

2.7. Global commands

In addition to the commands mentioned above, there are a number of commands that do not really form a part of the language of the interpreter, but allow the user a number of controls over the LiE session. All these commands can only be invoked from top level.

2.7.1. File management

It is possible to collect a number of commands to the LiE interpreter in a file and then execute these commands as if typed from the keyboard. If these commands are contained in the file ⟨ name ⟩, then execution of the commands can be invoked by the command '`read` ⟨ name ⟩'. The file can also be edited during the LiE session by giving the command '`edit` ⟨ name ⟩'; after editing is finished, the resulting file is directly read into LiE as if the `read` command were given. The editor which is invoked is either the standard editor of your machine, or, if you are in a UNIX environment and the shell variable `$EDITOR` has been set, the editor named by that variable. The command `edit` can also be used without a filename argument, in which case the same file is edited as in the previous `edit` command. The file named '`initfile`', if present in the directory from which LiE is invoked, will be read upon entrance of the program LiE, before the first prompt appears; the same file will also be used when no filename is supplied in the first `edit` commmand of a session.

To save the user defined functions of a particular session, execute the command '`write` ⟨ name ⟩'. As a result, these functions are written in the file ⟨ name ⟩. See also the command '`on monitor`' below.

2.7.2. Information retrieval

Information about a function, operator or a reserved word (like **for**) can be obtained by typing '`?` ⟨ topic ⟩' (you may also use '`help`' as a synonym for '`?`'). A list of the reserved words can be obtained by typing '`?index`'. Information produced by '`?`' (or '`help`') can also be written on a file by typing '`?` ⟨ topic ⟩ > ⟨ filename ⟩', or appended to an existing file by '`?` ⟨ topic ⟩ >> ⟨ filename ⟩',

Information about a mathematical term can be obtained by giving the command '`learn` ⟨ term ⟩'. For example '`learn lie group`' will give all available information on the term 'lie group' and on on any terms containing that string (this won't work unless you type lower case letters, apologies to Sophus Lie). A list of the documented terms can be obtained by entering '`learn index`'.

2.7.3. Memory management

Memory management is performed automatically, and should be of no concern to the user. At certain points, LiE will deem it advisable to reduce the amount of memory in use, and will do so by invoking the *garbage collector*, which attempts to locate and free objects that are no longer accessible to the user. Although this is generally done automatically at convenient points in the calculation, it is also possible to explicitly call the garbage collector by the function *gcol*, and it is also possible—by stating `off gc`—to (temporarily) inhibit garbage collection at points where one knows that there will be no memory to free anyway, see Section 2.7.4. To monitor the amount of memory in use, the function *used* provides the number of variables and functions in use at this point. There is no way to explicitly remove a global variable from LiE's tables, but by assigning 0 to the variable, most relevant resources occupied by the variable are freed.

2.7.4. System parameters

There is a number of system parameters, which may be set and altered by the user. The command to do this has the form '`on` ⟨ feature ⟩' or '`off` ⟨ feature ⟩'; the various features are given in th follwing table:

feature	default state	'`effect of`' non-default setting
`bigint`	on	'`off bigint`' banishes arbitrary length integers
`lprint`	on	'`off lprint`' prints vectors, matrices and polynomials in rectanglular from
`monitor`	off	'`on monitor`' writes all output to the file '`monfil`' in LiE's start-up directory, as well as to the screen,
`prompt`	on	'`off prompt`' suppresses the prompt character '`>`'
`runtime`	off	'`on runtime`' shows the amount of time spent executing each command, after printing its result
`gc`	on	'`off gc`' inhibits garbage collection,

The effect of '`off lprint`' on vectors and matrices is only slight: commas are replaced by spaces, and in case of matrices the square brackets bordering the rows are repleced by verical bars. For polynomials however the difference is significant: the terms are listed vertically by printing the exponents as rows of a matrix, with the coefficient of the term preceding vertical bar at the left of the matrix. The running time shown by '`on runtime`' is divided into 'user' time, spent on actual computation, and 'system' time spent on operating system services.

The system parameter that determines the ordering of terms in a polynomial (and of the rows in a matrix in calls of *sort* and *redsetmat*) has four possible values, and requires a slightly different form of the '`on`' command (while the '`off`' command is not used in this context):

`on + lex`	select lexicographic ordering
`on - lex`	select inverse lexicographic ordering
`on + degree`	select total degree ordering
`on - degree`	select inverse total degree ordering

Two more system parameters, which determine the amount of memory that LiE allocates for representing programs and data, take a numeric value, and are set by the commands

`on` *n* `maxnodes`	set maximum number of nodes (for programs) to *n*
`on` *n* `maxptrs`	set maximum number of objects to *n*

The current values of all the system parameters can be obtained by giving the '`on`' command without parameters (in this case the '`off`' command is synonymous to '`on`').

Finally, we repeat that the abort character ⟨control⟩`C` terminates the currently running command.

LiE Manual
Chapter 3. TERMINOLOGY

In LiE, various mathematical notions are encoded by means of a limited number of different types (*viz.* integer, vector, matrix, polynomial), and it is important to now how the mathematical notions and the concrete objects manipulated by LiE correspond. It is the purpose of the current chapter helps to explain these correspondences. To this end a large part of this chapter is dedicated to listing the names of mathematical notions that are representable in LiE, with an indicatation of how can be represented by LiE objects. For example, a *root* of a semisimple Lie group g of rank r may be represented by a vector $v = [v_1, \ldots, v_r]$ such that the given root is equal to $\sum_{i=1}^{r} v_i\alpha_i \in \bigoplus_{i=1}^{r} \mathbf{Z}\alpha_i$, where the α_i (for $1 \leq i \leq r$) are the fundamental roots of the root system of g.

It may be clear from this very example that some theoretical background is required in order to explain these things. We do not intend to give a comprehensive introduction to the subject here (for this one may consult standard textbooks, a number of which can be found in Chapter 7), but we shall try to give the basic definitions and properties that are relevant to understanding the mathematical functions present in LiE. The remainder of this chapter is divided into a number of sections, each one treating one of the follwing subjects: Lie groups and algebras, roots and weights, Weyl groups and their action, representations of Lie groups, and the symmetric groups and related matters (the same subdivision is used in Chapter 4 in which the mathematical functions built into LiE are discussed, and in the help system provided by LiE). At the end of each section an alphabetic listing of the relevant terms related to the subject is given for reference, with explananations (if you are unsure under which subject a term is classified, the index gives references to all terms). We start with listing the different ways in which several types of LiE objects may be interpreted in general.

Matrix A matrix can either stand for a linear transformation (acting by right-multiplication on row vectors) for a set of vectors, in which case each row of the matrix represents a vector in the set, or in a special way such as for a character table. For instance, a matrix representing a set of roots will be termed a *root matrix*. See also *character matrix*, *orbit matrix*, and *restriction matrix*.

Polynomial A polynomial may either stand for itself (i.e., for a Laurent polynomial), or it may encode a set of vectors with multiplicities. In the latter case each term represents the occurrence of its exponent in the indicated set (where it is usually interpreted as a weight), occurring with multiplicity equal to the coefficient of the term. See also *decomposition polynomial* and *multiplicity polynomial*.

Vector A vector may represent an element of a vector space (or strictly speaking rather of a free $\mathbf{Z}$-module, since its entries must be integral), such as the weight space, or it may just be interpreted as a set or sequence of integers. In the former case it is always to be interpreted as a row-vector, so that matrices are to be applied from the right. In either case there are a further distinctions as to how the vector is to be interpreted. See also *root vector*, *weight vector Weyl word*, *partition* and *toral element*.

3.1. Lie groups and algebras

As the textbooks say, Lie groups are groups that also have the structure of a (real or complex) differentiable manifold, such that the maps of multiplication and inversion are differentiable maps. This definition however is not the most useful viewpoint when we consider Lie groups as treated in LiE: the differentiable structure is beyond the scope of LiE's computations, and the package only rarely deals with individual elements of Lie groups. Moreover, LiE only deals with a particularly well behaved subclass of Lie groups, namely the connected reductive complex Lie groups. This class of groups the semisimple Lie groups, but also important non-semisimple groups, such as $GL(n, \mathbf{C})$ (the group of all invertible $n \times n$-matrices). The chosen class is quite convenient, mainly for two reasons: the groups have a clearly structured classification, as well as a pleasing representation theory.

By the classification of the connected reductive complex groups (cf. [Bourb 1975]), each such Lie group is the homomorphic image of a direct product of a simply connected semisimple complex group and a complex torus (i.e., a direct product of copies of $\mathbf{C}^*$), where the homomorphism has a finite kernel, which is contained in the center. The semisimple factor in the product may be reconstructed up to isomorphy as the universal cover of the commutator subgroup, and the torus factor as the identity component of the center. Every simply connected semisimple group in its turn is a direct product of simply connected simple groups. Each of the latter groups is isomorphic either to one of the classical groups $SL(n, \mathbf{C})$ (for $n \geq 2$; the Special Linear group, consisting of all $n \times n$ matrices with determinant 1), $Spin(n, \mathbf{C})$ (for $n \geq 5$; the Spin group, covering the Orthogonal group: the group of all invertible $n \times n$ matrices m with $m^{-1} = m^\top$), $Sp(2n, \mathbf{C})$ (for $n \geq 3$, the Symplectic group, consisting of all invertible $2n \times 2n$ matrices m with $m^{-1} = jm^\top j^{-1}$ for a fixed invertible antisymmetric matrix j), or to one of the exceptional groups, which have types E_6, E_7, E_8, F_4, and G_2.

The groups directly representable in LiE are the complex Lie groups which are a direct product of simply connected simple groups and a central torus (i.e., groups that do not need the homomorphism with finite central kernel). The type of such a group is formed by concatenating the types of the individual factors, where T_n is used to denote an n-dimensional torus. We shall occasionally use a type indication to stand for the group of that type itself. Since any (not necessarily simply) connected reductive complex group g is the quotient of a simply connected reductive group $\hat{g}$ by a finite central subgroup, g can be described by specifying the central elements of $\hat{g}$ that are in the kernel of the canonical morphism $\hat{g} \to g$. For example, if $g = GL(2, \mathbf{C})$, then $\hat{g}$ can be taken to be of type A_1T_1, the direct product of $SL(2, \mathbf{C}) = A_1$ and a 1-dimensional torus $\mathbf{C}^* = T_1$, and the canonical surjective morphism $\hat{g} \to g$ has kernel $\{(\mathbf{1}, \mathbf{1}), (-\mathbf{1}, -\mathbf{1})\} \subset A_1T_1$, where $-\mathbf{1} \in A_1$ and $-\mathbf{1} \in T_1$ stand for the central elements minus the identity in the respective groups. We shall assume from now on that g is simply connected, and that it is the direct product of simply connected simple groups, together forming the so-called *semisimple part* g' of g, and a torus S, the so-called central torus of g,

Any Lie group g contains subgroups that are isomorphic to a (complex) torus, and are moreover maximal (with respect to inclusion) for this property; such a subgroup is called a *maximal torus*. All maximal tori are conjugate in g, so we may fix an arbitrary maximal torus in g and call it T. Then T is the direct product of the central torus S and a maximal torus T' of the semisimple part, (which in turn is the product of maximal tori of the simple components). The *Lie rank* of g is the dimension of T, which we shall denote by r; the *semisimple Lie rank* of g is the Lie rank of g', we shall denote it by s.

Much of the structure of a Lie group can be deduced from the structure of a *Lie algebra* which it induces on the tangent space to the group taken at the identity element (in particular, any finite dimensional representation of one of them leads to a similar representation of the other), and indeed much of the theory of Lie groups is derived by studying the representation of the Lie group on its Lie algebra (by conjugation). In LiE the point of view of Lie algebras is usually not stressed, but many of the computations may be interpreted for Lie algebras as well as for Lie groups.

Central torus Each simply connected reductive Lie group g (the groups LiE deals with) splits as a direct product of a semisimple group (its semisimple part) and a torus; the latter torus which (contrary to the maximal torus of the semisimple part) lies in the center of g is called the central torus of g.

Diagram The (Dynkin) diagram of a semisimple Lie group is a graph indicating the isomorphy type of the group; the number of vertices is equal to the (semisimple) Lie rank, and the number of connected components of the diagram is equal to the number of simple factors of the group. The vertices are labeled with positive integer numbers, following the conventions of [Bourb 1975]. The diagram represents the information contained in the *Cartan matrix* of the group in a compact form.

Fundamental Lie subgroup A closed subgroup h of a Lie group g is called fundamental if it contains a maximal torus of g. If h contains T and is reductive, it is determined by the set of roots in the root system Φ of g that are also roots of h; these form a *closed subsystem* of roots.

General Linear group The group of all invertible linear transformations of a vector space V is called the general linear group of V, written $GL(V)$. Up to isomorphism this depends only on $n = \dim V$, and this group is also written as $GL(n, \mathbf{C})$ (assuming the vector space is over $\mathbf{C}$). A Lie group homomorphism of some Lie group to $GL(V)$ is called a representation of that Lie group on the vector spave V. See also *special linear group.*

Lie group A group is called a Lie group if its underlying set is a differentiable variety, and the multiplication and inversion maps are differentiable. The group is called complex, connected, simply connected, etc., if the variety is respectively complex, connected, simply connected, etc. Each *reductive* complex Lie group is an algebraic group and the representation theory can be dealt with in an entirely algebraic manner. See [Serre 1987].

Lie algebra A finite-dimensional vector space V supplied with a bilinear operation $[\cdot,\cdot]: V \times V \to V$ satisfying $[x, y] = -[y, x]$ and $[[x, y], z] + [[y, z], x] + [[z, x], y] = 0$ for all $x, y, z \in V$ (anti-commutativity and the Jacobi identity, respectively) is called a Lie algebra. Every Lie group defines a Lie algebra structure on the tangent space to the group at the identity element. Although Lie algebras play no explicit rôle in this package, the representation theory of simply connected reductive complex Lie groups which LiE deals with coincides with the representation theory of reductive Lie algebras over $\mathbf{C}$, see [Hum 1972]. See also [Jac 1962]

Maximal torus A *torus* that is not properly contained in any other torus within g is called a maximal torus of g. If g is a reductive Lie group, such tori exist and any two are conjugate. In LiE, we always assume a fixed maximal torus T of g to be chosen, and *weights* and *roots* are defined with respect to T.

Reductive group A group is reductive if each of its finite dimensional representations decomposes into a direct sum of irreducible representations. A connected reductive complex Lie group g is isomorphic to the quotient of the direct product of a semisimple group and a torus by a finite central subgroup. An example is the general linear group $GL(n, \mathbf{C})$. The (images of) the semisimple factor and the torus can be found as the commutator subgroup g' of g and the *central torus* of g respectively. In LiE, the type *group* always refers to a simply connected reductive complex Lie group (so no quotient is involved).

Semisimple element All conjugates of elements of the torus T are called semisimple elements (not to be confused with the term semisimple for groups); in any representation of g they correspond to diagonalisable transformations. Hence each conjugacy class of semisimple elements has representatives in T, and some elements of T namely those of finite order, can be represented in LiE; see below under *toral element*.

Semisimple group A reductive Lie group is called semisimple if it contains no non-trivial central torus. Note that a non-trivial semisimple group necessarily contains non-semisimple elements.

Special Linear group For a vector space M the special linear group $SL(M)$ is defined as the Lie subgroup of the *General Linear group* $GL(M)$ of all transformations with determinant equal to 1.

Torus A group which is isomorphic to $(\mathbf{C}^*)^n$ for some n is called a torus (plural: tori); it is a reductive Lie group of dimension n. Any subgroup of a Lie group g all of whose elements are semisimple is a torus, called a torus of g. Every torus of g is contained in a maximal torus, and every maximal torus is conjugate to T, the fixed maximal torus. See also *semisimple element*. A fundamental property of a torus is that all of its irreducible representations are 1-dimensional. Since in such a representation of T each element acts as a scalar, the representation is essentially given by an algebraic group morphism $T \to \mathbf{C}^*$, a so-called weight. Any representation of g may be restricted to a representation of T, and as such decomposed into 1-dimensional representations. The resulting formal sum of weights is called the (formal) character of the representation with respect to T.

3.2. Roots and weights

Consider the set $\Lambda(T)$ of group morphisms $T \to \mathbf{C}^*$ (or equivalently, of 1-dimensional T-modules); its elements are called *weights*. Weights may be composed in a natural way by multiplication as $\mathbf{C}^*$-valued functions, which makes $\Lambda(T)$ into an Abelian group. We use an additive notation for this group, and it is therefore convenient to denote the image of some $t \in T$ under weight $\lambda \in \Lambda(T)$ by t^λ, so that we have $t^{\lambda+\mu} = t^\lambda t^\mu$. As an Abelian group, $\Lambda(T)$ is isomorphic to $\mathbf{Z}^r$; moreover there is a natural $\mathbf{Z}$-linear action on $\Lambda(T)$ of the finite group $W = N_g(T)/T$, the *Weyl group* of g (with respect to T). The group $\Lambda(T)$ naturally decomposes into a direct sum $\Lambda(S) \oplus \Lambda(T')$; the subgroup $\Lambda(S)$ is pointwise fixed by W. The group T is diagonalisable in any g-representation. In other words, if M is a g-module, then the restriction of M to T is a direct sum of 1-dimensional T-modules, and therefore described by a set of weights (with multiplicities). The *adjoint representation* of g is its representation on the Lie algebra of g, which (as a set) is the tangent space to g at the identity element $\mathbf{1}$. The set of nonzero weights of T occurring in the adjoint representation is called the *root system* of g, and (often) denoted by Φ. The elements of Φ, the so-called *roots*, span the sublattice of $\Lambda(T')$ of finite index, known as the *root lattice*.

There is a non-degenerate W-invariant inner product on the root lattice $\mathbf{Z}\Phi$; it is unique up to a scalar factor for each simple factor of g, and can be chosen to take values in $\mathbf{Z}$. We choose such an inner product, and extend it to a bilinear symmetric positive definite form $(\cdot,\cdot)$ on $\Lambda(T)$ in such a way that $\Lambda(S)$ is perpendicular to $\Lambda(T')$, and the restriction to the former has an orthonormal basis. The reflections in W (acting on the weight lattice), are precisely the orthogonal reflections in the hyperplanes perpendicular to the roots (a pair of opposite roots giving rise to the same reflection).

Embedding $\Lambda(T')$ in a real vector space, we choose (and fix) a hyperplane H through the origin, but not through any root, and a half space with respect to H, which we shall call the 'positive half-space'. Then there is a unique system of *fundamental roots*, i.e., a set $\{\alpha_1,\dots,\alpha_s\} \subset \Phi$ of s linearly independent roots such that any root β is an integral linear combination of the α_i, and the non-zero coefficients are either all positive or all negative, according as β lies in the positive or negative half-space; we accordingly call β a positive or negative root. We have $(\alpha_i,\alpha_j) \leq 0$ for $i \neq j$. Apart from determining a choice of a set of positive roots, we shall make no use of the hyperplane H and the positive half-space.

We define a partial ordering of weights: for weights v, v' we write $v' \prec v$ if $v - v'$ is a linear combination of the fundamental roots with non-negative integral coefficients; we say that v' *lies under* a weight v, and that v is *higher than* v' (so by construction all positive roots are higher than 0, which in its turn is higher than all negative roots). Note that v and v' can only be comparable with respect to $\prec$ if they lie in the same coset of the root lattice; in particular any set of weights that has a highest element is contained in a single such coset.

Any root α defines a linear form $\langle\cdot,\alpha\rangle$ on $\Lambda(T)$ defined by $\langle x,\alpha\rangle = \frac{2(x,\alpha)}{(\alpha,\alpha)}$, which value is independent of the scalar involved in the choice of the inner product, and moreover is always *integral*. In fact there exist $\omega_1,\dots,\omega_s$ in $\Lambda(T')$ that form a 'dual basis' to the linear forms $\langle\cdot,\alpha_1\rangle,\dots,\langle\cdot,\alpha_s\rangle$, i.e., which are such that $\langle\omega_i,\alpha_j\rangle = \delta_{i,j}$ for all $1 \leq i,j \leq s$; from this it follows that the ω_i form a $\mathbf{Z}$-basis of $\Lambda(T')$. We extend $\omega_1,\dots,\omega_s$ by an orthonormal basis $\omega_{s+1},\dots,\omega_r$ of $\Lambda(S)$ to a basis of $\Lambda(T)$, called the basis of *fundamental weights*. Note that the image of a weight x under reflection in the hyperplane perpendicular to a root α is given by $x - \langle x,\alpha\rangle\alpha$, and therefore lies in the same coset of the root lattice as x.

Cartan matrix The matrix $\big(\langle\alpha_i,\alpha_j\rangle\big)_{1\leq i,j\leq s}$ is called the Cartan matrix (of the semisimple part) of g; its rows express the fundamental roots on the basis of fundamental weights.

Cartan type The Cartan type of a *closed subsystem* Ψ of roots of Φ is the type of the semisimple group h such that Ψ is isomorphic to the root system of h.

Closed subsystem Given a root system Φ, a closed subsystem is a subset Ψ that is itself a root system, and has the property that whenever $\alpha+\beta \in \Phi$ for α, $\beta \in \Psi$ then $\alpha+\beta \in \Psi$. If Φ is the root system of g, then every closed subsystem corresponds to a *fundamental Lie subgroup* of g.

Fundamental reflection For a chosen set of fundamental roots $\alpha_1,\dots,\alpha_s$, the reflections in the hyperplanes perpendicular to these roots are called fundamental reflections; they are often denoted by $r_1, \ldots, r_s$. These reflections generate the —termWeyl group.

Fundamental root It is often assumed that a subset of the roots has been chosen as the set of fundamental roots, and are then denoted by $\alpha_1, \ldots, \alpha_s$; this set must form a basis of the root lattice such that any root can be expressed as a linear combination of them with either all positive or all negative integer coefficients. This is the basis on which *root vectors* are expressed. The function *inprod* gives a W-invariant inner product for weights on this basis.

Fundamental weight For a chosen set of fundamental roots there is a basis of the *weight lattice* consisting of weights $\omega_1, \ldots, \omega_r$ such that $\langle \omega_i, \alpha_j \rangle = \delta_{i,j}$ for all $i, j \in \{1, \ldots, s\}$; these weights are called the fundamental weights. It is this basis on which *weight vector* are expressed.

Highest root This is the maximum of the set of roots with respect to the partial ordering '$\prec$' (see above). It is the *highest weight* of the adjoint representation.

Levi subgroup Any subset of the set of fundamental roots determines a *closed subsystem* (of which it is a basis fundamental roots) of the root system, and the semisimple part of the fundamental Lie subgroup corresponding to this subsystem is called a Levi subgroup of g. The Dynkin diagrams of the Levi subgroups of g are therefore obtained by taking subsets of nodes of the diagram of g and retaining the edges between elements of the subset.

One parameter subgroup Any 1-dimensional subtorus h of T is called a one parameter subgroup; there is a group isomorphism $\phi\colon \mathbf{C}^* \to h$. Such one parameters subgroups may be represented in the following way, which is very similar to the representation of *toral elements.* For $1 \le i \le r$ we have a group homomorphism $z \mapsto \phi(z)^{\omega_i}$ from $\mathbf{C}^*$ to $\mathbf{C}^*$; this homomorphism is equal to some map $z \mapsto z^{a_i}$ for $a_i \in \mathbf{Z}$. The one parameter subgroup h is now represented by the vector $[a_1, \ldots, a_r, 0]$, where the final 0 serves to distinguish it from toral elements, which are valid in the same positions where one parameter subgroups may be used (e.g., as parameter to *centroots*). The integers $a_1, \ldots, a_r$ should not all have a non-trivial factor in common, because the morphism ϕ would then fail to be injective. Any toral element obtained by substituting some number d for the final zero lies in h (it is $\phi(\zeta)$ for $\zeta = e^{2\pi \mathrm{i}/d}$). The restriction matrix of h is obtained by arranging the a_i (for $i = 1, 2, \ldots, r$) vertically into a one-column matrix.

Positive root A root that can be expressed as a linear combination of fundamental roots with non-negative coefficients is called a positive root. For every root α exactly one of $\{\alpha, -\alpha\}$ is positive.

Root A non-zero weight for the *adjoint representation* of g is called a root of g. For each root the orthogonal reflection in the hyperplane perpendicular to it preserves the weight lattice.

Root lattice The sublattice of the *weight lattice* generated by the roots of g is called the root lattice. For semisimple groups the root lattice has finite index in the weight lattice; for simple groups of type A_n, B_n, C_n, D_n, E_n, F_4 and G_2 this index is $n+1$, 2, 2, 4, $9-n$, 1 and 1 respectively. The *fundamental roots* form a basis of the root lattice, and the elements of the root lattice are *root vectors.* See also *weight.*

Root matrix A root matrix is a matrix whose rows specify a set of roots, represented as root vectros. Root matrices may be used to denote subsystems of the root sysytem of g.

Root system The set of all roots is called the root system of g. It is usually denoted by Φ.

Root vector When an elemnt of the root lattice is represented by its coefficients on the basis consisting of the fundamental roots $\alpha_1, \ldots, \alpha_s$, the result is called a root vector. So a root vector has as size the semisimple rank of the group, and such a vector $v = [v_1, \ldots, v_s]$ is interpreted as the sum $\sum_{i=1}^{s} v_i \alpha_i$.

Toral element To describe elements of T we can use the fundamental weights ω_i. Recall that weights are in fact mappings $T \to \mathbf{C}^*$, and a weight λ can therefore be evaluated at an element $t \in T$, the resulting value be written t^λ; the set of fundamental weights form a complete set of coordinates in the sense that any element $t \in T$ in uniquely determined by the values t^{ω_i} for $i = 1, \ldots, r$. Since LiE cannot represent arbitrary complex numbers, it explicitly deals only with torus elements of finite order, i.e., for which all t^{ω_i} are roots of unity. To this end, a vector $[a_1, \ldots, a_r, n]$ in LiE may represent the element $t \in T$ for which $t^{\omega_i} = e^{2\pi \mathrm{i} a_i / n} = \zeta^{a_i}$ for $i = 1, \ldots, r$, where $\zeta = e^{2\pi \mathrm{i}/n}$ is a canonical n-th root of unity. See also *one parameter subgroup*. Since this is not the usual presentation of a toral element in a Lie group like $GL(n, \mathbf{C})$ (namely by the diagonal entries occurring when the element is diagonalised), an example is given in Chapter 5 of how to transform from one presentation to another.

Weight A weight with respect to a torus T is an algebraic group morphism $T \to \mathbf{C}^*$; it describes a 1-dimensional representation of T. These arise in the decomposition of the restriction to T of representations of g, in which case they are called the weights of the g-representation with respect to T. The set $\Lambda(T)$ of weights is an Abelian group, where the group operation is multiplication of weights as C^*-valued functions (which corresponds to the tensor product of 1-dimensional T-representations); this is written additively, and we consequently use the exponential notation t^λ to indicate application of a weight λ to $t \in T$, so that we have $t^{\lambda+\mu} = t^\lambda t^\mu$. The *fundamental weights* span the weight lattice as a free $\mathbf{Z}$-module; expressing a weight on this basis we obtain a so-called weight vector.

Weight lattice The set $\Lambda(T)$ of all weights of g with respect to T is called the weight lattice. The addition defined for weights makes $\Lambda(T)$ into an Abelian group isomorphic to $\mathbf{Z}^r$.

Weight vector When a vector is represented by its coefficients on the basis consisting of the fundamental weights $\omega_1, \ldots \omega_r$ the result is called a weight vector. So a weight vector $v = [v_1, \ldots, v_s]$ is interpreted as the sum $\sum_{i=1}^{r} v_i \omega_i$.

3.3. The Weyl group and its action

Recall that the Weyl group W is defined as the quotient of the normaliser in G of T by T (which is its own centraliser). If g is a reductive group, its Weyl group is the same as the Weyl group of its semisimple part. By construction W has a faithful action by conjugation on T, which induces an action on $\Lambda(T)$; often we will identify W with the corresponding set of transformations of $\Lambda(T)$. A *fundamental domain* for this action is the set $\Lambda^+(T)$ of weights of the form $\sum_{i=1}^{r} a_i \omega_i$ with $a_i \geq 0$ for all $i \leq s$, which means that any weight can be transformed by W into a unique unique element of $\Lambda^+(T)$; the set $\Lambda^+(T)$ is usually referred to as the *Weyl chamber*. A weight is called *dominant* if it lies in $\Lambda^+(T)$. There is no direct relation between dominance and the ordering '$\prec$' (for instance for all positive roots α we have $0 \prec \alpha$, but usually very few (often only one) of these positive roots are dominant); however we have the following fact: the unique dominant weight in any W-orbit is also the highest element of that orbit.

The group W is generated by the *fundamental reflections*, i.e., the orthogonal reflections in the hyperplanes perpendicular to the fundamental roots; the reflection corresponding to α_i is denoted r_i. As we have seen, $xr_i = x - \langle x, \alpha_i \rangle \alpha_i$, where we follow the convention, used consistently throughout LiE, of writing linear transformations (and their matrices) to the *right* of the vector they operate upon. For any pair of distinct i, j with $1 \le i, j \le s$, the product $r_i r_j$ fixes the space perpendicular to both α_i and α_j, and induces a rotation in the plane spanned by α_i and α_j. The angle of rotation is $2\pi/m_{ij}$, where m_{ij} is the order of $r_i r_j$ (i.e., the least number $m > 0$ such that $(r_i r_j)^m = 1$). Consequently we have $(\alpha_i, \alpha_j) = -\sqrt{(\alpha_i, \alpha_i)(\alpha_j, \alpha_j)} \cos(\pi/m_{ij})$, which holds also in the case $i = j$, since $m_{ii} = 1$. Then W has the following abstract presentation:

$$W = \langle\, r_1, \ldots, r_s \mid (r_i r_j)^{m_{ij}} = 1 \,\rangle.$$

This presentation of W in terms of generators and relations shows that W is a Coxeter group. Elements of W can be represented in LiE both as products of fundamental reflections (see *Weyl word* below) and as $r \times r$ matrices. There are convenient ways to switch from one representation to another.

Coxeter matrix A Coxeter matrix is a symmetric matrix $m = (m_{i,j})_{1 \le i,j \le s}$ with positive integer coefficients such that $m_{i,j} = 1$ if and only if $i = j$. Such a matrix is used to define a Coxeter group: the group presented by $\langle\, g_1, \ldots, g_s \mid (g_i g_j)^{m_{i,j}} = 1 \,\rangle$. The presentation of W given above shows that every Weyl group is a Coxeter group, with Coxeter matrix given by $m_{i,j} = \mathrm{order}(r_i r_j)$.

Distinguished coset representative Within the Weyl group W we may consider left-, right- and double cosets with respect to a subgroup (or in the case of double cosets, two subgroups) generated by fundamental reflections; in each case the unique element of smallest *length* in its coset is called the distinguished coset representative.

Dominant weight A weight whose inner products with all fundamental roots are non-negative is called dominant. Therefore, if the weight is written on the basis of the *fundamental weights* $\omega_1, \ldots, \omega_r$, then the first s coefficients (corresponding to the semisimple part of the weight lattice $\Lambda(T)$) are non-negative.

Exponents The exponents of a Lie group g form a sequence of numbers $e_1, \ldots, e_r$, where r is the Lie rank of g, such that the polynomial $\sum_{w \in W} X^{l(w)}$, where l denotes the length function on the Weyl group, decomposes as a product $\prod_{i=1}^{r} \sum_{j=0}^{e_i} X^j$. Another property of the exponents is that the algabra of polynomial functions invariant under the action of the Weyl group of g in its standard reflection representation is generated by r homogeneous polynomials of respective degrees $e_1 + 1$, $e_2 + 1$, $\ldots$, $e_r + 1$. Usually the exponents of g are given in weakly increasing order.

Length The length of a Weyl group element w is the smallest number l such that w is a product of l fundamental reflections. Hence, it is the size of a *reduced Weyl word* representing w.

Orbit When a group W acts (from the right) on a set X, any $x \in X$ has an orbit, which is the set of all distinct values of $x \cdot w$ for $w \in W$.

Orbit matrix When a finite group acts on the weight or root lattice, any orbit may be represented by an orbit matrix, each row of which represents one element of the orbit.

Reduced Weyl word When an element w of the Weyl group is expressed as a product $r_{a_1} \cdots r_{a_m}$ of fundamental reflections, and no product of fewer than m fundamental reflections yields w then the sequence $[a_1, \ldots, a_m]$ is a reduced Weyl word for w.

Reflection A Weyl group element that acts on the weight lattice, fixing a sublattice of rank $r - 1$, is an orthogonal reflection in the hyperplane perpendicular to some root. The Weyl group is generated by such reflections.

Weyl group The Weyl group W is defined as the quotient of the normaliser $N_g(T)$ of the maximal torus T in g by the centraliser of T in g (which is T itself). W is a finite group, and has a faithful linear representation on the *weight lattice* $\Lambda(T)$, and the elements of W are often identified with their images in this representation. The *fundamental reflections* $r_1, \ldots, r_s$ in this representation are canonical generators of W.

Weyl word An element of the Weyl group W may be presented as a product of the fundamental reflections r_i $(1 \leq i \leq s)$. If $r_{a_1} \cdots r_{a_m}$ is such a product, the corresponding Weyl group element may be represented by the so-called Weyl word $[a_1, \ldots, a_m]$.

3.4. Representation theory

An important reason for choosing reductive groups as the class of groups to work with in LiE, are the nice properties of representations of such groups. A representation of a Lie group g on a finite dimensional vector space V is a Lie group homomorphism $g \to GL(V)$. Equivalent information is given by specifying a (left) action of g on V such that each map $v \mapsto g \cdot v$ is linear and depends in a differentiable way on g; when taking this point of we we call V a *g-module*. A g-module V is called *irreducible* if it is non-zero, and has no subspaces fixed under the action of g except 0 and V itself. Two fundamental facts about reductive groups are of great importance. First, every g-module decomposes as a direct sum of irreducible representations, i.e., every g-stable subspace has a g-stable complementary subspace. Second, the set of (finite dimensional) irreducible representations is in bijection with the set $\Lambda^+(T)$ of dominant weights, by assigning to each irreducible module its highest weight (which always exists, is unique, and occurs with multiplicity 1). According to the first fact each module M is determined up to isomorphism by the *multiplicity* or *frequency* in M of each irreducible module, (i.e., the number of times it occurs in a direct sum decomposition), while according to the second fact this may be recorded by the set of the highest weights of consituent irreducible modules, with their multiplicities. Representing this set with multiplicities by a polynomial we obtain a *decomposition polynomial.*

It is also possible to represent the set of *all* weights occurring in M, i.e., the character of M, by a polynomial Since W permutes the weigths occurring in the character of M, it suffices for the determination of the character to find just the *dominant* weights occurring in it with their multiplicities; recording these in a polynomial we obtain a *multiplicity polynomial* for the module M.

On the set of g-modules a number of operations can be defined, such as formation of cartesian products and tensor products; also, if a Lie group homomorphism

$f\colon h \to g$ is given then any g-module may be viewed (by restriction) via f as h-module (this is called *branching* from g to h). In terms of characters of the g-modules these operations are easily computed, because each weight corresponds to a 1-dimensional T-module. Cartesian and tensor products correspond to addition respectively multiplication of the polynomials representing the characters. Branching amounts to a linear transformation being applied to all of the exponents in such a polynomial, corresponding to the transition from weights for the maximal torus of g to that of h; the matrix representing the linear transformation is called the *restriction matrix*. Even when the maximal tori of g and h should coincide, the restriction matrix may not be equal to identity, since it should perform the coordinate transformation from the basis of fundamental weights for g to those for h. Despite the simplicity of these operations for characters, it is awkward to have to compute the characters for any modules one would like to perform these operations upon, since the character of a module is usually very much larger than its decomposition polynomial. Therefore some of the most powerful built-in functions of LiE deal with the computation of these operations on the level of decomposition polynomials.

Adjoint representation Each Lie group g acts on its *Lie algebra* (whose underlying space is the tangent space to the group at the identity element) by conjugation, and this defines a *representation* of the group, the so-called adjoint representation. The non-zero weights of this representation are called the *roots* of g and all have multiplicity 1.

Branching Branching is another word for restricting a g-module M to a subgroup h of g. Suppose h is a closed reductive Lie subgroup of the Lie group g. The branching problem concerns finding the decomposition into highest weight modules of M when viewed as an h-module. Since the maximal torus T_g of g is unique up to conjugacy, and similarly for h, the maximal torus T_h of h may be chosen within T_g. Consequently, each weight with respect to T_g determines by restriction a weight with respect to T_h, which defines a linear transformation $\Lambda(T_g) \to \Lambda(T_h)$. The matrix m which describes this transformation on the respective bases of fundamental weights, plays a crucial rôle in the function *branch*. The function *resmat* helps to find the restriction matrix in cases where h is a *fundamental Lie subgroup*. See Chapter 5 for further examples of restriction matrices.

Character For a representation of a group on a finite dimensional vector space we may define a function on the group by assigning to each group element the trace of the corresponding transformation of the vector space. This function, which is constant on conjugacy classes, is called the character of the representation. For reductive complex Lie groups the character determines the representation up to isomorphism, and this is already true for the restriction of the character to the maximal torus T. Now the restriction to T of the representation decomposes into a direct sum of 1-dimensional representations, and the character of such a 1-dimensional representation is just a *weight*. Hence the restriction to T of the character of the whole representation can be correspondingly written as a formal sum of weights (formal because we don't use the Abelian group structure of $\Lambda(T)$ here, but just count the occurring weights with multiplicities, in other words, the sum is taken in the group algebra of $\Lambda(T)$) and this is called the formal character of the representation. In LiE, the formal character of an irreducible representation given by its highest weight can be obtained by calling *branch* with subgroup T_r (i.e., the maximal torus), and restriction matrix $id(r)$.

Decomposition polynomial The decomposition of a g-module M into irreducible modules may be represented by a decomposition polynomial d. Each term $nX\lambda$ of d represents a dominant weight λ such that the highest weight module V_λ occurs in M with multiplicity n. In certain circumstances we allow n to be negative, in which case there is no module corresponding to d, but we may think of M as a formal sum (with integral scalar coefficients) of irreducible modules. In this case M is called a virtual module, and the polynomial a *virtual decomposition polynomial.*

Degree The dimension of the underlying vector space of a representation is called the degree of the representation.

Highest weight The maximum of the set of weights of some irreducible representation of g with respect to the partial ordering '$\prec$' is called the highest weight; it always exists and is a dominant weight that occurs with multiplicity 1. Conversely, every dominant weight occurs as the highest weight of a unique irreducible representation V_λ of g. By definition $\lambda' \prec \lambda$ holds if and only if $\lambda - \lambda'$ is a sum of *positive roots*, and in this case λ is called higher than λ'.

Highest weight module For a dominant weight λ the unique irreducible representation of g with λ as *highest weight*, is called the highest weight module (or representation) of g for λ, and is denoted V_λ

Irreducible representation A representation of a group g is called irreducible if the representation space has no proper non-zero subspace that is stable under g. In case g is a reductive group it suffices that the representation space cannot be decomposed as a *direct sum* of two non-trivial g-stable subspaces.

Module See *representation.*

Multiplicity polynomial Sets of weights with multiplicities may be represented by a multiplicity polynomial m, where each distinct weight v with multiplicity n is represented by a term of m, with coefficient n and exponent v, where $v$is to be interpreted on the basis of fundamental weights. In the case of a *virtual multiplicity polynomial*, multiplicities are allowed to be negative.

Representation An action by linear transformations of a group g on a finite dimensional vector space V is called a (linear) representation of the group; the space V is then called a *module* for g. This is equivalent to giving a (Lie) group morphism $g \to GL(V)$. The irreducible representations of finite groups as well as Lie groups are (up to equivalence) determined by their *characters.* For reductive Lie groups, the irreducible representations are parametrised by their *highest weights.* For the general and special linear groups, the representations can alternatively be indexed by *partitions* (this is where Young tableaux come in): in the case of the *special linear group* $SL(n, \mathbf{C})$, the representation corresponding to the partition $v = [v_1, \ldots, v_d]$ (with $d \leq n$) has highest weight $[v_1 - v_2, v_2 - v_3, \ldots, v_{n-1} - v_n]$, where $v_i = 0$ for $d < i \leq n$. The standard module of $SL(n, \mathbf{C})$, obtained from the injective morphism $SL(n, \mathbf{C}) \to GL(n, \mathbf{C})$, corresponds to the partition [1] and has highest weight $[1, 0, \ldots, 0]$. The partition $[d]$ corresponds to the d-th symmetric power of the standard module, which has highest weight $[d, 0, \ldots, 0]$, and the partition $[1, 1, \ldots, 1]$ of d corresponds to the d-th alternating power of the standard module, which has highest weight $[0, \ldots, 0, 1, 0, \ldots, 0]$, with coefficient 1 in the d-th position.

Restriction matrix If h is a reductive subgroup of g, and a maximal torus of h is chosen within the maximal torus T of g, then any *weight* of g with respect to T

(which is a function on T) becomes by restriction to the maximal torus of h a weight of h. Consequently there is a map from the weight lattice of g to that of h, and this map is linear; it can therefore be given by a matrix, called the restriction matrix for the subgroup h. Each row of this matrix represents the restriction to the maximal torus of h of a *fundamental weight* of g, viewed as a weight of h. The restriction matrix plays a rôle in *branching*.

Virtual decomposition polynomial See *decomposition polynomial.*

Virtual multiplicity polynomial See *multiplicity polynomial.*

3.5. The Symmetric group and related matters

Although it is not a (connected) Lie group, the Symmetric group enters into a number of computations performed by LiE, in particular into *plethysm*. The representation theory of the General Linear group is closely linked with that of the Symmetric group, and either of these theories has a convenient description in terms of partitions and Young tableaux, whereas such a description is not applicable to reductive Lie groups in general. We do not intend to go deeply into these matters here (see [JamKer 1981] for details), suffice it here that partitions of n parametrise the irreducible representations of the Symmetric group S_n on n letters, and that partitions of arbitrary numbers into at most n parts provide an alternative way (besides dominant weights) to parametrise the irreducible representations of GL_n.

To explain the relation of the Symmetric group to representations of arbitrary reductive Lie groups, consider some g-module V and its tensor square $V \otimes V$. The (diagonal) action of g on $V \otimes V$ obviously commutes with the involution of that space that exchanges the two tensorands, and consequently the two eigenspaces of that involution (viz. the spaces of symmetric respectively antisymmetric tensors) are submodules of $V \otimes V$. Therefore we may define operations of forming the symmetric and antisymmetric tensor square of a module, and the ordinary tensor square is the direct sum of these. More generally we may consider arbitrary symmetric and antisymmetric tensor powers of V, consisting of the (fully) symmetric respectively antisymmetric tensors in $V \otimes V \otimes \cdots \otimes V$. For $n > 2$ however, the n-th symmetric and antisymmetrtic tensor powers together do not combine to the full n-th tensor power $V^{\otimes n}$, rather one can decompose $V^{\otimes n}$ into parts corresponding to all of the irreducible representations of S_n (not just the linear ones). The part thus obtained for the S_n-representation R_λ corresponding to a partition λ can be written as a tensor product of some g-module $V^{(\lambda)}$, say, with that S_n-representation R_λ; the module $V^{(\lambda)}$ is then called the plethysm of V with respect to the partition λ.

Character matrix For the symmetric group on n letters, the conjugacy classes are parametrised by *partitions* of n, where the parts of the partition correspond to the disjoint cycles of the permutation. Therefore a *character* χ of the symmetric group may be represented by a character matrix, which is a matrix with $n+1$ columns in which the first n entries of each row represent a partition μ of n (padded with trailing zeros) and the last entry is the value $\chi(\mu)$ of the character χ on the conjugacy class corresponding to μ.

Partition A partition of a natural number n is a weakly decreasing sequence of numbers whose sum is n; adding or removing trailing zeros does not alter the partition. Any partition of n can be represented as a vector $v = [v_1, \ldots, v_n]$ of length n. The LiE function *partitions*(n) produces a matrix whose rows represent the partitions of n. Partitions of n parametrise the conjugacy classes of the symmetric group on n letters and also their irreducible characters; they also parametrise representations of $GL(M)$.

Plethysm A representation of a group g on a vector space M corresponds to a group morphism $g \to GL(M)$; as such it can be composed with any representation of the group $GL(M)$ on a vector space N, giving rise to a representation of g on the space N. Now if we take for the representation of $GL(M)$ the irreducible one parametrised by the partition λ, then the resulting representation of g is called the plethysm, or symmetrised tensor, of M with respect to λ.

Symmetric group The set of permutations of $\{1, \ldots, n\}$ is called the symmetric group on n letters, oftem denoted by S_n. Its conjugacy classes are described by *partitions*, as well as its characters. They play a rôle in *plethysm*.

LiE Manual

Chapter 4. BUILT-IN MATHEMATICAL FUNCTIONS

In this chapter, we list the mathematical functions built into LiE. With each function listed, we give an interpretation of its arguments and the result of its call; furthermore, whenever worthy of mention, a brief indication is given of the algorithm involved in its implementation. For terminology see Chapter 3.

For each function we give a sample heading, in a format similar to what a user defined function would start with, but we allow ourselves to use uppercase and Greek letters, replace any semicolons by commas. A final parameter of type group may be enclosed in an extra pair of parentheses to indicate that it is optional; if corresponding argument is omitted in a call, the default group will be substituted. Then following a colon the result type is given, and whenever appropriate we give enclosed in square brackets additional information about how certain vectors, matrices and polynomials among the parameters and the result should be interpreted.

The possible interpretations for an object of type vector are

- *root*, indicating that it is expressed on the basis of fundamental roots,
- *weight*, indicating expression on the basis of fundamental weights,
- *ints*, denoting the set or sequence of integers forming its entries,
- *Weyl word*, denoting a Weyl group element expressed as a product of fundamental reflections,
- *toral*, denoting either an toral element of finite order or a one parameter subgroup, as decribed in Section 3.3, or
- *partition*, denoting a partition in the usual way.

For objects of type matrix the possible interpretations are

- *lin*(a,b), representing the matrix of a $\mathbf{Z}$-linear transformation, always assumed to act from the right on vectors, where a gives the interpretation (basis) of the vectors acted upon, and b gives the interpretation of the vectors yielded,
- *character*, representing the character of a representation of a symmetric group by its character matrix, or
- *vectors*, *roots*, *weights*, *torals* or *partitions*, representing a set of equal sized vectors without multiplicities—each row giving one vector—with the indicated interpretation of the individual vectors.

Finally, for polynomials the possible interpretations are

- *polynomial*, representing itself as polynomial
- *decomposition*, representing a g-module by the decomposition polynomial for its decomposition into irreducible g-modules,
- *dominant*, representing a set of dominant weights with multiplicities (often the dominant part of the formal character of a representation) by a multiplicity polynomial,

The terms used here are decribed in more detail in Chapter 3. The notation V_λ will be used througout to denote the irreducible g-module with highest weight λ.

4.1. Lie groups

center ((**grp** g)): **mat** [*result: torals*]. Returns a matrix whose rows are semisimple elements or one parameter subgroups generating the center of g. The center of a semisimple Lie group g is a finite Abelian group isomorphic to the quotient of the weight lattice by the root lattice (for reductive groups the central torus is also included). For most simple groups g the center is a cyclic group of order *detcartan*(g) (which order appears in the last column of the result), but for groups of type D_{2n}, the center is a Klein 4-group, so simple components of g of type D_{2n} will account for two rows of the result.

diagram ((**grp** g)): **vid**. Prints the Dynkin diagram of g, also indicating the type of each simple component printed, and labeling the nodes as done by Bourbaki (for the second and further simple components the labels are given an offset so as to make them disjoint from earlier labels).

dim ((**grp** g)): **int**. Returns the dimension of the Lie group g, which is equal to *dim*(*adjoint*$(g), g)$. *Algorithm:* We compute $2 * \mathit{numproots}(g) + \mathit{lierank}(g)$.

liecode (**grp** g): **vec** [*result: ints*]. It is required that g be a simple group or a torus; the function returns a vector $[t, n]$ of size 2, such that $\mathit{liegroup}(t, n) = g$.

liegroup (**int** t, **int** n): **grp**. Returns a torus or a simple group according to the following rule: $\mathit{liegroup}(0, n) = T_n$, $\mathit{liegroup}(1, n) = A_n$, $\mathit{liegroup}(2, n) = B_n$, $\mathit{liegroup}(3, n) = C_n$, $\mathit{liegroup}(4, n) = D_n$, $\mathit{liegroup}(5, n) = E_n$, $\mathit{liegroup}(6, 4) = F_4$, $\mathit{liegroup}(7, 2) = G_2$, and for any other numbers an error is indicated. This function can be useful in order to run examples over many Lie groups using a **for** loop.

lierank ((**grp** g)): **int**. Returns the Lie rank of g; for simple groups and tori this equals $\mathit{liecode}(g)[2]$, while for composite groups it is the sum of the Lie ranks of the component groups.

4.2. Root systems

cartan ((**grp** g)): **mat** [*result: lin(root, weight)*]. Returns the Cartan matrix of g, which is the transformation matrix from the root lattice to the weight lattice, using the bases of fundamental roots and fundamental weights respectively. Hence the i-th row of the Cartan matrix equals the i-th fundamental root, expressed as weigth vector. For simple groups g the labeling of the fundamental roots is Bourbaki's, see [Bourb 1968]. When g is semisimple, the (i, j)-entry of the Cartan matrix is $\langle \alpha_i, \alpha_j \rangle$. When the semisimple rank s of g is differs from the rank r, then the matrix is not square, as it is an $s \times r$ matrix, but all entries beyond column s are zero.

cartan (**vec** α, β, (**grp** g)): **int** [α, β: *root*]. Returns the 'Cartan product' $\langle \alpha, \beta \rangle$, i.e., the integral value $2(\alpha, \beta)/(\beta, \beta)$, where β must be a root, and α is any root vector. [This is is not really an inner product because the function is not linear in β. The function *is* linear in α, and indeed any weight would have been acceptable in place of α, still giving an integral value; nevertheless, to avoid confusion, and because it is most common to take for α a root, we stick to the root basis for α as well as for β]. See also *inprod* and *norm*.

carttype (**mat** R, (**grp** g)): **grp** [R: *roots*]. Returns type of the fundamental Lie subgroup whose root system is the minimal subsystem of the root system of g containing all the roots in R. A basis of fundamental roots of this subsystem may be obtained as *fundam*(R, g). See also *closure* and *centrtype*. *Algorithm:* The same algorithm as *fundam* is performed, but only the type of the root system is returned.

centroots (**vec** t, (**grp** g)): **mat** [t: *toral*, *result*: *roots*]. Returns the matrix whose rows form the set of all positive roots centralising the semisimple element $t \in T$ (or the specified one parameter subgroup). Here a root $\alpha \in \Phi$ is said to *centralise* t if t commutes with all elements of the fundamental Lie subgroup of type A_1 and closed subsystem of roots $\{\alpha, -\alpha\}$. Equivalently, α centralises t if and only if α (which is a weight, and hence a map $T \to \mathbf{C}^*$) vanishes in t. *Algorithm:* Let n be the final entry of t, and t' the vector of remaining entries. First all positive roots are obtained by *posroots*, from which those roots α are selected for which $\alpha_w * t' \equiv 0 \pmod{n}$, where α_w denotes α expressed on the basis of fundamental weights, and $\alpha_w * t'$ is the standard inner product.

centroots (**mat** S, (**grp** g)): **mat** [S: *torals*, *result*: *roots*]. Returns the matrix whose rows form the set of all positive roots centralising the semisimple elements and/or one parameter subgroups represented by the rows of S, which set is the intersection of all sets *centroots*(t, g) with t traversing the rows of S. One may apply *carttype* or *fundam* to the result to obtain the type respectively the set of fundamental roots of the centraliser. See also *centrtype*.

centrtype (**vec** t, (**grp** g)): **grp** [t: *toral*, *result*: *roots*]. Returns the centraliser $C_g(t)$ of the semisimple element $t \in T$ (or of the specified one parameter subgroup); effectively only the type is computed. See also *centroots*. [Actually the centraliser (although connected) need not be simply connected, so the interpretation of the type **grp** of Section 2.2.5 does not admit a precise description of the actual centraliser; the result refers to the unique simply connected group C covering the centraliser subgroup (in other words, there is a finite central subgroup Z of C such that the precise centraliser is isomorphic to the quotient C/Z of C by Z).]

centrtype (**mat** S, (**grp** g)): **grp** [S: *torals*, *result*: *roots*]. Returns the (universal cover of the) centraliser of the semisimple elements and/or one parameter subgroups of T represented by the rows of S, i.e., the intersection of the groups *centrtype*(t, g) for t traversing the rows of S. *Algorithm:* The set *centroots*(S, g) is divided into connected components (where a pair of roots is considered to be joined if they have a non-zero inner product); then in most cases LiE recognises the type from the size of these components. This function can also be computed as *carttype*(*centroots*$(S, g), g)$, which provides a useful check, since in that case the result is obtained by analysing the Dynkin diagram for a base of fundamental roots for the centraliser, rather than by simple counting. (A pre-LiE version of this function, only implemented for types E_n, has been used for [CohGri 1987].)

closure (**mat** R, (**grp** g)): **mat** [R, *result*: *roots*]. Returns the basis of fundamental roots of the minimal closed subsystem of roots of the group g that contains all the roots in R, and moreover consists of positive (for g) roots only. *Algorithm:* First *fundam*(R, g) is computed. Then if g has roots of different lengths, all pairs (α, β) of short roots in the resulting set are tested to see whether $\alpha - \beta$ is a positive root (necessarily a long one), and if so this root replaces α. It can be shown that such changes do not destroy property that the set of roots is fundamental (no positive inner products), so *fundam* need not be applied to the result once more.

detcartan ((**grp** g)): **int**. Returns the determinant of *cartan*(g). This number is the index of the root lattice in the weight lattice, and it is also the order of the center of g. See also *icartan*.

domweights (**vec** λ, (**grp** g)): **mat** [λ: *weight*, *result*: *weights*]. Returns the set of dominant weights which lie under λ, i.e., the set $\{\mu \in \Lambda^+(T) \mid \mu \prec \lambda\}$. This is equal to the set of weights that occur in *domchar*(λ, g). *Algorithm:* Starting with the singleton set $\{\lambda\}$, the closure is formed within the set $\Lambda^+(T)$ under the operation of subtracting positive roots. Note that it would not suffice to subtract just fundamental roots, because certain weights $\mu \in \Lambda^+(T)$ would then only be reachable via weights that are not dominant.

fundam (**mat** R, (**grp** g)): **mat** [R, *result*: *roots*]. Returns the basis of fundamental roots of the minimal subsystem of the root system of g that contains all the roots in R, and moreover consists of positive (for g) roots only. The order in which the the fundamental roots are returned is compatible with the standard labeling for a root system of type *carttype*(R, g). *Algorithm:* As a criterion for a set of positive roots to be a fundamental basis for the minimal subsystem containing them, LiE uses the condition that all mutual inner products be ≤ 0 (note that this implies that the roots are independent). First, all negative roots in R are replaced by their opposites, then each pair of roots that has a positive inner product is replaced by the positive basis of fundamental roots of the rank 2 subsystem they generate, while duplicates are removed by calls of *redsetmat*. This is repeated until no more changes occur.

highroot ((**grp** g)): **vec** [*result*: *root*]. Returns the highest root of the root system of the group g, which must have exactly one simple component (for otherwise there exists no highest root). This root is the last row of *posroots*(g). See also *adjoint*.

icartan ((**grp** g)): **mat** [*result*: *lin*(*weight*, *root*)]. Returns *detcartan*(g) times the inverse of *cartan*(g). The scalar factor *detcartan*(g) is required in order to keep all matrix entries integral. To transform an element of the root lattice that is given as λ in weight coordinates to root coordinates, compute $\lambda *$*icartan*$(g)/$*detcartan*(g).

inprod (**vec** x, y, (**grp** g)): **int** [x, y: *root*]. Returns the Weyl group invariant inner product of x and y. The inner product is normalised such that for each simple component of g the short roots α have *inprod*$(\alpha, \alpha) = 2$.

norm (**vec** α, (**grp** g)): **int**. Returns the norm *inprod*(α, α) of the root vector α. When α is a root, this is one of $\{2, 4, 6\}$, and the inner product is chosen such that for each simple component the short roots have norm 2. Note that this normalisation differs from that used in [Bourb '68] in the case of groups of type B_n, as the short roots are given norm 1 there.

numproots ((**grp** g)): **int**. Returns the number of positive roots of the root system of g, which is equal to *rowsize*(*posroots*(g)). The number of all roots is twice as much, and can also be computed as *dim*(g) − *lierank*(g).

posroots ((**grp** g)): **mat** [*result*: *roots*]. Returns a matrix whose rows are the positive roots of g. The first rows are the fundamental roots (i.e., the top r rows form the matrix *id*(r), and if g is simple the last row, which has index *numproots*(g), is *highroot*(g).

4.3. The Weyl group

dominant (**vec** λ, (**grp** g)): **vec** [λ, *result*: *weight*]. Returns the unique dominant weight in the Weyl group orbit of the weight λ.

dominant (**mat** m, (**grp** g)): **mat** [m, *result*: *weights*]. Returns the set of weights obtained by replacing each row of m by the unique dominant weight in its Weyl group orbit.

exponents ((**grp** g)): **vec** [*result*: *ints*]. Returns the exponents of the given Lie group. For composite groups the exponents are not necessarily increasing, as they are grouped according to the simple factors of the group, with the exponents for the central torus (all zeros) at the end.

length (**vec** w, (**grp** g)): **int** [w: *Weyl word*]. Returns the length of the Weyl group element w. If w is already reduced (e.g., after $w = reduce(w, g)$), then $length(w) = size(w)$. *Algorithm:* The function $reduce(w, g)$ is simulated, recording only length changes.

longword ((**grp** g)): **vec** [*result*: *Weyl word*]. Returns a Weyl word for longest element of the Weyl group. *Algorithm:* We compute $wword(-all\_one[lierank(g)], g)$.

lreduce (**vec** l, w, (**grp** g)): **vec** [l: *ints*, w, *result*: *Weyl word*]. The set l determines a subgroup W_l of W generated by the fundamental reflections r_i for $i \in l$. The function returns a Weyl word for the distinguished representative (element of minimal length) of the left coset $W_l w$. This Weyl word is obtained by deleting certain entries from w; in particular, if w is already a reduced expression for the distinguished representative, then w itself is returned. *Algorithm:* A variant of the algorithm for *reduce* is used, replacing the strictly dominant weight by one that has W_l as stabiliser.

lrreduce (**vec** l, w, r, (**grp** g)): **vec** [l, r: *ints*, w, *result*: *Weyl word*]. The sets l and r determine subgroups W_l and W_r of W generated by the fundamental reflections r_i for $i \in l$ respectively for $i \in r$. The function returns a Weyl word for the distinguished representative (element of minimal length) of the double coset $W_l w W_r$. This Weyl word is obtained by deleting certain entries from w; in particular, if w is already a reduced expression for the distinguished representative, then w itself is returned. *Algorithm:* After computing $lreduce(l, w, g)$ the resulting reflections are applied from right to left to a weight whose stabiliser is W_r, and each reflection that stabilises the intermediate value is thrown away. It can be shown that the result is still left reduced with respect to l.

orbit (**vec** v, **mat** M): **mat** [*result*: *vectors*]. Here v is a vector with an arbitrary interpretation, and M is a matrix whose column size c equals $size(v)$, and whose row size is a multiple of c, say kc. We interpret M as a collection of k square matrices of size $c \times c$, vertically concatenated. The function *orbit* attempts to compute the orbit of v under the group generated by the collection of matrices, i.e., a minimal set V of vectors containing v and closed under right multiplication by any of the matrices in the given collection. As the orbit might be infinite, and the algorithm has no means to detect this situation, it gives up when more than 1000 vectors in the orbit have been computed. For larger orbits, see $orbit(n, v, M)$.

orbit (**int** n, **vec** v, **mat** M): **mat** [*result*: *vectors*]. This function operates in the same way as $orbit(v, m)$, but n replaces the limit of 1000 elements in the orbit.

reduce (**vec** w, (**grp** g)): **vec** [w, *result*: *Weyl word*]. Returns a Weyl word of minimal length representing the same element of W as w. This Weyl word is obtained by deleting certain entries from w; in particular, if w is already a reduced expression, then w itself is returned. See also *lreduce* en *rreduce* and *lrreduce*. *Algorithm:* We apply the reflections in the word w from left to right to a strictly dominant weight, and whenever the intermediate value is found to have a negative coefficient at the position of the reflection being applied (i.e., a negative inner product with the corresponding simple root), then the reflection in question is cancelled against a previous one, which exists by the exchange condition.

reflection (**vec** α, (**grp** g)): **mat** [α: *root*, *result*: *lin*(*weight*, *weight*)]. Returns matrix of the reflection of the weight lattice in the hyperplane perpendicular to the root α, expressed with respect to the basis of fundamental weights. See also *waction*.

rreduce (**vec** w, r, (**grp** g)): **vec** [r: *ints*, w, *result*: *Weyl word*]. The set r determines a subgroup W_r of W generated by the fundamental reflections r_i for $i \in r$. The function returns a Weyl word for the distinguished representative of the right coset wW_r. This Weyl word is obtained by deleting certain entries from w; in particular, if w is already a reduced expression for the distinguished representative, then w itself is returned.

waction (**vec** λ, **vec** w, (**grp** g)): **vec** [λ: *weight*, w: *Weyl word*]. (Weyl action) Returns the weight that is the image $\lambda \cdot w$ of the weight λ under the Weyl group element $w \in W$.

waction (**vec** w, (**grp** g)): **mat** [w: *Weyl word*, *result*: *lin*(*weight*, *weight*)]. Returns the matrix giving the action of the Weyl group element $w \in W$ on the weight lattice, expressed on the basis of fundamental weights. See also *reflection* and *wword*.

worbit (**vec** λ, (**grp** g)): **mat** [λ: *weight*, *result*: *weights*]. (Weyl orbit) Returns the orbit of the weight λ under the Weyl group of g. *Algorithm:* for the classical groups of types A_n, B_n, C_n and D_n, the orbit is generated by permutations and (for types other than A_n) sign changes, after a suitable linear transformation, using a procedure similar to *nextpermu*. For the exceptional groups (of type E_n, F_4, and G_2), a large subgroup of the Weyl group W is chosen that is of classical type, for which the same method is employed; it remains to traverse the small number of cosets of this subgroup in W. This algorithm is much faster than the general function *orbit*.

worbitsize (**vec** λ, (**grp** g)): **mat**. (Weyl orbit size) Returns the length of the orbit of the weight λ under the Weyl group of g. This is equal to *worder*(g)/*worder*(I, g), where I is a vector whose entries indicate the positions at which the vector λ has zero entries.

worder ((**grp** g)): **int**. (Weyl group order) Returns the order of the Weyl group of g.

worder (**vec** I, (**grp** g)): **int** [I: *ints*]. Returns the order of the subgroup W_i of the Weyl group of g generated by the fundamental reflections r_i for $i \in I$. This subgroup is the stabiliser subgroup of any weight vector that has zero entries precisely at positions i for which $i \in I$. *Algorithm:* We compute *worder*(*carttype*$(R, g), g)$, where R is the set of roots obtained by taking for each element $i \in I$ the i-th fundamental root.

wrtaction (**vec** α, w, (**grp** g)): **vec** [α: *root*, w: *Weyl word*]. (Weyl root action) Returns the root that is the image $\alpha \cdot w$ of the root vector α under the Weyl group element $w \in W$.

wrtaction (**vec** w, (**grp** g)): **mat** [w: *Weyl word*, *result*: *lin*(*root*, *root*)]. Returns the matrix giving the action of the Weyl group element $w \in W$ on the root lattice, expressed on the basis of fundamental roots.

wrtorbit (**vec** α, (**grp** g)): **mat** [α: *root*, *result*: *roots*]. (Weyl root orbit) Returns the orbit of the root vector α under the Weyl group of g.

wword (**mat** m, (**grp** g)): **vec** [m: *lin*(*weight*, *weight*), *result*: *Weyl word*]. Returns a Weyl word for the Weyl group element w—if it exists—whose its action on the weight lattice is given by the square matrix m. This function is the inverse of *waction* applied to Weyl words, except that it may return another representative for the same element; in fact (after *setdefault*(g)), for each Weyl word w the call *wword*(*waction*(w)) returns a canonical representative for the equivalence class of w. See also *waction*.

wword (**vec** λ, (**grp** g)): **vec**. Returns a Weyl word for a Weyl group element w sending the weight λ to a dominant weight. In fact, w is the distinguished representative of the coset wW_S, where W_S is the stabiliser of $\lambda' = dominant(\lambda)$ (here S is the set of indices of fundamental reflections which stabilise λ', i.e., the set of indices i for which $\lambda'[i] = 0$).

4.4. Representations

adams (**int** n, **vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *dominant*]. Returns the virtual multiplicity matrix of the virtual module of the simple group g, whose character is obtained from that of V_λ by multiplying all the occuring weights by n, while retaining the multiplicities. The adams operator is the 'weight analog' of the operator that, given a character χ of a group g and a number n, computes the decomposition of the class function $\gamma \mapsto \chi(\gamma^n)$ as an integral linear combination of irreducible characters. The adams operator is used in *plethysm*, *symtensor*, and *alttensor*. *Algorithm:* Effectively, *vdecomp*(*domchar*(λ, g) $*$ *id*(n), g) is computed.

adams (**int** n, **pol** d, (**grp** g)): **pol** [d: *decomposition*, *result*: *dominant*]. This is like *adams*(n, λ, g), but with the irreducible module V_λ replaced by the (reducible) module represented by the decomposition polynomial d.

adjoint ((**grp** g)): **vec** [*result*: *weight*]. Returns the highest weight of the adjoint representation of the group g. The group has to be simple, for otherwise the adjoint representation is not irreducible; en example of how the decomposition matrix of the adjoint representation can be computed for non-simple groups is given in Section 5.2.3. Since the non-zero weights of the adjoint representation are precisely the roots, one has *adjoint*(g) = *highroot*(g) $*$ *cartan*(g).

alttensor (**int** n, **vec** λ, (**grp** g)): **mat** [λ: *weight*, *result*: *decomposition*]. (alternating tensor) Returns the decomposition matrix of $\bigwedge^n V_\lambda$, the n-th exterior power of V_λ. The group g has to be simple. See also *symtensor* and *plethysm*.

branch (**vec** λ, **grp** h, **mat** m, (**grp** g)): **mat** [λ: *weight*, m: *lin*(*weight*, *weight*), *result*: *decomposition*]. Returns the decomposition matrix of the restriction to h of V_λ, with respect to the restriction matrix m. Here the matrix m is such that any weight λ', (expressed on the basis of fundamental weights for g) when restricted to the maximal torus of h becomes a weight $\lambda' * m$ (expressed on the basis of fundamental weights for h); the group g has to be simple. For fundamental Lie subgroups (among which the Levi subgroups) this matrix can be obtained by use of *resmat*. Branching to T_r with $m = id(r)$, where $r = lierank(g)$ amounts to computing the character of V_λ. *Algorithm:* The whole character of V_λ is traversed by generating for each weight occuring in *mul*(λ, g) its Weyl group orbit. To every weight thus generated the matrix m is applied; if the result is a dominant weight of h, it is appended as a row to a matrix a. Finally $decomp(a, h)$ is computed. Within each Weyl group orbit LiE generates the weights one at a time, using a dynamic version of *worbit* to prevent storage problems.

collect (**mat** d, **grp** h, **mat** m, (**grp** g)): **mat** [d, *result: decomposition*, m: *lin*(*weight*, *weight*)]. The matrix m should be invertible (and in particular square); let $r = m^{-1}$. Then *collect* returns the decomposition matrix of the g-module whose restriction to the reductive subgroup h with respect to the restriction matrix r has decomposition matrix d (provided that such a matrix exists). In other words, it is an inverse of *branch* in the same sense that *decomp* is an inverse of *mul*: the call $collect(branch(\lambda, h, r, g), h, m, g)$ should return the decomposition matrix $[\lambda + 1]$. *Algorithm:* Essentialy, this function is identical to *branch*, except that no Weyl group orbits are generated, since it is assumed that for any weight $\lambda \in \Lambda(T_h)$ the corresponding weight $\lambda * m \in \Lambda(T_g)$ can *only* be dominant if λ was already dominant; this assumption is valid if m is the inverse of a resrtiction matrix to a subgroup. The fact that *collect* performs the inverse action of *branch* is mainly accounted for by the fact that the inverse of the restriction matrix is to be supplied.

contragr (**vec** λ, (**grp** g)): **vec** [λ, *result: weight*]. Yields the highest weight of the contragredient (or dual) representation V_λ^* of V_λ, which equals $dominant(-\lambda, g)$. The group g has to be simple.

decomp (**mat** m, (**grp** g)): **mat** [m: *dominant*, *result: decomposition*]. Returns the decomposition matrix of the g-module with multiplicity matrix m, in other words, it is essentially an inverse of *mul*: the call $decomp(mul(\lambda, g), g)$ should return the decomposition matrix $[\lambda + 1]$, indicating that a single irreducible constituent λ was found with multiplicity 1. See also *vdecomp*.

dim (**vec** λ, (**grp** g)): **int** [λ: *weight*]. Returns the dimension of the representation V_λ.

domchar (**vec** λ, (**grp** g)): **mat** [λ: *weight*, *result: dominant*]. Returns the polynomial representing the dominant part of the character of the g-module V_λ. *Algorithm:* Freudenthal's multiplicity formula, see [Hum 1972] and [Kruse 1971].

domchar (**vec** λ, μ, (**grp** g)): **int** [λ, μ: *weight*]. Returns the multiplicity of μ in the character of V_λ. The weight λ should be dominant, but μ may be any weight.

ptensor (**int** n, **vec** λ, (**grp** g)): **mat** [λ: *weight*, *result: decomposition*]. Returns the decomposition matrix of the n-th tensor power $\bigotimes^n V_\lambda$ of V_λ. The group g has to be simple.

ptensor (**int** n, **mat** d, (**grp** g)): **mat** [d, *result: decomposition*]. Returns the decomposition matrix of the n-th tensor power of the g-module with decomposition matrix d. The group g has to be simple.

resmat (**mat** R, (**grp** g)): **mat** [R: *roots*, *result*: *lin*(*weight*, *weight*)]. It is assumed that the set R consists of roots forming a fundamental basis for a closed subsystem Φ' of the root system Φ of g (as for instance obtained by a call of *closure*). The function returns the restriction matrix for the semisimple Lie subgroup of g with root system Φ'.

spectrum (**vec** λ, t, (**grp** g)): **vec** [λ: *weight*, t: *toral*, *result: ints*]. Let n be the last entry of t, then the semisimple element $t \in T$ will act in any representation of g as a diagonalisable transformation whose as eigenvalues are all n-th roots of unity. The function *spectrum* returns the vector of length n, whose $i+1$-st entry $(0 \le i < n)$

is the multiplicity of the eigenvalue ζ^i in the action of the semisimple element on the irreducible g-module V_λ, where ζ is the complex number $e^{2\pi i/n}$. The group g has to be simple. The result can also be obtained by calling *branch* to compute the restriction to the one parameter subgroup containing the semisimple element, see Section 5.5.3. *Algorithm:* The character is computed using *mul* and *worbit*; for each occurring weight the contribution to the result is easily computed. [A pre-LiE version of this function, only implemented for E_n, has been used for [CohGri 1987].]

symtensor (**int** n, **vec** λ, (**grp** g)): **mat** [λ: *weight*, *result*: *decomposition*]. (symmetric tensor) Returns the decomposition matrix of $S^n(V_\lambda)$, the n-th symmetric tensor of V_λ. The group g has to be simple. See also *alttensor* and *plethysm*. *Algorithm:* We use the recursion

$$n \cdot symtensor(n,\lambda) = \sum_{k=1}^{n} symtensor(n-k,\lambda) \otimes adams(k,\lambda).$$

This formula turns into a recursion formula for *alttensor* upon including a sign $(-1)^{k-1}$ in the summand.

tensor (**vec** λ, μ, (**grp** g)): **mat** [λ, μ: *weight*, *result*: *decomposition*]. Returns the decomposition matrix of the tensor product $V_\lambda \otimes V_\mu$ The group g has to be simple. *Algorithm:* Klimyk's formula has been implemented, see [Hum 1972, Exerc. 24.9]. Like in *branch*, a dynamic version of *worbit* is used to prevent storage of a complete Weyl group orbit.

tensor (**vec** λ, μ, ν, (**grp** g)): **int** [λ, μ, ν: *weight*]. Returns the multiplicity of the weight ν in the tensor decomposition of $V_\lambda \otimes V_\mu$. The group g has to be simple.

tensor (**mat** d, d', (**grp** g)): **mat**. Returns the decomposition matrix of the tensor product of the g-modules with respective decomposition matrices d and d'. The group g has to be simple.

vdecomp (**mat** m, (**grp** g)): **mat** [m: *dominant*, *result*: *decomposition*]. (virtual decomposition) Returns the virtual decomposition matrix of the virtual g-module with multiplicity matrix m. The algorithm is the same as for *decomp*, but no restriction is put on the sign of the multiplicities. This function is used in *adams*.

4.5. Operations related to the Symmetric group

nextpart (**vec** λ): **vec** [λ, *result*: *partition*]. Returns the next partition of $|\lambda|$ in reverse lexicographic order. If λ is the last one, i.e., if $\lambda = [1,1,\ldots,1]$, it will return λ again. See also *partitions*.

nextpermu (**vec** p): **vec** [p, *result*: *ints*]. Returns the next permutation of the entries of p, in reverse lexicographical order reading from right to left. If p is the last such permutation, i.e., if the entries of p are increasing, then p itself will be returned again. If there are repetitions among the entries of p, then this function will not attempt to permute identical entries, and in such cases it will take fewer applications of *nexpermu* to go from the weakly decreasing order to the weakly increasing order. See also *symorbit*.

partitions (**int** n): **mat** [*result*: *partitions*]. Returns a matrix whose rows are the partitions of n in reverse lexicographic order, and extended by zeros to length n. See also *nextpart*.

plethysm (**vec** λ, μ, (**grp** g)): **mat** [λ: *partition*, μ: *weight*, *result*: *decomposition*]. Returns the decomposition matrix of the g-module obtained from V_μ by taking the symmetrised tensor with respect to the partition λ. For example $plethysm([n], \mu, g)$ equals $symtensor(n, \mu, g)$ and $plethysm([1, 1, \ldots, 1], \mu, g)$ where the partition has n parts is equal to $alttensor(n, \mu, g)$ [it makes sense to check these facts since the algorithms differ]. The group g has to be simple. *Algorithm:* We use the classical Frobenius Formula (cf. [And 1967] and [JamKer 1981])

$$plethysm(\lambda, \mu) = \frac{1}{n!} \bigoplus_{\kappa \in \mathcal{P}_n} conjord(\kappa) \chi^\lambda(\kappa) \bigotimes_{i=1}^{l(\kappa)} adams(\kappa_i, \mu),$$

where $n = |\lambda|$, κ runs over all partitions of n, the number $conjord(\kappa)$ counts then order of the conjugacy class in the symmetric group on n letters of permutations with cycle type κ, χ^λ is the irreducible character of that symmetric group corresponding to the partition λ, and $l(\lambda)$ denotes the number of non-zero parts κ_i of κ. Hence the algorithm uses *addmul*, *partitions*, *symchar*, *adams*, and *tensor*.

symchar (**vec** λ): **mat** [λ: *partition*, *result*: *character*]. (symmetric group character) Returns the character matrix of the character χ^λ of the symmetric group on $|\lambda|$ letters, corresponding to the partition λ. *Algorithm:* For each partition μ in $partitions(|\lambda|)$ the function $symchar(\lambda, \mu)$ is called.

symchar (**vec** λ, **vec** μ): **int** [λ, μ: *partition*]. (symmetric group character) We should have $|\lambda| = |\mu|$; the function returns the (integral) value $\chi^\lambda(\mu)$ of the character of the symmetric group on $|\lambda|$ letters corresponding to λ on the conjugacy class with cycle type μ. *Algorithm:* We use the formula that expresses the character as an alternating sum of characters of permutation representations on sets of "flags", see [JamKer 1981].

symorbit (**vec** v): **mat** [*result*: *vectors*]. (symmetric group orbit) The symmetric group on n letters acts on $\mathbf{Z}^n$ by permuting the coordinates; the function returns the orbit of v in this action, where $n = size(v)$. The rows of the result are ordered reverse lexicographically, reading from right to left. See also *nextpermu*.

LiE Manual

Chapter 5. EXAMPLES

In this chapter we illustrate how LiE can be used to study Lie groups and their representations, and how one can use the built-in functions and the capabilities of the interpreter to tailor solutions to specific problems.

5.1. General

5.1.1. Reversing the ordering

The standard function *sort* sorts the entries of a vector v into decreasing order. To sort a vector into increasing order, call '$-sort(-v)$'. The same trick works for matrices instead of vectors.

5.1.2. Union of sets of vectors

Suppose a and b are matrices representing sets of vectors. Then a matrix representing the union of these set can be obtained by the call '*redsetmat*(a^b)'.

5.1.3. Sum and product of vector entries

The following commands define functions that compute the sum and product of the entries of a vector.

```
sum(vec v) =  loc ans = 0;  for i in v do ans = ans + i od; ans
prod(vec v) = loc ans = 0;  for i in v do ans = ans * i od; ans
```

Incidentally, there is a slicker solution in the first case, namely to form the inner product with the all-one vector, so one could alternatively define

```
one(int i) = 1
ones(int i) = make(one, i)
sum(vec v) = v * ones(size(v))
```

The latter solution is more efficient than the former one, and even then, most time is spent computing *ones*(*size*(v)); this is so because built-in operations (such as the standard inner product) are executed much more efficiently than programs executed by the interpreter.

5.1.4. Comparing groups

The operator == is not defined for groups. The function *equal* defined below will test equality of groups, where groups that differ in the order of their simple factors—although isomorphic—are considered to be distinct.

```
equal(grp g, h) = \
   if compsize(g) != compsize(h) then 0 else \
      for i = 0 to compsize(g) do \
         if liecode(g[i]) != liecode(h[i]) then break(0) else 1 fi \
      od \
   fi
```

5.2. Roots

Here are a few simple examples of how to obtain information about root systems.

5.2.1. All roots

The function *roots* that computes the full root system of g can be defined as follows:

```
roots(grp g) = loc m = posroots(g); m^-m
```

5.2.2. The half sum of the positive roots

In many cases one needs the weight

$$\rho = \frac{1}{2} \sum_{\alpha \in \Phi^+} \alpha,$$

the half sum of the positive roots. It can be computed directly by

```
rho(grp g) = loc sum = null(lierank(g)); \
   for alpha row posroots(g) do sum = sum + alpha od; sum/2
```

or, using the same trick as in Section 5.1.3, and the same function *ones*,

```
rho(grp g) = posroots(g) * ones(numproots(g))/2
```

Using the fact that ρ, when expressed on the basis of fundamental weights has all coordinates equal to 1, there is an even quicker solution to this question, namely to use the coordinate transformation *icartan*/*detcartan*:

```
rho(grp g) = ones(lierank(g)) * icartan(g)/detcartan(g)
```

The only problem with this solution is that it fails for non-simple groups, since these are refused by *icartan* and *detcartan*. This could be circumvented by a loop similar to the one in Section 5.1.4.

5.2.3. Adjoint representation of a non-simple group

The function *adjoint* has only been defined for simple groups g. For general groups g, the following function computes a the decomposition matrix of the adjoint representation.

```
gadjoint(grp g) = loc d = null(0,0); \
   for i = 1 to compsize(g) do d = blockmat(d,[adjoint(g[i])]) od; \
   if lierank(g[0]) == 0  # no central torus # \
   then *(*d + ones(rowsize(d)))  # add multiplicities 1 # \
   else loc tr = lierank(g[0]); \
      *(*blockmat(d, null(1,tr)) + (ones(rowsize(d)) + tr))\
   fi
```

5.3. Weyl words

5.3.1. From a Weyl word to a Weyl group element

The function *wword* transforms a matrix on the weight basis into a corresponding Weyl word. For the inverse function, the function *waction* is useful:

```
# welt is short for Weyl element #
welt(vec w) = loc r = size(w);  loc m = id(r); \
    for i = 1 to r do m[i] = waction(m[i], w) od;  m
```

It is also possible to use *weylmat*, which actually performs the requested operation, but returns a matrix on the root basis. So it is necessary to conjugate by the cartan matrix:

```
welt(vec w) = cartan * weylmat(w) * icartan/detcartan
```

which assumes, as does the first solution, that the default group has been set appropriately.

5.3.2. The Coxeter matrix

The Coxeter matrix of a Weyl group is the matrix with entries $m_{i,j}$ equal to the order of the product $r_i r_j$ of the fundamental reflections r_i and r_j. Here is a (rather inefficient) way to compute it.

```
coxmat() =  m = id(lierank()); \
    for i = 1 to rowsize(m) - 1 do for j = i + 1 to rowsize(m) \
    do m[i,j] = ord(fund_refl(i) * fund_refl(j));  m[j,i] = m[i,j] \
    od od;  m
fund_refl(int n) = reflection(id(lierank)[n])
ord(mat m) =  loc p = m;  loc idmat = id(rowsize(m)); \
    for i = 1 to 6 do if p == idmat then break(i) else p = p * m fi od
```

Note how the function *fund_refl* obtains standard basis vectors as rows of the identity matrix. In the same vein it is possible implement the function *ones* by taking the diagonal of the identity matrix. Of course this is a rather wasteful approach when the vectors become really big, but if their size does not exceed 100, say, then this solution is probably as efficient as any, since the built-in operation of matrix creation is really quite fast.

5.3.3. All reduced Weyl words of a given element

Tits has shown that, to produce all reduced Weyl words corresponding to the same Weyl element, all that is needed is to start with one such word, and to continue substituting occurrences of the subword $[i,j,i,\ldots]$ of length m, where m is the order of the product $r_i r_j$ of the corresponding fundamental reflections, by $[j,i,j,\ldots]$ of the same length. The following routine *nextrewrite* could form a basic ingredient in the enumeration of all equivalent Weyl words: it produces the indicated replacement (if possible) in the Weyl word v for the subword that begins at the k-th entry of v.

```
# try rewriting reduce(v) at position k #
setdefault(g)
nextrewrite(vec v;  int k) =  loc v = reduce(v); \
    loc m = coxmat[v[k], v[k + 1]];  loc check = 1; \
    for j = 1 to (m - 1)/2 do \
        if 2 * j + k > size(v) || v[2 * j + k] != v[k] then check = 0;  break fi \
    od; \
    if check then for j = 1 to m/2 - 1 do \
        if 2 * j + k + 1 > size(v) || v[2 * j + k + 1] != v[k + 1] \
            then check = 0;  break fi \
    od fi; \
    if check then vswap(v, k, m) else v fi
```

```
vswap(vec v; int k, m) = loc t = v[k + m - 2]; \
   for j = k to k + m - 1 do v[j] = v[j + 1] od; v[k + m - 1] = t; v
```

The function *coxmat* is as in the previous subsection.

5.3.4. The Bruhat ordering

The following function *bruhat* returns a Weyl word for each Weyl group element that is covered by a given element v in the so-called Bruhat order.

```
bruhat(vec v) = loc v = reduce(v); loc m = null(0, size(v) - 1); \
   for i = 1 to size(v) do \
      loc w = reduce(v - i); if size(w) == size(v) - 1 then m = m + w fi \
   od; \
   m = redsetmat(m); \
   # it remains to check whether two rows represent wwords # \
   # corresponding to the same Weyl group element # \
   rho = ones(lierank); \
   for i = 1 to rowsize(m) - 1 do for j = i + 1 to rowsize(m) do \
      if waction(rho, m[i]) == waction(rho, m[j]) then m[j] = m[i] fi \
   od od; \
   redsetmat(m)
```

5.4. Cosets in The Weyl group

There are many ways to compute cosets in W with respect to Weyl subgroups generated by a subset of the set of fundamental reflections. Here, we show how to recover some of the results in [BrouCoh 1985].

5.4.1. Right cosets

Suppose S is a subset of $\{1, \ldots, r\}$ and W is a Weyl group of rank r. Then there is a natural system of representatives of the cosets of the Weyl subgroup W_S of W generated by the fundamental reflections r_i for $i \in S$. This is the set of all distinguished right coset representatives, i.e.,, all elements w of W that satisfy $rreduce(w, S) = w$. Here is how to generate this set, using the fact that W_S is the stabiliser of any weight vector that has zeros precisely at those positions whose index occurs in S.

```
charv(vec s) = loc y = ones(lierank); \
   for i = 1 to size(s) do y[s[i]] = 0 od; y
rcosets(vec r) = for wt row worbit(charv(r)) do print(wword(wt)) od
```

Again, before invoking the function *rcosets*, a default group has to be set; for example, after the above definitions of *charv* and *lcosets* have been read, the left coset representatives for the subsystem $A_1A_1A_1$ in D_4 can be found as follows:

```
setdefault(D4); w = [1,3,4]; rcosets(w)
```

Note that $[1, 3, 4]$ represents the nodes corresponding to the subsystem $A_1A_1A_1$, which can be verified by calling $diagram(D_4)$. Another—more elaborate—way to verify this is to ask for the Cartan type of the subsystem generated by the fundamental roots with indices 1, 3, and 4, by calling $carttype(id(4) - 2, D_4)$.

5.4.2. Left cosets

Using the fact that a Weyl word v is left reduced with respect to the subset S of $\{1, \ldots, s\}$ if and only if its inverse $[v[l], v[l-1], \ldots, v[1]]$, where $l = size(v)$, is right reduced with respect to S, we can write the following variation to the previous example to obtain a print of the list of left coset representatives:

```
$inverse(\textbf{vec } v) =$   $\textbf{loc } vinv = v;$ $\textbf{loc } s = size(v) + 1;$ \
    $\textbf{for } i = 1 \textbf{ to } s - 1 \textbf{ do } vinv[s - i] = v[i] \textbf{ od};$ $vinv$
$lcosets(\textbf{vec } l) =$ \
    $\textbf{for } x \textbf{ row } worbit(charv(l)) \textbf{ do } print(inverse(wword(x))) \textbf{ od}$
```

where *charv* is as before.

5.4.3. Double cosets

We now construct a function *dcosets* printing the full set of distinguished double coset representatives, displayed as left and right reduced Weyl words, with respect to specified subsets L and R of $\{1, \ldots, r\}$. It suffices to modify *rcosets* such that it only prints those Weyl words (already right reduced for R) that are left reduced for L.

```
$dcosets(\textbf{vec } l, r) =$ \
    $\textbf{for } x \textbf{ row } worbit(charv(r)) \textbf{ do } \textbf{loc } w = wword(x);$ \
        $\textbf{if } w == lreduce(l, w) \textbf{ then } print(w) \textbf{ fi}$ \
    od
```

Of course it is also possible to put the coset representatives in a matrix. For this purpose, the Weyl words need to have the same length, which can be achieved by padding with zeros, as already illustrated in the function *bruhat* above. A good upper bound for the number of columns needed is $lrreduce(v, longword, w)$.

5.5. Semisimple elements

In LiE, a semisimple element is represented by a vector $[a_1, \ldots, a_r, d]$. This vector corresponds to the element t of the maximal torus T with $t^{\omega_i} = e^{2\pi \mathrm{i} a_i/d}$ for $1 \le i \le r$.

5.5.1. $SL(n, \mathbf{C})$

For the special linear group $SL(n, \mathbf{C})$ there is a much more familiar way to describe a semisimple element, namely by its diagonal entries in diagonalised form. If t is a diagonal matrix with entries $(t_1, \ldots, t_n)$ on the main diagonal in the standard representation, then the values of the fundamental weights ω_i on t are given by

$$t^{\omega_i} = \prod_{j=1}^{i} t_j.$$

Therefore, for g of type A_{n-1}, let t be a semisimple element whose diagonalised form has entries $[\zeta^{b_1}, \ldots, \zeta^{b_n}]$ along the main diagonal, where $\zeta = e^{2\pi \mathrm{i}/d}$ is an d-th root of unity (note that $\sum_{j=1}^{n} b_j \equiv 0 \pmod{d}$ since $t \in SL(n, \mathbf{C})$). Then t can be represented in LiE by applying the following function *mksselt* (an abbreviation for make semisimple element), to the vector $[b_1, \ldots, b_n]$ and the number d:

```
$mksselt(\textbf{vec } b; \textbf{ int } d) =$   $\textbf{loc } n = size(b);$ \
    $\textbf{for } i = 2 \textbf{ to } n - 1 \textbf{ do } b[i] = (b[i-1] + b[i]) \,\%\, d \textbf{ od};$ $b[n] = d;$ $b$
```

Note that we use the parameter b itself (in fact a copy of the actual argument) to build up the result in; all entries may be reduced modulo d, and the redundant final entry is used to record the denominator d.

5.5.2. $SO(12, \mathbf{C})$

Here is yet another example, now with the group of type D_6. Consider the standard 12-dimensional representation where it acts as the orthogonal group $SO(12, \mathbf{C})$ (note: since "the group of type D_6" should be read as the simply connected group of that type, which is $Spin(12, \mathbf{C})$, this is not a faithful representation: the kernel consists of a central subgroup of order 2). We fix a basis $e_1, \ldots, e_6, f_1, \ldots, f_6$ of the underlying 12-dimensional complex vector space with respect to which the bilinear form $\langle \cdot, \cdot \rangle$ fixed by $SO(12, \mathbf{C})$ satisfies $\langle e_i, e_j \rangle = \langle f_i, f_j \rangle = 0$ and $\langle e_i, f_j \rangle = \delta_{i,j}$ for all $i, j \in \{1, \ldots, 6\}$. Suppose now that $t \in SO(12, \mathbf{C})$ is given by the diagonal matrix with diagonal entries $[\zeta^{a_1}, \ldots, \zeta^{a_6}, \zeta^{-a_1}, \ldots, \zeta^{-a_6}]$, where again $\zeta = e^{2\pi i/d}$. Then the following function, using the given matrix m and the integer d that is assumed to have an appropriate value, transforms the vector $[a_1, a_2, \ldots, a_6]$ into the form used by LiE to represent t.

$$m = [[2,2,2,2,1,1],[0,2,2,2,1,1],[0,0,2,2,1,1],\backslash$$
$$[0,0,0,2,1,1],[0,0,0,0,1,1],[0,0,0,0,-1,1]]$$
$$mkss(\mathbf{vec}\ a) = a * m + d$$

5.5.3. Spectrum

The function *spectrum* provides a means to recognize the semisimple element specified in a more natural form. For instance, we perform the following computation for a semisimple element t of order 2 in $SL(5, \mathbf{C})$:

setdefault(A_4); $t = [1,0,0,0,2]$; $sr = [1,0,0,0]$ # standard representation #
spectrum(sr, t)

which returns $[3, 2]$, showing that t (an element of order 2) has 3 eigenvalues 1, and 2 eigenvalues -1 in the standard representation. It is therefore conjugate to the element *mksselt*($[0,0,0,1,1]$, 2), with *mksselt* as above, which equals $[0,0,0,1,2]$; the element t itself can be obtained as *mksselt*($[1,1,0,0,0]$, 2). To obtain information about the whole 1-dimensional torus containing t (which may be represented by replacing the final entry of t by 0), one can use the function *branch*. The restriction matrix needed for such a 1-dimensional torus is essentialy obtained by transposition of the vector, in the current case $*[t-5]$. Computing *branch*(sr, T_1, $*[t-5]$) we find the matrix

$$\begin{pmatrix} 0 & 3 \\ -1 & 1 \\ 1 & 1 \end{pmatrix},$$

which shows that an arbitrary element of that 1-dimensional torus parametrised by some $z \in \mathbf{C}^*$ has 3 eigenvalues 1, 1 eigenvalue z and 1 eigenvalue z^{-1}; this is in accordance with the fact that such an element has matrix

$$\begin{pmatrix} z & 0 & 0 & 0 & 0 \\ 0 & z^{-1} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(the element t corresponds to $z = -1$). The centraliser of t can be found by the call *centrtype*(t), which returns $A_2A_1T_1$, and the centraliser *centrtype*($t - 5 + 0$) of its containing 1-dimensional torus is A_2T_2. On the other hand, *spectrum*(*adjoint*, t)[1] returns the dimension of the Lie subalgebra fixed by t (where t is viewed as an automorphism of the Lie algebra of g), which is the Lie algebra of the centraliser of t; the call returns 12, which is indeed equal to *dim*(*centrtype*(t)). To obtain the corresponding dimension of the centraliser of the 1-dimensional torus we can again use *branch*: the call *branch*(*adjoint*, $T_1, *[t-5]$) returns

$$\begin{pmatrix} -2 & 1 \\ 2 & 1 \\ 0 & 10 \\ -1 & 6 \\ 1 & 6 \end{pmatrix},$$

which shows that that subalgebra fixed by this torus has dimension 10, in accordance with $dim(A_2T_2)$. In general we see that the function *spectrum* may be simulated by using *branch*, as follows:

```
$spec(\mathbf{vec}\ wt, t) =$ **loc** $s = size(t);$ **loc** $d = t[s];$ **loc** $res = null(d);$ \
    **for** $x$ **row** $branch(wt, T_1, *[t-s])$ \
    **do loc** $p = x[1] \% d + 1;$ $res[p] = res[p] + x[2]$ **od**; \
    $res$
```

5.5.4. Branching to a centraliser

We continue with the semisimple element of the preceding paragraph; we wish to compute how the standard representation decomposes when restricted to the centraliser of the semisimple element t, which we have already seen to be of type $A_2A_1T_1$. We start with computing the centraliser more explicitly by calling *centroots*(t); this returns

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

This is the full set of positive roots centralising t; we would like to have a basis of fundamental roots and the corresponding type, to which end we compute $f =$ *fundam*($) and *carttype*($f$), which give respectively

$$f = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \text{and} \qquad A_2A_1,$$

where we note that the central torus part of the centraliser is no longer represented, since there are no corresponding roots. In order to branch to the centraliser we need the restriction matrix $m =$ *resmat*(f) which gives

$$m = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Finally we can compute *branch*(sr, A_2A_1, m), which returns

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

indicating that the standard representation of $SL(5, \mathbf{C})$ decomposes into the direct product of the standard representations of $SL(3, \mathbf{C})$ and $SL(2, \mathbf{C})$, when restricted to the centraliser of t.

5.6. Checks

Numerous checks are possible to verify the consistency between results produced by different functions. We have already mentioned many of them above and in Chapter 4, which we will not repeat here, but we note a number of additional checks that can be made.

5.6.1. Multiplicities

Since *mul*(λ) computes the dominant part of the character of V_λ, it is possible to check the dimension of V_λ; we should have

$$\sum_{\mu\in\Lambda^+(T),\mu\prec\lambda} mul(\lambda,\mu) * worbitsize(\mu) = deg(\lambda).$$

The best way to chack this in LiE is by the following function:

$checkdim(\mathbf{vec}\ wt) = \ \mathbf{loc}\ c = size(wt)+1;\ \mathbf{loc}\ d = 0;\ \backslash$
$\quad\mathbf{for}\ r\ \mathbf{row}\ mul(wt)\ \mathbf{do}\ d = d + worbitsize(r-c) * r[c]\ \mathbf{od};\ d == deg(wt)$

The function *mul* itself is also very useful in performing tests, since many operations such as *tensor* and *plethysm* have an easily expressed effect on the characters, because the representation theory of tori is much simpler than that of semisimple Lie groups.

5.6.2. Branching

We may similarly check for branching operations that the dimension of the resulting sum of h-modules equals that of the original g-module V_λ. We should have

$$\sum_{\mu\in\Lambda^+(T_h)} mul(branch(\lambda,h,m),\mu) * deg(\mu,h) = deg(\lambda),$$

where h is a reductive subgroup of g with restriction matrix m (when m is not really a restriction matrix from g to h, the test may easily fail). Here we have used a function $mul(m,\lambda)$ which extracts the multiplicity of a weight λ from a multiplicity matrix m; such a function is not built-in, but the user might add such a function written in the LiE language. In the present case however, that function is not really needed; we may write a function *chkbranch* as follows:

$chkbranch(\mathbf{vec}\ wt;\ \mathbf{grp}\ h;\ \mathbf{mat}\ m) = \ \mathbf{loc}\ c = lierank(h)+1;\ \mathbf{loc}\ d = 0;\ \backslash$
$\quad\mathbf{for}\ r\ \mathbf{row}\ branch(wt,h,m)\ \mathbf{do}\ d = d + r[c] * deg(r-c,h)\ \mathbf{od};\ d == deg(wt)$

5.6.3. The functions *symtensor* and *alttensor*

It was already mentioned how *symtensor* and *alttensor* can be compared with special cases of *plethysm*. Here is how to check that the second tensor power of a module decomposes into a symmetric and alternating part:

$addmul(alttensor(2,wt), symtensor(2,wt)) == redmulmat(ptensor(2,wt))$

The same simple relation does not hold for third and higher tensor powers, since one needs all plethysms to decompose the tensor power, moreover *plethysm*(λ,μ) occurs a number of times in *ptensor*(n,μ) with $n = |\lambda|$. The number of times it occurs is the dimension of the representation S_n corresponding to λ, .i.e, $\chi^\lambda(e)$, which can be computed by

$chardim(\mathbf{vec}\ lambda) = symchar(lambda, ones(sum(lambda)))$

We can now set up the following test.

$nmul(\mathbf{int}\ n;\ \mathbf{mat}\ m) = \ \#\ \text{multiply multiplicities by}\ n\ \#\ \backslash$
$\quad\mathbf{loc}\ s = colsize(m);\ \mathbf{loc}\ f = id(s);\ f[s,s] = n;\ m * f$

```
chkptensor(int n; vec wt) = loc d = null(0, size(wt) + 1); \
   for lambda row partitions(n) \
   do d = addmul(d, nmul(chardim(lambda), plethysm(lambda, wt))) od; \
   d == redmulmat(ptensor(n, wt))
```

5.7. Extending the function *branch*

The function *branch* is defined only for irreducible modules and simple groups. We now show how the function can be extended within the LiE language to deal with reducible modules and composite groups.

5.7.1. Branching reducible modules

When we are given a decomposition matrix d instead of a highest weight λ, it is not difficult to determine the branching to a subgroup, using the built-in function *branch*. The function *nmul* that multiplies multilpicities by a factor was already defined above; we can now proceed to define

```
branch(mat d; grp h; mat m; grp g) = \
   loc res = null(0, lierank(h) + 1); loc c = colsize(d); \
   for r row d do res = addmul(res, nmul(r[c], branch(r - c, h, m, g))) od; \
   res
```

Note that we can use the same name *branch* as the built-in function has, since they can be distinguished by the types of their arguments, making it clear for instance that this function does not recursively call itself, but rather it calls the built-in function.

5.7.2. Branching in semisimple groups

In order to define *branch* in non-simple groups g, we first need to consider the basic case $g = h \times h$, in which h is embedded diagonally (the restriction matrix consists of two identity matrices on top of each other). In this case a weight for g is just a pair (λ, μ) of weights for h, and branching $V_{(\lambda,\mu)}$ from g to h gives $V_\lambda \otimes V_\mu$, so this case reduces to the function *tensor*. From this one deduces the following procedure for branching in non-simple groups: take the restriction matrix apart into pieces that map the weight lattices of the individual components of g to that of h, compute the branching of the appropriate parts of the weight vector (or decomposition matrix) to h, and then form the tensor product in h of the results.

```
gbranch(vec wt; grp h; mat m; grp g) = \
   loc c = colsize(m); \
   if lierank(h) != c || lierank(g) != rowsize(m) \
      then error("wrong size restriction matrix") fi; \
   loc r = lierank(g[0]); loc wk = null(r); loc mk = null(r, c); \
   loc i = lierank(g) - r; \
   for j = 1 to r do mk[j] = m[i + j]; wk[j] = wt[i + j] od; \
   loc res = [wk * mk + 1];  # torus part, no branching # \
   i = 0; \
   for k = 1 to compsize(g) \
   do r = lierank(g[k]); wk = null(r); mk = null(r, c); \
      for j = 1 to r do mk[j] = m[i + j]; wk[j] = wt[i + j] od; \
      res = tensor(res, branch(wk, h, mk, g[k]), h); i = i + r \
   od; \
   res
```

5.8. Overflow

Due to the choice of type **int** rather than **bin** for matrix and vector entries, vector and matrix operations leading to big integer entries are not to be trusted. In the example below, we found an 'orbit' of length 33, apparently due to the computer's arithmetic modulo 2^{32}.

$r = \mathit{reflection}([1,1,1,1], D_4);\ \ \mathit{orbit}([1,0,0,0], 2 * r)$

5.9. Maximal semisimple subgroups

Information on subgroups of a Lie group can be stored on a file and read whenever convenient. We have begun such a documentation by creating a file MAXSUB, defining two functions giving information about maximal subgroups simple groups of rank ≤ 8, one function giving the subgroup types, the other giving the corresponding restriction matrices (which are needed for *branch*).

5.9.1. Levi subgroups

Before going into the more involved examples, we note that the maximal Levi subgroups, i.e., those fundamental Lie subgroups of which a system of fundamental roots can be obtained by removing a node from the diagram of g, can be dealt with in a uniform way. Here are the definitions of some functions which suffice to determine branching:

$\mathit{levimat}(\textbf{int}\ i) = \mathit{fundam}(\mathit{id}(\mathit{lierank}) - i)$ # remove i-th row and reorder #

$\mathit{levitype}(\textbf{int}\ i) = \mathit{carttype}(\mathit{levimat}(i))$

$\mathit{levidiagram}(\textbf{int}\ i) = \mathit{diagram}(\mathit{levitype}(i))$

$\mathit{levires}(\textbf{int}\ i) = \mathit{resmat}(\mathit{levimat}(i))$

$\mathit{levibranch}(\textbf{vec}\ v;\ \textbf{int}\ i) = \backslash$
  $\textbf{loc}\ m = \mathit{levimat}(i);\ \mathit{branch}(v, \mathit{carttype}(m), \mathit{resmat}(m))$

It will be clear that *levibranch* gives the decomposition matrix of the Levi subgroup of type *levitype*. The diagram printed by *levidiagram* gives the ordering of the fundamental roots of the Levi subgroup, with respect to which ordering the restriction matrix (returned by *levires*) and the resulting decomposition matrix are given.

5.9.2. The functions *maxsub* and *resmat*

This function prints a list of isomorphism types of non-maximal rank maximal semisimple subgroups of g (We believe, but do not guarantee that the list is complete!). Also $\mathit{resmat}(g, g', n)$ returns the restriction matrix of the n-th maximal subgroup of g in the list produced by *maxsub*, that has type g'. We list here only the part of the MAXSUB file pertaining to the types E_6, E_7, E_8, F_4, and G_2; the complete file is part of the LiE package. The first part of the file (more than half of it) deals with auxiliary functions of purely administative nature, then comes the actual data about maximal subgroups.

```
off gc
# Since garbage collection can be time consuming and does not  #
# make sense during reading this file in is better to turn it  #
# off. But do not forget to put a on gc at the and of the file.#
# Global variables: #
stackgroup=T0
resmatgroup=T0
nsubgr=0
sta=T0;stb=T0;stc=T0;std=T0;ste=T0;stf=T0;stg=T0
rga=[[]];rgb=[[]];rgc=[[]];rgd=[[]];rge=[[]];rgf=[[]];rgg=[[]]

equal(grp g,h)=\
if compsize(g)!=compsize(h) then xxeq=0\
else xxeq=1;\
for t=0 to compsize(g) do\
if !(liecode(g[t])==liecode(h[t])) then xxeq=break(0) fi od\
fi; xxeq

prstack()=\
if nsubgr>0 then print(sta) fi;\
if nsubgr>1 then print(stb) fi;\
if nsubgr>2 then print(stc) fi;\
if nsubgr>3 then print(std) fi;\
if nsubgr>4 then print(ste) fi;\
if nsubgr>5 then print(stf) fi;\
if nsubgr>6 then print(stg) fi

# A set of functions in order to put a sequence #
# of groups on the stack: sta, stb,..,stg.      #
stack(grp ga)=nsubgr=1;sta=ga
stack(grp ga,gb)=nsubgr=2;sta=ga;stb=gb
stack(grp ga,gb,gz)=nsubgr=3;sta=ga;stb=gb;stc=gz
stack(grp ga,gb,gz,gd)=nsubgr=4;sta=ga;stb=gb;stc=gz;std=gd
stack(grp ga,gb,gz,gd,ge)=nsubgr=5;sta=ga;stb=gb;stc=gz;\
std=gd; ste=ge
stack(grp ga,gb,gz,gd,ge,gf)=nsubgr=6;sta=ga;stb=gb;stc=gz;\
std=gd;ste=ge;stf=gf
stack(grp ga,gb,gz,gd,ge,gf,gg)=nsubgr=7;\
sta=ga;stb=gb;stc=gz;std=gd;ste=ge;stf=gf;stg=gg

# Find the place j on the stack, such that the i-th #
# appearance of the group g has number j.           #
onstack(grp h;int i)=\
xxon=0;\
for j=1 to nsubgr do\
if equal(h,maxsub(j)) then\
i=i-1;\
if i==0 then xxon=break(j) fi fi od;\
xxon
```

```
# A set of functions in order to put a sequence #
# of matrices on the stack: rga, rgb,..,rgg.    #
resm(mat ga)=rga=ga
resm(mat ga,gb)=rga=ga;rgb=gb
resm(mat ga,gb,gz)=rga=ga;rgb=gb;rgc=gz
resm(mat ga,gb,gz,gd)=rga=ga;rgb=gb;rgc=gz;rgd=gd
resm(mat ga,gb,gz,gd,ge)=rga=ga;rgb=gb;rgc=gz;rgd=gd;rge=ge
resm(mat ga,gb,gz,gd,ge,gf)=rga=ga;rgb=gb;rgc=gz;rgd=gd;rge=ge;\
rgf=gf
resm(mat ga,gb,gz,gd,ge,gf,gg)=\
rga=ga;rgb=gb;rgc=gz;rgd=gd;rge=ge;rgf=gf;rgg=gg

# Getting the n-th matrix on the stack rga,..,rgg. #
resm(int n)=\
if n==1 then ans=rga fi;\
if n==2 then ans=rgb fi;\
if n==3 then ans=rgc fi;\
if n==4 then ans=rgd fi;\
if n==5 then ans=rge fi;\
if n==6 then ans=rgf fi;\
if n==7 then ans=rgg fi;\
*ans

# Some help functions: #
e1()=e(1)
e2()=e(2)
e3()=e(3)
e4()=e(4)
e5()=e(5)
e6()=e(6)
e7()=e(7)
e8()=e(8)
l()=lierank(resmatgroup)
e(int i)=xxxx=null(l); xxxx[i]=1; xxxx
e(int i,j)=xxxx=null(l); xxxx[i]=1; xxxx[j]=xxxx[j]+1; xxxx
e(int i,j,k)=xxxx=null(l); xxxx[i]=1; xxxx[j]=xxxx[j]+1;\
xxxx[k]=xxxx[k]+1; xxxx

# Put all maximal subgroups on the stack #
# and returns them in a list.            #
maxsub(grp g)=\
if liecode(g)[1]<5 || 7<liecode(g)[1] then\
print(\
"Maximal subgroups available only for simple groups of type EFG."\
) fi;\
if !equal(g,stackgroup) then stackfil(g) fi;\
prstack
```

```
# Getting the n-th group on the stack sta,..,stg. #
maxsub(int i)=\
ans=T0;\
if i==1 then ans=sta fi;\
if i==2 then ans=stb fi;\
if i==3 then ans=stc fi;\
if i==4 then ans=std fi;\
if i==5 then ans=ste fi;\
if i==6 then ans=stf fi;\
if i==7 then ans=stg fi;\
ans

# Getting the restriction matrix for the group g with #
# subgroup h. The k indicates the k-th occurrence of h #
# as subgroup. Omitting k is the same as taking k=1. #
resmat(grp g,h;int k)=\
if liecode(g)[1]<5 || 7<liecode(g)[1] then\
print("Resmat available only for simple groups of type EFG.") fi;\
if !equal(g,stackgroup) then stackfil(g) fi;\
xxre=onstack(h,k);\
if xxre==0 then error("Not available as maximal subgroup") fi;\
if !equal(g,resmatgroup) then resmatfil(g) fi;\
resm(xxre)

resmat(grp g,h)=resmat(g,h,1)

# The concrete information for groups of type EFG. #
stackfil(grp g)=\
stackgroup=g;\
if equal(g,E6) then stack(C4,F4,A2,G2,A2G2) fi;\
if equal(g,E7) then stack(A2,A1,A1,A1F4,G2C3,A1G2,A1A1) fi;\
if equal(g,E8) then stack(G2F4,C2,A1A2,A1,A1,A1) fi;\
if equal(g,F4) then stack(A1,A1G2) fi;\
if equal(g,G2) then stack(A1) fi

resmatfil(grp g)=\
resmatgroup=g;\
if equal(g,E6) then resm(\
[e(3,5),e(1,6),[0,0,1,2,1,0],e2],\
[e2,e4,e(3,5),e(1,6)],\
[[2,1,2,5,5,2],[2,4,5,5,2,2]],\
[[2,1,2,5,2,2],e(2,3,5)],\
[e(1,3,4)+e(2,3),e(4,5,6)+e(5,2),e1+e(4,6,2),e(3,4,5)])\
fi;\
if equal(g,E7) then resm(\
[[4,7,9,11,10,6,6],[4,4,6,11,7,6,0]],\
[[34,49,66,96,75,52,27]],\
[[26,37,50,72,57,40,21]],\
[[0,1,0,2,1,2,1],e1,e(3,4),e(5,6,2),e(4,5,7)],\
[[1,0,2,1,1,2,1],e(4,5,2),[0,0,1,1,1,0,1],\
[1,0,0,1,1,1,0],e(3,4,2)],\
[[2,3,4,4,5,4,1],[2,1,2,4,4,1,0],[0,1,1,1,0,1,1]],\
[[4,8,10,18,12,8,6],[6,7,10,12,11,8,3]])\
fi;\
if equal(g,E8) then resm(\
[[1,0,2,1,1,2,1,1],e(4,5,2),e(5,6,7),\
e(2,3,4),e1+e(4,5,6),e(7,8)+e(3,4,5)],\
[[4,6,8,16,12,8,8,2],[4,6,8,9,8,7,3,3]],\
[[8,12,16,22,16,14,10,6],[2,3,4,8,6,4,4,1],[2,3,4,5,6,4,1,1]],\
[[72,106,142,210,172,132,90,46]],\
[[60,88,118,174,142,108,74,38]],\
[[92,136,182,270,220,168,114,58]])\
fi;\
if equal(g,F4) then resm(\
[[22,42,30,16]],\
[[4,4,4,2],e(1,2,4),e(2,3)]) fi;\
if equal(g,G2) then resm([[6,10]]) fi
on gc
```

LE Manual

Chapter 6. SYNTAX

In this chapter the complete formal syntax accepted by the interpreter is given for reference. It is given in the usual form of a BNF context free grammar. Literally represented symbols are given in `typewriter type`, and every rule is terminated by a period. At the end ellipses (...) occur twice, we assume that the reader is familiar with the enumeration of the alphabet. The syntax includes a few cases that are not described anywhere in this manual, such as the command `type` ⟨expression⟩. These are of little or no interest to the average user, but you may experiment if you like; the reason we give them here is mainly because one should be aware that the identifiers ocurring in these rules *are* reserved words, and should not be used for variables or functions.

⟨command⟩ ::= ⟨series⟩ | ⟨function definition⟩ | `learn` ⟨tail⟩ | `listvars`
  | `listfuns` | `listops` | ⟨on⟩ ⟨identifier⟩ | `off` ⟨identifier⟩
  | ⟨on⟩ | `off` | `read` ⟨tail⟩ | `edit` ⟨tail⟩ | `edit` | `write` ⟨tail⟩
  | `exec` ⟨tail⟩ | `monfil` ⟨tail⟩ | `type` ⟨arithmetic expr⟩ | ⟨quit⟩
  | ⟨quit⟩ ⟨tail⟩ | ⟨help⟩ ⟨subject⟩ | ⟨help⟩ ⟨subject⟩ `>` ⟨tail⟩
  | ⟨help⟩ ⟨subject⟩ `>>` ⟨tail⟩ | `:` ⟨tail⟩ | ⟨empty⟩ .

⟨series⟩ ::= ⟨statement⟩ | ⟨statement⟩ `;` ⟨series⟩ | ⟨statement⟩ `;` .

⟨statement⟩ ::= ⟨assignment⟩ | ⟨expression⟩ | `return` ⟨expression⟩
  | `break` ⟨expression⟩ | `return` | `break` | `setdefault`
  | `setdefault` ⟨expression⟩ | `;` .

⟨assignment⟩ ::= ⟨identifier⟩ `=` ⟨expression⟩ | `loc` ⟨identifier⟩ `=` ⟨expression⟩
  | ⟨arithmetic expr⟩ `+=` ⟨expression⟩ | ⟨selection⟩ `=` ⟨arithmetic expr⟩ .

⟨expression⟩ ::= ⟨arithmetic expr⟩ | ⟨logical expr⟩ .

⟨arithmetic expr⟩ ::= ⟨variable⟩ | ⟨number⟩ | ⟨group⟩
  | ⟨string⟩ | ⟨arithmetic expr⟩ ⟨operator⟩ ⟨arithmetic expr⟩
  | `-` ⟨arithmetic expr⟩ | `*` ⟨arithmetic expr⟩ | `X` ⟨arithmetic expr⟩
  | `(` ⟨arithmetic expr⟩ `)` | ⟨selection⟩ | `[` ⟨list option⟩ `]`
  | ⟨block⟩ | ⟨identifier⟩ `( )` | ⟨identifier⟩ `(` ⟨list option⟩ `)`
  | ⟨conditional expr⟩ | ⟨loop⟩ | `make (` ⟨variable⟩ `,` ⟨arithmetic expr⟩ `)`
  | `make (` ⟨variable⟩ `,` ⟨arithmetic expr⟩ `,` ⟨arithmetic expr⟩ `)`
  | ⟨apply⟩ `(` ⟨variable⟩ `,` ⟨arithmetic expr⟩ `,` ⟨arithmetic expr⟩ `)` .

⟨logical expr⟩ ::= ⟨arithmetic expr⟩ ⟨relation⟩ ⟨arithmetic expr⟩
  | ⟨expression⟩ ⟨boolean operator⟩ ⟨expression⟩ | `!` ⟨expression⟩
  | `(` ⟨logical expr⟩ `)` .

⟨selection⟩ ::= ⟨arithmetic expr⟩ `[` ⟨list option⟩ `]`
  | ⟨arithmetic expr⟩ `|` ⟨arithmetic expr⟩ .

⟨ variable ⟩ ::= ⟨ identifier ⟩ | ⟨ sysident ⟩ .

⟨ conditional expr ⟩ ::= if ⟨ expression ⟩ then ⟨ series ⟩ else ⟨ series ⟩ fi
  | if ⟨ expression ⟩ then ⟨ series ⟩ fi .

⟨ loop ⟩ ::= for ⟨ identifier ⟩ = ⟨ arithmetic expr ⟩ to ⟨ arithmetic expr ⟩ do ⟨ series ⟩ od
  | for ⟨ identifier ⟩ in ⟨ arithmetic expr ⟩ do ⟨ series ⟩ od
  | for ⟨ identifier ⟩ row ⟨ arithmetic expr ⟩ do ⟨ series ⟩ od
  | while ⟨ expression ⟩ do ⟨ series ⟩ od .

⟨ function definition ⟩ ::= ⟨ identifier ⟩ (⟨ formals ⟩) = ⟨ series ⟩
  | ⟨ identifier ⟩ ( ) = ⟨ series ⟩ | ⟨ identifier ⟩ (⟨ formals ⟩) { ⟨ series ⟩ }
  | ⟨ identifier ⟩ ( ) { ⟨ series ⟩ } .

⟨ formals ⟩ ::= ⟨ type ⟩ ⟨ variables ⟩ | ⟨ type ⟩ ⟨ variables ⟩ ; ⟨ formals ⟩ .

⟨ variables ⟩ ::= ⟨ variable ⟩ | ⟨ variable ⟩ , ⟨ variables ⟩ .

⟨ list option ⟩ ::= ⟨ list ⟩ | ⟨ empty ⟩ .

⟨ list ⟩ ::= ⟨ expression ⟩ | ⟨ expression ⟩ , ⟨ list option ⟩ .

⟨ block ⟩ ::= { ⟨ series ⟩ } | { ⟨ series ⟩ } (⟨ list ⟩) .

⟨ on ⟩ ::= on | on ⟨ number ⟩ | on + | on - .

⟨ empty ⟩ ::= .

⟨ number ⟩ ::= ⟨ digit ⟩ | ⟨ digit ⟩ ⟨ number ⟩ .

⟨ digit ⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .

⟨ identifier ⟩ ::= ⟨ lower case letter ⟩ | ⟨ identifier ⟩ ⟨ letter or digit ⟩ .

⟨ lower case letter ⟩ ::= a | b | ... | z .

⟨ letter or digit ⟩ ::= ⟨ lower case letter ⟩ | ⟨ digit ⟩ | _ | A | ... | Z .

⟨ sysident ⟩ ::= $ | $ ⟨ number ⟩ .

⟨ group ⟩ ::= ⟨ simple group ⟩ | ⟨ group ⟩ ⟨ simple group ⟩ .

⟨ simple group ⟩ ::= ⟨ family ⟩ ⟨ number ⟩ .

⟨ family ⟩ ::= A | B | C | D | E | G | T .

⟨ operator ⟩ ::= + | - | * | / | % | ^ | X | Y .

⟨ relation ⟩ ::= == | != | < | > | <= | >= .

⟨ boolean operator ⟩ ::= && | || .

⟨ string ⟩ ::= " { any sequence of characters except "\"" and newline } " .

⟨ tail ⟩ ::= { any sequence of characters except "(" and newline } .

⟨ help ⟩ ::= help | ? .

⟨ subject ⟩ ::= ⟨ empty ⟩ | { any sequence of characters not including spaces, newline, parentheses or ">" } .

⟨ quit ⟩ ::= quit | exit | @ .

⟨ type ⟩ ::= int | vec | mat | trm | pol | grp .

⟨ apply ⟩ ::= iapply | vapply | mapply .

LiE Manual
Chapter 7. REFERENCES

A list of the main books and papers that have been of use and/or influence to us while preparing Lie and may be of use to anyone wishing to be familiarised with Lie groups. As for a survey of the field, the list is far from complete.

[And 1977]	C. M. Andersen, *Clebsch-Gordan series for symmetrized tensor products*, J. Math. Phys., **8** (1977), 988-997.
[BecKol 1977]	R. E. Beck & B. Kolman (eds.), *Computers in Nonassociative Rings and Algebras*, Acad. Press, New York, 1977.
[Bourb 1968]	N. Bourbaki, *Groupes et algèbres de Lie, Chap 4, 5, et 6*, Hermann, Paris, 1968.
[Bourb 1975]	N. Bourbaki, *Groupes et algèbres de Lie, Chap 7 et 8*, Hermann, Paris, 1975.
[Brem ea 1985]	M. R. Bremner, R. V. Moody, J. Patera, *Tables of dominant weight multiplicities for representations of simple Lie algebras*, Monographs and Textbooks in Pure and Appl. Math. 90, Dekker, New York, 1985.
[BrouCoh 1985]	A. E. Brouwer & A. M. Cohen, *Computation of some parameters of Lie geometries*, Annals of Discrete Math., **26** (1985), 1-48.
[CohGri 1987]	A. M. Cohen & R. L. Griess, *On finite simple subgroups of the complex Lie group of type E_8*, pp. 367-405 in: Proc. of Symp. in Pure Math. 47[2] (eds.: P. Fong), Amer. Math. Soc., Providence, 1987.
[Hum 1974]	Humphreys, J. E., *Introduction to Lie algebras and representation theory*, Springer, New York, 1974.
[Jac 1962]	N. Jacobson, *Lie algebras*, Wiley & Sons, New York, 1962.
[JamKer 1981]	G. James & A. Kerber, *The Representation Theory of the Symmetric Group*, Addison-Wesley, Reading MA, 1981.
[Kruse 1971]	M. I. Krusemeyer, *Determining multiplicities of dominant weights in irreducible Lie algebra representations, using a computer*, BIT, **11** (1971), 310-316.
[McKay ea 1981]	W. G. McKay & J. Patera, *Tables of dimensions, indices and branching rules for representations of simple Lie algebras*, Lecture Notes in Pure and Appl. Math. 69, Dekker, New York, 1981.
[MoodPat 1984]	R. V. Moody & J. Patera, *Characters of elements of finite order in Lie groups*, SIAM J. Alg. Discr. Meth., **5** (1984), 359-383.

[Serre 1987] J.-P. Serre, *Complex Semisimple Lie algebras*, Springer Verlag, Berlin, 1987.

[Tits 1967] J. Tits, *Tabellen zu den einfachen Lie Gruppen und ihren Darstellungen*, Lecture Notes in Math. 40, Springer, Berlin, 1967.

LiE Manual
Chapter 8. INDEX

In this index you will find all functions, and operators defined in LiE, and many of the commands, keywords and terms that are used. When a term coincides with the name of a function, references to both the term and the function are listed after the function name.

LiE MANUAL

Table of Contents