LiE MANUAL

describing LiE version 2.1

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

[BMP] 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.

[McKPa] 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 on-line 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) 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 has been tested on following computers: SGI and SUN workstations. Since August 1996, LiE is available for free

through the WWW, or by anonymous ftp. For information about acquiring LiE see the WWW page `http://www.cwi.nl/~maavl/LiE`, or contact by anonymous ftp the machine `ftp.cwi.nl` and go to the directory `pub/maavl`. You may also contact the authours by electronic mail: Arjeh M. Cohen `amc@win.tue.ln` or Marc van Leeuwen `M.van.Leeuwen@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 on-line 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`	types of simple factors ($T_4 = (\mathbf{C}^*)^4$)
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; they involve 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 semisimple elements, and the spectra of such elements on modules. In order to describe these functions, it is necessary to introduce the relevant mathematical terms and concepts, and the way in which they 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 how this can be done 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 give names only to 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 of the mathematical functions and a first version of this manual, and is the project leader. Marc van Leeuwen is the main author of this manual, has implemented a number of algorithms, has rewritten most of the mathematical functions to comply with the standards of version 2.0, and has since reworked most of the code, improving algorithms and documenting it according to the literate programming paradigm. Bert Lisser has built the interpreter, provided the information for Chapter 2 of this manual, and has been the main engineer of the project throughout. 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.

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, which 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 in Section 2.4, the functions which are not related to Lie groups. Section 2.5 discusses the ingredients needed to construct larger programs, and Section 2.6 shows how to define ones own functions. Finally Section 2.7 describes some features providing the user with additional control over LiE. A note about the typography: in the introductory section of this chapter, 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 remainder of this manual a more aesthetically pleasing form of rendering expressions is chosen, distinguishing identifiers (*italic* type), keywords (**bold** type) and commands directly addressed to the interpreter of LiE (`typewriter` type).

2.1. A first look

After LiE has been installed on your computer (see the leaflet accompanying the software package for instructions about how to do this) an interactive session of LiE can be started by entering the command `LiE` to the computer (or by whatever other means the operating environment may provide for starting up programs). 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, which if necessary will invoke some of the mathematical functions, or perform some other action. 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.

A first concern after entering LiE is of course how one can get out again; to this end 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 form of commands are those instructing LiE to perform arithmetic computations; the interpreter then behaves like a pocket calculator, except that it only deals with integral values, which may on the other hand be arbitrarily large. So one may simply enter an arithmetic expression, which will then be evaluated and the resulting value is printed.

```
> 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 also performs calculations with vectors and matrices with integer entries. The ordinary arithmetic operators can be used to indicate vector and matrix addition and multiplication. In addition an interpretation is often given to these operators when used with certain other combinations of operands, including some cases where no such meaning is generally accepted. It has been proven quite useful however to have a simple notation for certain frequently used operations (mostly with operands of mixed types); the arithmetic operators lend themselves well to the purpose, even if these operations do not always satisfy the same laws as the arithmetic operations usually associated with the operators. Here are a few operations with vectors; rather than giving complete definitions of the operations invoked—this will be done further on in this chapter—we invite the reader to try to guess the definitions from the examples.

```
> 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", whose first entry gives the exponent of the first indeterminate, the second entry the exponent of the second indeterminate, etc. Moreover, LiE will not mix terms with different numbers of indeterminates, so zeros should be added if necessary to make all exponents the same size. Finally negative integer entries are allowed in the exponents, so we are in fact dealing with Laurent polynomials with coefficients in $\mathbf{Z}$. 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]
> $^2
      16X[-2,8] - 24X[ 0,6] +  9X[ 2,4]
> $+X[6,7,8]
Number of variables in polynomials unequal
( 2 <-> 3 variables).
(in +)
> (4X[-1,4]-3X[1,2])*(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)

        0 2
        |
        |
0---0---0---0---0---0---0
1   3   4   5   6   7   8
E8
```

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

The user may also define in 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+l >= 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`' already mentioned above, and others to control the input and output flow. Some examples of the latter are

```
on monitor
edit script
```

the first of which starts recording the session on a file "`monfil`", and the second of which invokes an editor on the file "`script`". (Note that some of these commands are machine dependent: some computer systems do not provide a standard editor, and for such computers an `edit` command might be simply ignored). The file name `monfil` is the default name LiE chooses, although it can be altered by the `monfil` command. The name `script` on the other hand could be replaced by any other file name; that file should however contain only LiE commands (and comments), because the file will be read by LiE automatically upon completion of the edit session.

Finally, there are some features of general assistance, such as

```
? functions
listvars
learn lie group
```

The first of these lists all mathematical functions built into LiE, the second lists all variables that the user has been given a value, and the third 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**, which stands for void; this is not really a type since there is no void value that could be assigned to a variable or passed to a function, but it is used to indicate the result type of a function that does not return any value. Variables do not have a declared type: they simply assume the type of any value that is assigned to them. However, once they have been created, variables cannot change their type during a computation (i.e., from within a user defined function or a loop or even from within a conditional statement): their type can only change by an assignment typed directly by the user.

We end this introductory section with a few details that are important to know when you start using LiE.

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 `?Lie_rank` will return information on the built-in function *Lie_rank*. For built-in functions, similar information can also be found in Chapter 4 of this manual.

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

2.1.3. Identifiers

Identifiers, which represent variables or functions, consist of a sequence of letters (lower or upper case), digits and underscores; the first character must be a letter, and if it is an upper case letter it must be immediately followed by a non-digit (this requirement is necessary in order to be able to distinguish between identifiers and groups).

There is also a special variable **$**, which contains at any time the value returned by the last command that did in fact deliver a value (so assignments and calls for help etc. do not alter the value of **$**). 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 by entering the command `read` ⟨name⟩. The same file can be edited by issuing the command `edit` ⟨name⟩. Once a filename has been given it becomes the default, and can be omitted from following `read` or `edit` commands. When the editing session is completed, the file is automatically read in. There are also possiblities for recording results permanently on output files; see Section 2.7.1 for further details.

2.1.5. Comments

Comments may be given anywhere within a line of input, and are enclosed between a pair of characters '`#`'. If there is no closing '`#`' on the same line as the opening one, the comment is closed at the end of the line.

2.1.6. Escape to the shell

The character '`:`' appearing as the 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). It is processed by 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). If a sequence of commands is to be given to the shell, it can be convenient to type '`: sh`' or '`: csh`', and to return to LiE after the sequence of commands by typing '`exit`'.

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 greater detail.

2.2.1. Integer

Integers are represented in LiE by values of type **int**. As usual, they may be denoted by a sequence of digits, optionally preceded by a minus sign (although, strictly speaking '-1' is already a formula, consisting of the monadic '$-$' operator applied to the positive integer 1).

Integers and coefficients of polynomials effectively have unlimited length. In contrast, the integer entries of vectors, matrices and the exponents in polynomials are restricted by the word size of the machine (usually this allows values up to 2^{31}

in magnitude to be distinguished). 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 this limit. Note that whereas a warning is issued if one tries to insert too big an integer into a vector, matrix, or polynomial exponent, no such warning is generated when overflow occurs within an operation on vectors, matrices or (quite unlikely) polynomials themselves, e.g., when calculating a high 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]` or `[]`. It is also possible to denote vectors whose size is determined at run time by means of the function calls *null*(n) and *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 $1 \leq i \leq n$; if v is in fact a vector variable (rather than some other vector valued expression) then assignment to the individual entries $v[i]$ is possible. 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, consisting 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],[4-7,11]]`, or `[v,-v,[35,61],4*v]` after having assigned `v=[6,9]`. Since matrices are always rectangular, it is required that all vectors occurring in the list have the same size: they will be taken in order to form the successive rows of the matrix. Note that although it is possible to denote matrices with 0 columns in this way, one cannot represent matrices with 0 rows by such an expression; such matrices can however be created with the a call of the form *null*$(0, n)$. Very often matrices in LiE are interpreted as sets or sequences of vectors, in which case these are always taken to be the rows (rather than the columns) of the matrix. The notation for entering matrices is in accordance with this convention. The functions *null* and *all_one* can be used to create matrices as well as vectors: to this end they should be called with two integer arguments, the first of which determines the number of rows, and the second the number of columns of the matrix. Like with vectors, the entries are either all set to 0 or to 1, depending on whether *null* or *all_one* was used.

A matrix is printed in the same way as it is entered, with the vectors representing the rows on separate lines, and with the opening and closing brackets and commas of the outer level in the first non-empty column (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). For matrices with 0 rows this method would lead to ambiguity; therefore `null(0,`n`)` is printed instead in such cases.

Since a matrix is often viewed as a sequence of its rows, the i-th row of 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 either as $m[i,j]$ or as $m[i][j]$, both denoting the same entry. If m is a matrix variable then assignment is possible both to $m[i]$ and to $m[i,j]$ (but not to $m[i][j]$ because although $m[i]$ is a vector expression, it is not a variable). If m is a matrix and v is a vector, then the expression $m|v$ denotes the index of the first row of m which is equal to v, or 0 if no such row exists. 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*, 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. Terms with equal exponents 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 has a single 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 exponent as a single integer, which will be 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) and *poly_one*(n) respectively; these calls are equivalent to $0X$ *null*(n) and $1X$ *null*(n).

The terms of a polynomial are always automatically sorted; the criterion by which they 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 exponent entries takes precedence over the lexicographic ordering) or *height ordering* (which is useful when the exponents are weights for the default group: the term with highest weight as exponent will be sorted into the last position) and the reverse ordering of any of these possibilities. See Section 2.7.4 for the commands to select the sorting criterion. This ordering influences the selection of terms: the i-th term of a polynomial p can be referred to as $p[i]$. Terms are not a separate data type in LiE however, so the selection will return a polynomial consisting of a single term. The coefficient of the i-th term of p can be obtained as *coef*(p,i), and the exponent of that term as *expon*(p,i) (this is a vector).

Further functions to obtain information about polynomials are *n_vars*, giving the number of indeterminates, *length*, giving the number of non-zero terms, *deg*, 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 when 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 assigning another term to it, but note that because the polynomial is normalised afterwards by possibly rearranging and merging of terms, it is not generally true after assigning $p[i] = term$ that the condition $p[i] == term$ holds.

2.2.5. Group

A value of type **grp** specifies an isomorphism class of reductive complex Lie groups with simply connected semisimple part. An object of type **grp** records a (possibly empty) sequence of simple groups, and the dimension of the central torus. For details about the meaning of these terms we refer to Section 3.1. At this point it suffices to know that each simple group is encoded in 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 further restrictions that $n \geq 2$ 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 $L = G$. An n-dimensional torus is encoded as 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 return `C3B12A4A1E7T10` (fortunately the most frequently used groups do not have such a long sequence of simple factors).

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 $g[0]$. For g as in the above example, we have $g[0] = T_{10}$, $g[1] = C_3$, $g[2] = B_{12}$, etc.

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 doubling any double quotes ocurring in the string itself, for instance `"Say ""hello!"""`, which prints as '`Say "hello"`'. Whenever two string denotations are separated by white space only they are concatenated by LiE, so very long strings may be given on successive lines, with each opening quote being closed on the same line. It is not possible to put a newline character into a string, but after each *print* statement a newline is issued. To display several results and pieces of text on a single line concatenation using the '+' operator can be used.

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 expressions 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'; rather 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 to represent **true**. There are however syntactic restrictions that prevent performing 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 have included these conditions in braces, where σ_a stands 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**	$[a[1], a[2], \ldots, a[\rho_a], b] \quad \{\kappa_a = \sigma_b\}$
		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	**vec**	**tex**	$a + t$ where t is textual representation of b
		vec	**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**	$a - b \quad \{\sigma_a = \sigma_b\}$
		mat	**mat**	**mat**	$a - b \quad \{\rho_a = \rho_b, \kappa_a = \kappa_b\}$
		pol	**pol**	**pol**	$a - b \quad \{\nu_a = \nu_b\}$
		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) $\quad \{\sigma_a = \sigma_b\}$
		int	**mat**	**mat**	$a \cdot b$ (scalar multiplication by a)
		vec	**mat**	**vec**	ab (right multiplication by matrix b) $\quad \{\sigma_a = \rho_b\}$
		mat	**mat**	**mat**	ab (matrix multiplication) $\quad \{\kappa_a = \rho_b\}$
		mat	**vec**	**vec**	$ba^\top = (ab^\top)^\top$ (left multiplication of column vector b by matrix a) $\quad \{\kappa_a = \sigma_b\}$
		int	**pol**	**pol**	$a \cdot b$ (scalar multiplication by a)
		pol	**pol**	**pol**	$a * b$ (polynomial multiplication) $\quad \{\nu_a = \nu_b\}$
		pol	**mat**	**pol**	multiply all exponents of terms of a on the right by b and normalise result $\quad \{\nu_a = \rho_b\}$
		pol	**int**	**pol**	$a * (b * id(\nu_a))$ (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)
		pol	**int**	**pol**	Replace each coefficient c of a by c/b.

		pol	**vec**	**pol**	For each i with $1 \leq i \leq \nu_a$, replace the i-th entry e_i of any exponent of a by $e_i/b[i]$. $\{\nu_a = \sigma_b\}$
%	7	**int**	**int**	**int**	$a \bmod b \quad \{b > 0; 0 \leq a \bmod b < b\}$
		vec	**int**	**vec**	$[a[1] \bmod b, \ldots, a[\sigma_a] \bmod b]$
		mat	**int**	**mat**	$[a[1] \,\%\, b, \ldots, a[\rho_a] \,\%\, b]$ (see previous line)
		pol	**int**	**pol**	Replace each coefficient c of a by $c \,\%\, b$.
		pol	**vec**	**pol**	For each i with $1 \leq i \leq \nu_a$ and $b[i] \neq 0$, replace the i-th entry e_i of any exponent of a by $e_i \,\%\, b[i]$. $\{\nu_a = \sigma_b\}$
^	8	**int**	**int**	**int**	a^b
		mat	**int**	**mat**	a^b (matrix power) $\{\rho_a = \kappa_a\}$
		pol	**int**	**pol**	a^b (polynomial power)
		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) $\{\kappa_a = \kappa_b\}$
		pol	**pol**	**pol**	$a' * b'$ where a' is obtained by adding ν_b zeros to every exponent of a, and b' by prefixing every exponent of b by ν_a zeros (multiplication with disjoint sets of indeterminates)
!	3	**vec**		**vec**	$[a[\sigma_a], \ldots, a[1]]$ (reversal)
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)
		grp	**grp**	**int**	$a = b$ (order of simple factors is relevant)
		tex	**tex**	**int**	$a = b$
!=	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)

As suggested by the comments, the logical operators '| |' and '&&' are lazy with respect to their second argument: this argument is not evaluated if the outcome of the relation is already determined by that of the first argument. Therefore conditions such as '$i > 0$ && $i \leq size(v)$ && $v[i] > 0$' may be safely evaluated. Apart from these two cases, it is not defined in which order the operands of an operator are evaluated. As a general rule the order of evaluation of subexpressions is not defined unless a specific order is explicitly stated in this manual (so one should not depend on any specific order used to evaluate function arguments, the entries of a vector expression, etc.).

As stated earlier, many meanings are attached to arithmetic operators, not all of which are customary. Most of these meanings have proven to be useful operations in writing programs, but one is warned that the usual arithmetic identities are no longer valid for such uses of operators. For instance, if v is a vector, then $v-3$ means something quite different from $v+-3$, and similarly $(v+4)-2$ differs from $v+(4-2)$ (due to left associativity of operators of equal priority, the parentheses could have been omitted in the former expression). Also, if i is an integer and p a polynomial, then in $i*p$ the coefficients of p are multiplied by i, while in $p*i$ the exponents of p are, so there is no commutativity in this case.

For the inverse of these operations on p, namely the two ways of "dividing by i", it would be unnatural to use the order of the operands as a means of discrimination. Therefore p/i means division of the coefficients of p by i, but division of the exponents by i can be achieved by $p/[i,\ldots,i]$ (to be precise: by $p/(i*all_one(n_vars(p)))$). Division of a polynomial by a vector can be used more generally to divide the exponents of each indeterminate by a separate value. These operations have their analogs for '%' in place of '/'; in the case of "polynomial modulo vector" an exponent position may be left unchanged by supplying a zero entry in the vector. There is no exact inverse of the operation of dividing a polynomial by vector, but the operation "polynomial times matrix" provides an even more general operation; this operation is quite natural and useful when polynomials are used to encode sets of vectors with multiplicities, as is often done in LiE. Another operation whose utility stems from this use of polynomials, and from using matrices to represent sets of vectors without multiplicities, is the operation X from matrices to polynomials. The function *support*, treated in Section 2.4.2, provides a transition in the opposite direction, which forgets the multiplicities.

2.4. Using functions

2.4.1. Function call

A function call has the form

⟨ name ⟩ (⟨ arg_1 ⟩, . . . , ⟨ arg_n ⟩)

where ⟨ name ⟩ is the name of the function to be called, and ⟨ arg_1 ⟩, . . . , ⟨ arg_n ⟩ are arbitrary expressions giving the actual arguments of the function; among the possibly numerous definitions for the given function name, the unique 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 empty parentheses after a name that refers to a variable, but this seems to be needlessly misleading.) Whenever a function is called, its arguments are evaluated first.

2.4.2. Basic functions

A number of built-in functions which are not of Lie group theoretic nature 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
gcd	**vec** v	**int**	the greatest common divisor of all 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
sort	**vec** v	**vec**	vector with same entries as v, but sorted into decreasing order
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
vec_mat	**mat** m	**vec**	concatenation of rows of m: $m[1]\hat{}m[2]\hat{}\cdots$
mat_vec	**vec** v; **int** n	**mat**	matrix with n columns, and rows $[v[1], \ldots, v[n]]$, $[v[n+1], \ldots, v[2n]], \ldots$ $\{n$ divides $\sigma_v\}$
block_mat	**mat** a, b	**mat**	the block matrix $\left(\begin{smallmatrix} a & 0 \\ 0 & b \end{smallmatrix}\right)$
sort	**mat** m	**mat**	matrix with same rows as m, but sorted using the same criterion as selected for polynomial exponents

unique	**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).
row_index	**mat** m; **vec** v; **int** l, u	**int**	the first index i with $l \leq i \leq u$ and $m[i] = v$, or 0 if none exist.
n_vars	**pol** p	**int**	the number ν_p of indeterminates of p
length	**pol** p	**int**	the number of non-zero terms of p
deg	**pol** p	**int**	the total degree of p
coef	**pol** p; **int** n	**int**	the coefficient of the n-th term of p
expon	**pol** p; **int** n	**vec**	the exponent of n-th term of p
support	**pol** p	**mat**	A matrix whose rows are the exponents of p
poly_null	**int** n	**pol**	the zero polynomial in n indeterminates
poly_one	**int** n	**pol**	the unit polynomial in n indeterminates
n_comp	**grp** g	**int**	the number of simple components of g
length	**tex** s	**int**	The length of the string s
fmt	**int** n, w	**tex**	place digits of n in field of width at least $\|w\|$, left-justifying if $w < 0$.
void	any x	**vid**	no result, useful to force void type, for instance to make types match between branches of a conditional clause
print	any x	**vid**	print the value of x
used		**int**	the number of objects currently in use
gcol		**vid**	invoke the garbage collector, see Section 2.7.3.
error	**tex** t	**tex**	print text t and terminate; prompt for new command

The following remarks may be made. The function *unique* uses a heapsort algorithm, rather than the quicksort algorithm used by *sort*, which avoids internal stack build-up, and guarantees that the running time is independent of any ordering already present in the input to *unique* (the sorting implicitly applied to the terms of polynomials, whenever necesssary, is also done by heapsort). The result of *unique*(m) is equal to *support*($X\, m$). The single term of a zero polynomial does not influence the outcome of the functions *length* and *support*, but the total degree of a zero polynomial is taken to be 0, lacking a proper way to represent $-\infty$. Finally note that the expressions $m|v$ and $p|v$ for a matrix m, polynomial p and vector v (which didn't appear in the tables since they are syntactically neither formulae nor function calls) are analogous to *row_index*, but their range of search cannot be restricted by a lower and upper bound. If the functionality of *row_index* is needed for polynomials, it may be used in combination with *support* (which preserves the order into which the exponents were sorted).

2.5. Statements and clauses

We have treated the main ways of building 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). A precise description of the relationships between these categories can be found in the syntax which is given in Chapter 6. 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 three somewhat specialised kinds of statements, namely the **break**, **return** and **setdefault** statements. We note that the commands `on` and `off`, as well as the related commands `savestate` and `restorestate`, which are described in Section 2.7.4, also qualify syntactically as 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.

⟨ identifier ⟩ = ⟨ expression ⟩

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 ⟨ expression ⟩. For more details see Section 2.6.2.

⟨ identifier ⟩ [⟨ expression_1 ⟩] = ⟨ expression_2 ⟩

Here ⟨ identifier ⟩ must denote a vector, matrix or polynomial variable, and correspondingly ⟨ expression_2 ⟩ must be of type integer, vector, or polynomial respectively, while ⟨ expression_1 ⟩ must be of type integer in all cases. Both expressions are evaluated, and the value of ⟨ expression_2 ⟩ 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 ⟨ expression_1 ⟩. In the case of a matrix or polynomial variable it is required that the yield of ⟨ expression_2 ⟩ has the same size as the row or term replaced by it; in particular it may not be a polynomial of length > 1.

⟨ identifier ⟩ [⟨ expression_1 ⟩, ⟨ expression_2 ⟩] = ⟨ expression_3 ⟩

Here ⟨ identifier ⟩ must denote a matrix variable, and all expressions must be of type integer; the value yielded by ⟨ expression_3 ⟩ replaces the entry of the matrix variable whose indices are the values yielded by ⟨ expression_1 ⟩ and ⟨ expression_2 ⟩.

⟨ identifier ⟩ | ⟨ expression_1 ⟩ = ⟨ expression_2 ⟩

Here ⟨ identifier ⟩ must denote a polynomial variable, ⟨ expression_1 ⟩ must be of type vector and ⟨ expression_2 ⟩ of type integer. The term of the polynomial is searched whose exponent coincides with the value of ⟨ expression_1 ⟩ (if none exists, a new such

term with coefficient 0 is created), and the coefficient of that term is replaced by the value of ⟨ expression$_2$ ⟩.

⟨ variable ⟩ += ⟨ expression ⟩

where ⟨ variable ⟩ has one of the forms

⟨ identifier ⟩
⟨ identifier ⟩ [⟨ expression$_1$ ⟩]
⟨ identifier ⟩ [⟨ expression$_1$ ⟩, ⟨ expression$_2$ ⟩]

In each case the statement is equivalent to

⟨ variable ⟩ = ⟨ variable ⟩ + ⟨ expression ⟩

except that the expressions in brackets, if present, are only evaluated once; moreover 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 occur as constituent parts of clauses. A series is nothing more than a sequence of statements, separated by semicolons:

⟨ statement$_1$ ⟩; ⟨ statement$_2$ ⟩; ⋯; ⟨ statement$_n$ ⟩

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

2.5.3. Blocks

The first and simplest kind of clause that we shall treat is a block. It is formed by enclosing a series in braces:

{ ⟨ series ⟩ }

A block is an expression, which allows the value of a series to enter into larger expressions. 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. The command

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

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

2.5.4. Conditional clauses

There are two forms of conditional clauses:

if ⟨ expression ⟩ **then** ⟨ series$_1$ ⟩ **else** ⟨ series$_2$ ⟩ **fi**

and

if ⟨ expression ⟩ **then** ⟨ series$_1$ ⟩ **fi**

In each case ⟨ expression ⟩ is evaluated first; if the (integer) value yielded is unequal to 0 then ⟨ series$_1$ ⟩ is evaluated and its value becomes the value of the conditional clause, and otherwise ⟨ series$_2$ ⟩ is evaluated if present and its value becomes that of the

conditional clause. In the second form of the conditional expression, where ⟨ series$_2$ ⟩ is absent, it is required that ⟨ series$_1$ ⟩ has void type, so that no value is yielded either way, and the type of the conditional expression can be established before executing it.

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 ⟨ expression ⟩ **do** ⟨ series ⟩ **od**

When a while loop is executed, the ⟨ expression ⟩ is first evaluated; if it yields 0 then the execution of the loop terminates, and otherwise the ⟨ series ⟩ 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 ⟨ series ⟩, or void if the ⟨ expression ⟩ had value 0 the first time it was evaluated.

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

for ⟨ identifier ⟩ = ⟨ expression$_1$ ⟩ **to** ⟨ expression$_2$ ⟩ **do** ⟨ series ⟩ **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 ⟨ expression$_1$ ⟩ is assigned to the loop variable, and the value of ⟨ expression$_2$ ⟩ 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 ⟩ = ⟨ expression$_1$ ⟩ **downto** ⟨ expression$_2$ ⟩ **do** ⟨ series ⟩ **od**

It is identical to the first from, except that the loop variable is *decremented* at the end of the loop, and that the loop terminates when the loop variable becomes *smaller* than the limit computed in ⟨ expression$_2$ ⟩.

The third 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* += *entry* **od**; *s*

The last form is analogous to the third, 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. This form of loop clause is particularly useful since ⊔⊔E often encodes a set of vectors as the set of rows of a matrix; for instance to print the values returned by the function *n_tabl* for each of the partitions of 7, it suffices to enter

for *lambda* **row** *partitions*(7) **do** *print*(*n_tabl*(*lambda*)) **od**

2.5.6. Break and return

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 clause (but not in any loop clause 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.

```
prime(int n) = loc v = [2];
    for i = 3 to n do if prime_test(i) then v+ = i fi od; v
prime_test(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 *prime_test* uses the local variable *v* of *prime*, which is possible according to the (dynamic) binding rules for variables; see Section 2.6.2.

---

* This is true for any ordinary use of **break**, but in fact the rule is a bit more complicated, since ⊔⊔E completes the evaluation of any statement that is being evaluated as part of the loop which is being terminated (but of course a *series* is completed when its current statement is). 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. Therefore unusual effects only happen if the **break** statement is contained in a clause which is being used as a proper subexpression of some statement within the loop. For instance in '*a* = {**break** 5}' the value 5 is assigned to the variable *a*, instead of forming the result of the loop.

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

return or **return** ⟨ expression ⟩

In the same fashion as for **break**, the expression after **return** will determine the result of the function. The function *prime_test* in the previous example could therefore also have been written as

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

2.5.7. Setdefault

The statement **setdefault** 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 computations 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: by convention this group is passed as the final argument. Since one often does a number of computations for the same group, one may however define for convenience a default group. When this has been done it is allowed to omit the final argument specifying the group: the default group will implicitly be assumed. To set the default group, execute **setdefault** ⟨ expression ⟩ with ⟨ expression ⟩ yielding the desired group; to find out what the current default group is, execute **setdefault** without parameters, which will print the default group. Either form of the **setdefault** command returns void. As an example, the commands

setdefault A_3; *W_orbit*$([1,1,1])$

will produce the same result as *W_orbit*$([1,1,1], A_3)$, but it will also have set the default group to A_3, so that this group 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 greater detail.

Functions can only be defined on top level, i.e., not within other 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 symbols used (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; the real execution starts only after this

has been successfully done. Before the function is invoked, all of its arguments are computed; thereafter the function itself is executed.

Similarly to operators, there can be more than one meaning attached to one same function identifier, as long as these meanings 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 (although we have tried to include all types that could meaningfully be associated with an operation among the predefined functions). 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:

⟨ name ⟩ = (⟨ type ⟩⟨ variables ⟩; ... ; ⟨ type ⟩⟨ variables ⟩) = ⟨ series ⟩
⟨ name ⟩ = (⟨ type ⟩⟨ variables ⟩; ... ; ⟨ type ⟩⟨ variables ⟩) { ⟨ series ⟩ }

where ⟨ type ⟩ stands for one of the keywords **int**, **vec**, **mat**, **pol**, **grp** and **tex**, and ⟨ variables ⟩ stands for one or more identifiers, separated by commas. The identifiers denote the parameters of the function, in order; each identifier in ⟨ variables ⟩ has the type specified by the preceding ⟨ type ⟩. (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. Note however that the ⟨ series ⟩ in the first form may well be a single block, in which case it very much resembles the second form; this form has been used frequently in the examples, where the '=' serves for command prolongation to the second line.)

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:

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

Now the call $f(3)$ yields 6, while $f($`"7^51="`$, 7, 51,$ `" (44 digits)."`$)$ prints

```
7^51=1258925529853188502634196238398754544475 8745 (44 digits).
```

and yields no value, $gcd(51592783, 2373146567)$ yields 1991, 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 finds integers k, l satisfying $d = kx + ly$, by means of the so-called "extended Euclidean algorithm". The result is encoded as a vector $[d, k, l]$.

$ext_gcd(\textbf{int}\ x, y) =$
{ **loc** $m = [[x, 1, 0], [y, 0, 1]]$;
invariant: $m[i, 1] = x * m[i, 2] + y * m[i, 3]$ for $i \in \{1, 2\}$
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, blocks and function parameters. We now discuss them in greater detail.

During execution, ULE 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 (the reason we mention 'series' rather then 'clauses' is that each execution of a loop body is considered separately). Here is a complete list of the series that correspond to separate levels:

- A series enclosed in curly braces '{' and '}', forming a block,
- A series between **then** and **else** (or **fi**) or between **else** and **fi**,
- A 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 ULE 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. The following example illustrates these rules:

$a = 3$; **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.

Whenever a variable is used in an expression, or occurs as left hand side of an assignment without **loc**, a check is made whether there exists a variable of that name at the current or any higher level (in that order), until eventually it is checked whether 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 was being assigned to a new variable is created at the current level (as if the assignment were preceded by **loc**), but if the value of the variable was needed, 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. Examples can even be constructed in which during the execution of a single command one same identifier denotes different variables (possibly even of different types). To make the validity of a function independent of the environment it is called from, the use of **loc** is always recommended for variables storing intermediate results within functions.

Although the call-by-value rule excludes the possibility of a function modifying by assignment to its own parameters any values in another function calling it, it *can* modify the local variables (and even the parameters) of such a calling function by means of direct assignments to them (provided these assignments are not shielded by any **loc**). Unrestricted use of this feature may lead to ugly programs, but it may be used in a special way to alleviate the restriction that functions cannot be defined within others: when an auxiliary function is *only* called by a specific other function, it can access the variables of the calling functions as if it were defined local to that function at the point of the call. The technique of using non-local variables to imitate a local function definition is exemplified in several places in this manual (in most cases it is explicitly indicated), for instance in the function *prime_test* of Section 2.5.6 above.

2.6.3. Make and apply

To LiE, functions differ from values in the sense that they cannot 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 arguments, and that yield values computed using this function.

The function that appears as an argument to **make** or **apply** should be user defined. It is treated as a mathematical function, so it should not have side effects (i.e., external changes resulting from calling the function, other than the value yielded), because 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. 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. Similarly, we have

$$\mathbf{make}(f, m) = [[f(m[1,1]), \ldots, f(m[1,c])], \ldots, [f(m[r,1]), \ldots, f(m[r,c])]]$$

where r and c are the numbers of rows and columns of m. In other words $\mathbf{make}(f, m)$ is a matrix m' of the same size as m, with $m'[i,j] = f(m[i,j])$ for all i, j.

Similar operations are available for functions of two integer arguments. For $f\colon (\mathbf{int}, \mathbf{int}) \to \mathbf{int}$ 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. By way of example, with the function *gcd* of Section 2.6.1 we have

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

Again there are variants to present arbitrary sample data to f, namely by providing a pair of vectors or matrices of the same size, where the first argument to f is taken from the first, and the second argument from the second vector respectively 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. Thus, for example $\mathbf{make}(gcd, [3,5,8,21,91], [8,10,12,14,39])$ yields $[1,5,4,7,13]$. Similarly we have

$$\mathbf{make}(f, m, m') = [[f(m[1,1], m'[1,1]), \ldots], \ldots, [\ldots, f(m[r,c], m'[r,c])]]$$

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 UE, however:

$\mathbf{iapply}(f, n, n')$	$= f^n(n')$	where $f\colon \mathbf{int} \to \mathbf{int}$
$\mathbf{vapply}(f, n, v)$	$= f^n(v)$	where $f\colon \mathbf{vec} \to \mathbf{vec}$
$\mathbf{mapply}(f, n, m)$	$= f^n(m)$	where $f\colon \mathbf{mat} \to \mathbf{mat}$

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

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

in other words, $\mathbf{vapply}(f,n,n')$ 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 $\mathbf{vapply}(f,4,3) = [3,6,12,24,48]$. A final variant of **vapply** uses a function $f\colon \mathbf{vec} \to \mathbf{int}$ to incrementally build up a vector; it can be formulated in terms of the first instance of **vapply**:

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

As indicated, 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:

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

With this function we compute the first 12 Fibonacci numbers in the sequence starting with $[1,1]$ by calling $\mathbf{vapply}(f,10,[1,1]) = [1,1,2,3,5,8,13,21,34,55,89,144]$. 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 some additional control over the way LiE operates. Unless noted otherwise these commands are not statements, implying that they 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 ⟩'. (For very long computations it may be advisable however not to start an interactive session at all, but to use input/output redirection as provided by the operating system, and moreover to run the job in the background. The behaviour of LiE is the same when its input is redirected from a file as when that file is processed via the `read` command, except that in the latter case control will return to the terminal upon completion.)

A file can be edited during the LiE session by giving the command '`edit` ⟨ name ⟩'; upon completion of the editing session 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` command of a LiE session. In a UNIX environment the shell variable `$PAGER` is also used: output of large amounts of text, such as produced for intance by `listfuns`, `help` and `learn`, are processed via the progam indicated by `$PAGER` (the default is to use the program `more`). This feature can be used for instance in a window environment to display help texts etc. in separate windows.

To save the variables and user defined functions of a particular session, execute the command '`write` ⟨ name ⟩'. As a result, these functions are written in the file ⟨ name ⟩, in a format that can be read back by `read`. 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 '`?`'). 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 ⟩'. Because there are so many functions in the system, a directory structure has been imposed on the set of help entries. Each call of '`?` ⟨ topic ⟩' will place you into the directory of that topic, and the name of the directory will be printed as part of the prompt which immediately follows the help information. The directory you are in has no effect on searching a topic (it will be found if it is present anywhere in the directory structure), but the command '`?index`' will show just the entries of the current help directory. In this way it is possible to find out which functions or commands are most closely related to the previous one. The command '`?functions`' lists the full set of predefined functions, regardless of the current help directory.

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 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 little 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, and also between printing a result and prompting for a new command, it is also possible to explicitly call the garbage collector by the function *gcol*. It is also possible—by stating `off gc`—to (temporarily) inhibit garbage collection when one knows that there will be no memory to free anyway (see Section 2.7.4).

Inhibiting garbage collection may reduce the time needed to read in long files containing only function and variable definitions (if enabled the garbage collector is called after each definition). On the other hand it may in extreme cases be useful to call *gcol* immediately before calling a built-in function that uses very many objects, since the garbage collector is never invoked during the execution of a built-in function. The memory management monitors the *number* of objects in use by LiE, not their sizes, so if very large objects are being used the physical memory of the computer may become depleted before LiE notices any problems, thereby causing a fatal error and termination of the LiE session. See the command `maxobjects` below for dealing with such situations. To monitor the amount of memory occupied, the function *used* provides the number of objects currently in use (but possibly inaccessible). 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 that 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 the following table:

feature	default state	effect of non-default setting
`bigints`	`on`	'`off bigints`' changes the binding of integer arithmetic operators so that machine size integers will be used instead of arbitrary length integers; this is faster, but overflow will not be detected. Bigints produced in other ways, e.g., by polynomial arithmetic, can still be used, but conversion failure may occur when integer operations are applied to them.
`lprint`	`on`	'`off lprint`' prints vectors, matrices and polynomials in rectangular 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 replaced by vertical bars. For polynomials however the difference is significant: the terms are listed in a tabular form: the coefficients are printed in the leftmost column, and are followed by an asterisk; the entries of the exponents are printed in the columns to the right of the asterisk. The filename used for '`on monitor`' can be set differently by the command '`monfil` ⟨ filename ⟩'.

The system parameter that determines the sorting criterion used for ordering of

terms in a polynomial (and of the rows in a matrix in calls of *sort* and *unique*) has six 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
`on + height`	select increasing height ordering
`on - height`	select decreasing height ordering

The current values of all the system parameters can be obtained by giving the '`on`' or '`off`' command without parameters. In order to limit the effect of the commands `on` and `off` to the execution of the function in which they are needed, the command `savestate` is provided to save the state of all the parameters, whereupon it can be restored some later time by `restorestate`. The default group is also saved and restored by these commands. Pairs of `savestate`/`restorestate` commands may be nested. All commands mentioned up to here in this subsection are statements, and may correspondingly be used not only from top level, but also in function definitions, loops, etc.

There are two more system parameters, which determine the amount of memory that LiE allocates for representing programs and data. They take a numeric argument, as follows:

`maxnodes` n	set maximum number of nodes (for programs) to n
`maxobjects` n	set maximum number of objects to n

The numeric argument must be an explicit number; it may not be an expression. If the argument is omitted, the current setting if the parameter will be printed.

Increasing the number of nodes may be needed if heavily recursive functions are being used, but this rarely occurs. It is more common that the number of objects has to be modified. An object is any integer, vector, etc. in use by LiE, although some of them might be shared if they have equal values. Note however that each polynomial coefficient counts as an individual object (and the same is true for simple components of composite groups, but this is less likely to be of any importance). When the available number of nodes or objects runs out, an error message is printed and the computation aborted; in such cases one may increase the number of nodes or objects and restart the computation. In case of object table to overflow it is advisable to check the default group and the sorting criterion before proceeding, because some functions (most notably *branch*) temporarily change these, and aborting the computation may have given them no opportunity to restore the original values.

Setting `maxobjects` too low obviously may render certain computations impossible, but on the other hand setting `maxobjects` too high may mean that the garbage collector is not called in time, leading to exhaustion of physical memory (also the time needed for each garbage collection increases with `maxobjects`). As in instructive example, the authors once used LiE to solve a large system of linear equations over a finite field; this involved reading in a large matrix and then running a Gaus-

sian elimination program (the program is given in Section 5.1.8). During the input phase, `maxobjects` had to be set to a high value, because all the entries are converted to integer objects before being combined to a matrix. During the elimination phase however there were very few objects (typically one large matrix and a few vector and integer quantities), but their average size was considerable. Therefore the value of `maxobjects` had to be drastically decreased during this phase in order to allow the garbage collector to come into action at regular times, thus preventing memory from being swamped.

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). It is important to know the correspondences between the mathematical notions and the concrete objects manipulated by LiE; it is the purpose of the current chapter to explain them. To this end a large part of this chapter consists of a listing of the names of mathematical notions representable in LiE, with an indication of how they are 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 understand 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, treating the following subjects: Lie groups and algebras, roots and weights, Weyl groups and their action, matters related to the symmetric groups, and representations of Lie groups. 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 explanations (if you are not sure under which subject a term is classified, the index in Chapter 8 gives references to all terms).

It is advisable to skip certain parts of this chapter on first reading. The novice to the subject might want to read the main text of each of the subsections but skip the alphabetic listings, while the expert may wish to skip this chapter completely and turn to the alphabetical listing only to look up the representation of certain concepts. As this chapter is supposed to allow more or less *random access*, some repetitiveness has been unavoidable. We start by listing the different ways in which several types of LiE objects may be interpreted in general.

Matrix A matrix can either be interpreted as a linear transformation (acting by right-multiplication on row vectors), or as 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 a character table. For instance, a matrix representing a set of roots will be termed a *root matrix*. See also *orbit matrix*, and *restriction matrix*.

Polynomial A polynomial may either just represent itself, i.e., a Laurent polynomial (for instance in the case of the Kazhdan-Lusztig polynomials, which incidentally are always ordinary polynomials), or it may encode a set of vectors of equal size, with multiplicities. In the latter case each term represents the occurrence of its exponent in the indicated set (such exponents are always interpreted as weights), occurring with multiplicity equal to the coefficient of the term. On their turn, sets of weights with multiplicities may have different interpretations, leading to a further distinction between polynomials. Three important such interpretations are that of *decomposition polynomials*, *character polynomials*, and *dominant character polynomials*.

Vector A vector may represent an element of a vector space (or rather of a free **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 considering 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 contains 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). A complex torus (i.e., a direct product of copies of $\mathbf{C}^*$) is another example of a connected reductive but not semisimple group (in fact semisimple groups and tori are opposite extremes within the class of reductive groups). The chosen class of groups 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. [Bour7]), each such Lie group g is the homomorphic image of a direct product of a simply connected semisimple complex group and a complex torus, where the homomorphism has a finite kernel, which is contained in the center. The semisimple factor in the product may be reconstructed up to isomorphism as the universal cover of the commutator subgroup of g, and the torus factor as the identity component of the center of g. 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 Special Orthogonal group: the group of all matrices $m \in SL(n, \mathbf{C})$ 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 so-called exceptional groups. The classical groups are given a symbolic identification, called their type, of the form A_n, B_n, C_n, or D_n, where n can be an arbitrarily large integer; the exceptional groups have types E_6, E_7, E_8, F_4, and G_2. For classical groups the assignment of types is as follows:

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

Although formally LiE deals with complex rather than with real Lie groups, the representation theoretic computations may be interpreted for the compact real forms of the complex groups as well, since their (finite dimensional complex) representation theories are equivalent. The compact real Lie groups of classical type correspond to the indications A_n–D_n as follows:

$$A_n\colon SU(n+1, \mathbf{C}) \quad B_n\colon Spin(2n+1, \mathbf{R}) \quad C_n\colon U(n, \mathbf{H}) \quad D_n\colon Spin(2n, \mathbf{R})$$

where $\mathbf{H}$ stands for the skew field of the quaternions. In addition we mention the following special cases in low dimensions, which use the fact that certain "degenerate" classical types go by different names (e.g., B_1 is in fact A_1 and D_2 is A_1A_1). The group $U(1, \mathbf{C})$ (and $SO(2, \mathbf{R})$ which is identical) is represented as the torus T_1 (more generally $U(n, \mathbf{C})$ is represented by $A_{n-1}T_1$, just like the complex group $GL(n, \mathbf{C})$); $SO(3, \mathbf{R})$ is indicated by A_1 (which in fact represents its twofold cover $SU(2, \mathbf{C})$); $SO(4, \mathbf{R})$ is similarly indicated by A_1A_1, $SO(5, \mathbf{R})$ by B_2 or C_2, and $SO(6, \mathbf{R})$ by D_3 or A_3.

The groups directly representable in LiE are those complex Lie groups which, after dividing out their central torus, are simply connected; this means that they are a direct product of simply connected simple groups and a central torus, without the need to apply a 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. In some cases the mathematical specification would require that a function returns a group which does not fall into the mentioned class which is representable in LiE (e.g., this may be the case for the function *centr*). In such cases the group of which it is a central quotient with finite kernel, and which *is* simply connected modulo its central torus, is returned instead. The information that is lost in this way is the description of the finite kernel. For example, the group $g = GL(2, \mathbf{C})$ modulo its central torus (the 1-dimensional group of multiples of the identity matrix) is isomorphic to the projective group $PSL(2, \mathbf{C})$ which is not simply connected (it is covered twofold by $SL(2, \mathbf{C})$). Therefore g is represented in LiE by the group of type A_1T_1, which in fact describes the direct product $\hat{g} = SL(2, \mathbf{C}) \times \mathbf{C}^*$. In this case the canonical surjective morphism $\hat{g} \to g$ has kernel $\{(\mathbf{1}, 1), (-\mathbf{1}, -1)\} \subset A_1T_1$, where $\mathbf{1} \in A_1$ stands for the identity in SL_2. LiE does provide a representation for such *toral elements*; in the current case the non-trivial element of the kernel would

be represented as $[1,1,2]$ (see Section 3.2 for details). We shall assume from now on that g 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 study of the *Lie algebra* it induces on the tangent space to the group at the identity element. In particular, any finite dimensional representation of one of the two leads to a similar representation of the other. Indeed much of the theory of Lie groups is derived by studying the adjoint representation, which is the representation of the Lie group on its Lie algebra, acting 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 The center of any reductive Lie group g is the direct product of a torus and a finite group; the former (which is clearly the connected component of the center) is called the central torus of g. For the groups LiE deals with this central torus is even a direct factor of g itself, the other factor being the semisimple part of g.

Diagram The (Dynkin) diagram of a semisimple Lie group is a graph indicating the isomorphism 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 [Bour7]. 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 $\mathbf{C}$). This group is a Lie group, and any Lie group homomorphism of some Lie group g to $GL(V)$ is called a representation of that Lie group on the vector space V. See also *special linear group*.

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 [Hum1]. See also [Jac].

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].

Lie rank The dimension of a maximal torus of g is called the Lie rank of g.

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; *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 simply connected 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, an object of type *group* always refers to a group that itself is a direct product of a simply connected semisimple group and a torus (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. Obviously, each conjugacy class of semisimple elements has representatives in T. Some elements of T, namely those of finite order, can be represented in LiE; see in Section 3.2 under *toral element*.

Semisimple group A connected reductive Lie group is called semisimple if it contains no non-trivial central torus, or equivalently if it is equal to its commutator subgroup. Note that a non-trivial semisimple group necessarily contains non-semisimple elements.

Semisimple rank The semisimple rank of a group g is the Lie rank of its semisimple part, or stated differently, the Lie rank of g minus the dimension of its central torus.

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

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 closed connected 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; as such it decomposes into 1-dimensional representations. The resulting formal sum of weights is called the (formal) character of the representation with respect to T. This formal sum of weights can be represented by a polynomial, which is then called a *character polynomial*.

3.2. Roots and weights

Consider the set $\Lambda(T)$ of algebraic group morphisms $T \to \mathbf{C}^*$ (or equivalently, isomorphism classes of 1-dimensional T-modules); its elements are called *weights*. Weights may be composed by pointwise multiplication of $\mathbf{C}^*$-valued functions, which turns $\Lambda(T)$ into an Abelian group (in terms of 1-dimensional T-modules the operation corresponds to taking the tensor product). We use an additive notation for this group; 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 $\Lambda(S) \oplus \Lambda(T')$ where S and T' are as described in Section 3.1. The subgroup $\Lambda(S)$ is pointwise fixed by W. The group T, being reductive and Abelian, 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. The set of non-zero 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, which is 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 $\mathbf{Z}$. For definiteness we fix the inner product by the following requirements: for all roots $\alpha \in \Phi$ we have $(\alpha, \alpha) \geq 2$, and for the root system of any simple factor of g the value $(\alpha, \alpha) = 2$ occurs for some root. (In other words: the shortest roots are normalised to 2; this normalisation ensures that on the root lattice the inner product takes integral values). We may extend the inner product to $\Lambda(T)$ by letting $\Lambda(S)$ be perpendicular to everything (the extended inner product is non-degenerate only if g is semisimple). The reflections in W acting on $\Lambda(T)$ are precisely the orthogonal reflections in the hyperplanes perpendicular to the roots. A pair of opposite roots gives rise to the same reflection.

Any root α defines a linear form $\langle\,\cdot\,,\alpha\rangle$ on $\Lambda(T)$ by $\langle x,\alpha\rangle = \frac{2(x,\alpha)}{(\alpha,\alpha)}$, which value is independent of the scalars involved in the choice of the inner product, and moreover is always *integral*. The image of a weight x under reflection in the hyperplane perpendicular to a root α is given by $x - \langle x,\alpha\rangle\alpha$; in particular it lies in the same coset of the root lattice as x.

From the root lattice we may extend scalars to the real numbers, and obtain a real vector space $\Lambda(T') \otimes_{\mathbf{Z}} \mathbf{R}$, endowed with a Euclidean inner product, and an action of W preserving this inner product. Calling this Euclidean vector space E, and viewing the root system Φ as a subset of E, we have the following properties.

(1) Φ is finite, spans E, and does not contain 0.
(2) If $\alpha \in \Phi$, then $\mathbf{R}\alpha \cap \Phi = \{\alpha, -\alpha\}$.
(3) Φ is invariant as a set under the action of W.
(4) For every $\alpha, \beta \in \Phi$ we have $\langle\alpha,\beta\rangle \in \mathbf{Z}$.

These properties can be taken as axioms for the abstract concept of a root system in a Euclidean vector space, where W is taken to be the group generated by the orthogonal reflections in the hyperplanes perpendicular to the roots. The above properties characterise the root systems of Lie groups, in the sense that any abstract root system occurs (up to isomorphism) as the root system of a semisimple Lie group.

We choose (and fix) a hyperplane H in E 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 $\{\alpha_1,\ldots,\alpha_s\} \subset \Phi$ 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.

There exist weights $\omega_1,\ldots,\omega_s$ in $\Lambda(T')$ that form a 'dual basis' to the linear forms $\langle\,\cdot\,,\alpha_1\rangle,\ldots,\langle\,\cdot\,,\alpha_s\rangle$, i.e., that satisfy $\langle\omega_i,\alpha_j\rangle = \delta_{i,j}$ 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 $\omega_1,\ldots,\omega_s$ by a basis $\omega_{s+1},\ldots,\omega_r$ of $\Lambda(S)$ to a basis of $\Lambda(T)$, called the basis of *fundamental weights*.

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 *fundamental Lie subgroup* of g whose root system is Ψ.

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 $\alpha_1, \ldots, \alpha_s$, 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 *Weyl group*.

Fundamental root It is often assumed that a subset of the roots has been chosen as the set of fundamental roots, which 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 $i \in \{1, \ldots, r\}$ and $j \in \{1, \ldots, s\}$; these weights are called the fundamental weights. It is this basis on which *weight vectors* 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 of fundamental roots) of the root system, and 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 an algebraic 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 each i with $1 \leq i \leq r$ we have a group homomorphism $z \mapsto \phi(z)^{\omega_i}$ from $\mathbf{C}^*$ to $\mathbf{C}^*$, where ω_i is the i-th fundamental weight (see *weight*). This homomorphism $\mathbf{C}^* \to \mathbf{C}^*$ must be equal to some map $z \mapsto z^{a_i}$ with $a_i \in \mathbf{Z}$. The one parameter subgroup h is now represented by the vector $[a_1, \ldots, a_r, 0]$. 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 *cent_roots*). 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 n for the final zero lies in h (it is $\phi(\zeta_n)$ where $\zeta_n = e^{2\pi \mathbf{i}/n}$). The restriction matrix for h as subgroup of g can be 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 vectors. Root matrices may be used to denote subsystems of the root system 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 element 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 $i = 1, \ldots, r$. Although LiE cannot represent arbitrary complex numbers, it can represent torus elements of finite order, i.e., elements 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 $t^{\omega_i} = e^{2\pi \mathbf{i} a_i/n} = \zeta_n^{a_i}$ for $i = 1, \ldots, r$, where $\zeta_n = e^{2\pi \mathbf{i}/n}$ is a canonical n-th root of unity. It follows form this description that any a_i may be taken modulo n, and that all the entries (including the final n) may be multiplied by a common non-zero factor, without changing the indicated toral element. See Section 5.7 for examples showing how to bring a toral element into a canonical form (using the function *gcd*), and how to transform between this presentation of toral elements and the more usual presentation for classical Lie groups like $GL(n, \mathbf{C})$, namely by the diagonal entries of the diagonal matrix t. See also *one parameter subgroup*.

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 pointwise multiplication of weights

as $\mathbf{C}^*$-valued functions (which corresponds to the tensor product of 1-dimensional T-representations); this is written additively. 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 turns $\Lambda(T)$ 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 $\omega_1, \ldots, \omega_r$, it 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. Elements of W can be represented in LiE either as products of fundamental reflections (see *Weyl word* below) or as $r \times r$ matrices giving their action on $\Lambda(T)$. There are convenient ways of switching from one representation to another. As we have seen the action of the fundamental reflection r_i on $\Lambda(T)$ is given by $xr_i = x - \langle x, \alpha_i\rangle\alpha_i$, 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 $1 \leq i, j \leq s$, the product r_ir_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 $2\pi/m_{i,j}$, where $m_{i,j}$ is the order of r_ir_j (i.e., the least number $m > 0$ such that $(r_ir_j)^m = 1$). Consequently we have $(\alpha_i, \alpha_j) = -\sqrt{(\alpha_i, \alpha_i)(\alpha_j, \alpha_j)}\cos(\pi/m_{i,j})$, which holds also in the case $i = j$, since $m_{ii} = 1$. Then W has the following abstract presentation:

$$W = \langle\, r_1, \ldots, r_s \mid (r_ir_j)^{m_{i,j}} = 1 \,\rangle.$$

This presentation of W in terms of generators and relations shows that W is a *Coxeter group*. The numbers $m_{i,j}$ are easily determined from the Dynkin diagram of g: when nodes i and j are not directly connected we have $m_{i,j} = 2$, if there is a single bond between them then $m_{i,j} = 3$, if there is a double bond then $m_{i,j} = 4$ and if there is a triple bond the $m_{i,j} = 6$.

The Weyl groups of (semisimple) Lie groups form an important subclass of the finite Coxeter groups (in fact the only other irreducible finite Coxeter groups are the dihedral groups Dih_n of order $2n$ for $n \notin \{2, 3, 4, 6\}$, and the groups known as H_3 and H_4). A number of concepts are considered in LiE for Weyl groups, which are in fact defined for general Coxeter groups. For each $w \in W$ there exists a sequence $r_{i_1}, r_{i_2}, \ldots, r_{i_l}$ of fundamental reflections such that $w = r_{i_1} r_{i_2} \cdots r_{i_l}$. Such a sequence is called an expression for w, and the minimal length occurring among all expressions of w is called the *length* of w, denoted $l(w)$. An expression of minimal length for w (of which many different ones may exist) is called a *reduced expression*. An expression for w (whether reduced or not) may be represented in LiE by the vector $[i_1, i_2, \ldots, i_l]$ consisting of the consecutive indices of its fundamental reflections, which is said to be a *Weyl word* for w (for convenience it is also allowed to include zeros as entries of a Weyl word, without affecting the meaning). Each Coxeter group has a parity, i.e., the map $W \to \mathbf{Z}/2$ given by $w \mapsto l(w) \bmod 2$ is a morphism of groups (this follows directly from the presentation of the Coxeter group). In a representation of W in which its generators act as reflections—such as the action of a Weyl group on the weight lattice—the parity of any $w \in W$ gives the determinant of the corresponding linear transformation.

On any Coxeter group an important partial ordering is defined, called the *Bruhat order*, which will be denoted simply by '$\leq$'. The ordering is compatible with the length function on W: whenever $x \leq y$, we must have $l(x) \leq l(y)$. There are many equivalent definitions of the Bruhat order (see [Deodh]), one of which is the following. Define a reflection in W to be any conjugate wr_iw^{-1} of some simple reflection r_i. The Bruhat order is the partial ordering generated by the relations obtained by putting, for any $x \in W$ and any reflection $s \in W$, either $x \leq xs$ or $xs \leq x$, according as $l(x) < l(xs)$ or $l(xs) < l(x)$ (for reasons of parity we cannot have $l(x) = l(xs)$). In fact those cases for which $l(xs) = l(x)+1$ already generate the Bruhat order; in such a case x is called a *Bruhat descendent* of xs. It can be shown that for any $w \in W$ and any reduced expression for w, the condition $x \leq w$ for $x \in W$ is equivalent to the existence of an expression for x obtainable by removing a subset of the reflections in the reduced expression for w.

A concept which is in a sense a refinement of the Bruhat order is that of the *Kazhdan-Lusztig polynomials*. These are polynomials $P_{x,y}$ in one indeterminate with integer coefficients, defined for any $x, y \in W$. We have $P_{x,y} \neq 0$ if and only if $x \leq y$. For a precise definition and interpretation we refer to the defining paper [KaLu], where the indeterminate is called q. An elementary recursion relation defining these polynomials is given below.

Bruhat descendent For any element $w \in W$ the Bruhat descendents are those

elements $x \in W$ for which $x < w$ in the *Bruhat order* and the length $l(x)$ is exactly one less than $l(w)$. For any reduced expression of w, the set of expressions obtained from it by removing a single reflection in all positions where the resulting expression remains reduced, contains exactly one reduced expression for each Bruhat descendent of w. Also, any element $y \in W$ with $y \leq w$ can be obtained starting from w by repeatedly moving from an element to one of its Bruhat descendents.

Bruhat order The Bruhat order is a partial ordering defined on any Coxeter group. It can be determined by the following recursive definition. For the identity element $\mathbf{e} \in W$ we have $x \leq \mathbf{e} \iff x = \mathbf{e}$. For any other element $y \in W$ there is some simple reflection r_i such that $l(yr_i) < l(y)$; then for any $x \in W$ we have $x \leq y \iff x' \leq yr_i$, where x' is the element with the smaller length from $\{x, xr_i\}$. This definition is independent of the choice of r_i. Incidentally, the definition implies that the condition $l(yr_i) < l(y)$ used above is equivalent to $yr_i < y$. It is tempting to omit the function l from similar conditions (as occur for instance in the description of the Kazhdan-Lusztig polynomials), but we have not done this to avoid the suggestion that the full Bruhat order is involved in cases where a simple length comparison suffices.

Canonical Weyl word When representing Weyl group elements by *Weyl words* it is sometimes useful (for instance when testing for equality) to select for each $w \in W$ a particular expression among the many possibilities; this is then called the canonical Weyl word for W. In LiE we choose for this purpose the lexicographically first reduced expression for w.

Coxeter group A Coxeter group is a finitely presented group, where the presentation is determined by its *Coxeter matrix* $m = (m_{i,j})_{1 \leq i,j \leq s}$; the presentation of the Coxeter group is $\langle\, g_1, \ldots, g_s \mid (g_i g_j)^{m_{i,j}} = 1 \,\rangle$. Every Weyl group is a Coxeter group, with Coxeter matrix given by $m_{i,j} = \mathrm{order}(r_i r_j)$.

Coxeter matrix A Coxeter matrix is a symmetric matrix $m = (m_{i,j})_{1 \leq i,j \leq s}$ with positive integer entries, such that $m_{i,j} = 1$ if and only if $i = j$. Such a matrix is used to define a *Coxeter group*.

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. Note that this term refers to a Weyl group element as representative for a coset, not to a Weyl word representing that Weyl group element.

Dominant weight A weight whose inner products with all fundamental roots are non-negative is called dominant. Equivalently, 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 algebra 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, ..., e_r+1. Usually the exponents of g are given in weakly increasing order.

Kazhdan-Lusztig polynomial For any pair x, y of elements of a *Coxeter group* W a polynomial $P_{x,y}$ is defined, called Kazhdan-Lusztig polynomial. The following recursive description may be used to determine $P_{x,y}$. Unless $x \leq y$ in the *Bruhat order* we have $P_{x,y} = 0$; if $x = y$ then $P_{x,y} = 1$ (a constant polynomial). Otherwise let r_i be a simple reflection such that $l(yr_i) < l(y)$. Then if $l(x) < l(xr_i)$ we have $P_{x,y} = P_{xr_i,y}$; otherwise

$$P_{x,y} = P_{xr_i,yr_i} + XP_{x,yr_i} - \sum_{\substack{x \leq z < yr_i \\ l(zr_i) < l(z)}} \mu(z,y) X^{\frac{l(y)-l(z)}{2}} P_{x,z}$$

where $\mu(z,y)$ is the coefficient of $X^{\frac{l(y)-l(z)}{2}-1}$ in $P_{z,y}$ (taken to be 0 if the exponent is not integral). This recursive description can be deduced from equation (2.2.c) of [KaLu].

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.

Longest element In every finite Coxeter group W there is a unique element of maximal length. It is an involution (but in general not a *reflection*), and is called the longest element of 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 any lattice by integral matrices, an orbit may be represented by an orbit matrix, each row of which represents one element of the orbit.

Reduced Weyl word If an element w of the Weyl group is expressed as a product $r_{i_1} \cdots r_{i_m}$ of fundamental reflections, and no product of fewer than m fundamental reflections yields w, then the sequence $[i_1, \ldots, i_m]$ is a reduced Weyl word for w. In general such a reduced Weyl word is not uniquely determined by w, but see *canonical Weyl word*.

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 reflections are precisely the conjugates of the simple reflections, and the latter description makes sense for arbitrary Coxeter groups.

***R*-polynomial** For any pair x, y of elements of a *Coxeter group* W a polynomial $R_{x,y}$ in one indeterminate is defined, called R-polynomial. These polynomials are related to the Kazhdan-Lusztig polynomials (see Section 5.9.1). The following

recursive description may be used to determine $R_{x,y}$. Unless $x \leq y$ in the *Bruhat order* we have $R_{x,y} = 0$; if $x = y$ then $R_{x,y} = 1$ (a constant polynomial). Otherwise let r_i be a simple reflection such that $l(yr_i) < l(y)$. Then if $l(xr_i) < l(x)$ we have $R_{x,y} = R_{xr_i,yr_i}$; otherwise $R_{x,y} = (X-1)R_{x,yr_i} + XR_{xr_i,yr_i}$. For more details about the meaning of these polynomials and their relationship to the Kazhdan-Lusztig polynomials, one may consult [KaLu]; see also Section 5.9.1.

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* $\Lambda(T)$. 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 $r_{i_1} \cdots r_{i_l}$ is such a product, the corresponding Weyl group element may be represented by the so-called Weyl word $[i_1, \ldots, i_l]$. It is allowed to include entries equal to 0 in a Weyl word, which are ignored by LiE; no function that returns Weyl words will include such zeros in the result, except possibly as a padding at the right end when that Weyl word forms a row in a matrix of Weyl words of different lengths.

3.4. The Symmetric groups and related matters

Although they are not (connected) Lie groups, the Symmetric groups enter into a number of computations performed by LiE (in particular into *plethysm*). The representation theory of the General Linear groups is closely linked with that of the Symmetric groups (this goes much further than the observation that the Weyl group of GL_n is isomorphic to the symmetric group S_n). Either of these representation 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; one may consult [JaKe] for details. We will just recall the basic definitions, and explain how they relate to the objects manipulated by LiE.

The Symmetric group on n symbols, denoted S_n, consists of permutations of the set $\{1, 2, \ldots, n\}$; these are readily represented by vectors of length n containing the entries 1, 2, ..., n in permuted order. An important rôle in the theory of S_n is played by *partitions* of n; these are weakly decreasing sequences of natural numbers with sum equal to n. The individual terms of a partition are called its *parts*. Partitions of n parametrise the conjugacy classes of S_n in a natural way: the parts indicate the sizes of the cycles of the elements of the corresponding conjugacy class. Partitions also naturally parametrise the irreducible representations of S_n. For the precise correspondence of partitions to representations we refer to the literature, but we remark that LiE can compute the character χ_λ of the representation corresponding to the partition λ, i.e., the function mapping each permutation to its trace in that irreducible representation (in the case of S_n this is always an integral number). Since

this function is constant on each conjugacy class, it can be specified by giving its value on each such class; the value of χ_λ on the conjugacy class of cycle type μ will be written as $\chi_\lambda(\mu)$.

Partitions also provide an alternative way to represent dominant weights of SL_n or GL_n. Taking as maximal torus T in those groups the set of diagonal matrices, a weight (which is a function $T \to \mathbf{C}^*$) is a monomial expression in the diagonal entries, and is determined by the vector v of the n exponents occurring for these entries. This is not the expression on the basis of fundamental weights which is used throughout in LiE. The group SL_n has type A_{n-1}, and the coefficient of the i-th fundamental weight in the weight represented by v is the difference $v[i] - v[i+1]$ of two successive exponents. Note that a vector v all of whose entries are equal corresponds to the zero weight; this is because the product of all diagonal entries of a diagonal matrix is its determinant, which is identically 1 in SL_n. If GL_n is represented in LiE by a group of type $A_{n-1}T_1$, then the first $n-1$ coefficients on the basis of fundamental weights are determined as in the SL_n case; the final coefficient is equal to the sum of the entries of v. The fact that only one in every n possible weights of $A_{n-1}T_1$ can be so obtained, stems from the fact that GL_n is not itself the group encoded as $A_{n-1}T_1$, but rather a quotient of it by a central subgroup of order n.

The above correspondence applies to arbitrary weights; the dominant weights correspond to vectors v whose entries are weakly decreasing. In this way the dominant weights of SL_n are in bijection with partitions of arbitrary natural numbers with at most $n-1$ (non-zero) parts (the n-th part can be made 0 by a uniform shift of all the parts). For GL_n partitions only correspond to those dominant weights that involve only non-negative powers of the diagonal entries (i.e., the inverse of the determinant, although it is a dominant weight, is not used); such weights are in bijection with partitions with at most n parts. We will say that weights, when represented by vectors v as above, are expressed in *partition coordinates* (even if they should be non-dominant).

Here are some examples of these correspondences. The highest weight of 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 $[1]$, which is represented by $[1, 0, \ldots, 0]$ on the basis of fundamental weights. The partition $[d]$ corresponds to the highest weight of the d-th symmetric power (see below) of the standard module, and is represented by $[d, 0, \ldots, 0]$ on the basis of fundamental weights. The partition $[1, 1, \ldots, 1]$ of d corresponds to the highest weight of the d-th alternating power of the standard module, and is represented by $[0, \ldots, 0, 1, 0, \ldots, 0]$ on the basis of fundamental weights, with the coefficient 1 appearing in the d-th position.

Note that the action of the Weyl group of type A_{n-1} or $A_{n-1}T_1$—which is isomorphic to S_n—corresponds to the action of S_n permuting the n partition coordinates. In the representation theory of SL_n and GL_n partition coordinates are often more natural than coordinates on the basis of fundamental weights; they are for instance used in the Littlewood-Richardson rule.

To explain why the Symmetric group plays a rôle for representations of an arbi-

trary reductive Lie group g, 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 which exchanges the two tensorands. Consequently the two eigenspaces of that involution (viz. the spaces of symmetric and of anti-symmetric tensors) are g-submodules of $V \otimes V$. We call these two submodules the symmetric and alternating tensor square of V; the ordinary tensor square is their direct sum.

More generally we may consider arbitrary symmetric and alternating tensor powers of V, consisting of the (fully) symmetric respectively alternating tensors in $V^{\otimes n} = V \otimes V \otimes \cdots \otimes V$. For $n > 2$ however, the n-th symmetric and alternating tensor powers together do not combine to the full n-th tensor power. 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 so-called isotypical components. The isotypical component for the S_n-representation R_λ corresponding to a partition λ, is isomorphic (as $g \times S_n$-module) to the tensor product of some g-module $V^{(\lambda)}$, say, with that S_n-representation R_λ, in such a way that the group g acts via the tensorand $V^{(\lambda)}$ and S_n via the tensorand R_λ. The module $V^{(\lambda)}$ is then called the *symmetrised tensor power*, or *plethysm*, of V with respect to the partition λ.

Character polynomial 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_λ of the symmetric group may be represented by a character polynomial, which is a polynomial in n indeterminates, in which each exponent represents a partition μ of n (padded with trailing zeros) and its coefficient is the (integral) value $\chi_\lambda(\mu)$ of the character χ_λ 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 dominant weights of SL_n or GL_n.

Partition coordinates A weight x for a group of type A_{n-1} can be expressed in partition coordinates by forming a vector of length n whose i-th entry is the sum of the coefficients in x of the j-th fundamental weights for $j \geq i$ (note that the final entry is always 0). Conversely the coefficient of the i-th fundamental weight can be obtained as the difference between the i-th and the $i+1$-st partition coordinate. In LiE these conversions can be performed by the functions *to_part* and *from_part*. Partition coordinates are used for the function *LR_tensor*.

Robinson-Schensted correspondence The Robinson-Schensted correspondence is an algorithmically defined bijection between the elements of the Symmetric group S_n and the set of pairs of *Young tableaux* of equal shape with n entries. For a definition of the correspondence see [Knuth], pp. 48-69.

Shape The shape of a Young *tableau* is a partition describing the length of the rows of the tableau.

Symmetric group The set of permutations of $\{1, \ldots, n\}$ is called the Symmetric group on n letters, often denoted by S_n. Its conjugacy classes are described by *partitions*, as well as its characters.

Tableau A (Young) tableau is a an arrangement of a set $\{1, 2, \ldots, n\}$ of numbers into rows of weakly decreasing length, such that the numbers increase along rows and columns. The *shape* of a tableau is the sequence of its row lengths, which is a *partition*. A typical example is

1	2	4	6	11
3	5	8		
7	10			
9	12			

,

which has shape $[5, 3, 2, 2]$. In LiE, tableaux are represented linearly by vectors of size n. If t is such a vector, then $t[i]$ indicates the row number of the entry i in the 2-dimensional form. For instance, the tableau above would be encoded as $[1, 1, 2, 1, 2, 1, 3, 2, 4, 3, 1, 4]$. A function *print_tab* is provided to display the 2-dimensional form of a tableau. Young tableaux have many applications in the theory of the symmetric group, for instance the number of tableaux of shape λ is equal to the dimension of the irreducible representation of S_n corresponding to λ.

3.5. 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 view we call V a *g-module*. A g-module V is called *irreducible* if it is non-zero, and has no subspaces stable 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, (or equivalently, 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 of M), while according to the second fact this may be recorded by the set of the highest weights of its constituent irreducible submodules, counted with their multiplicities. Representing this set with multiplicities by a polynomial we obtain the *decomposition polynomial* for the module M.

It is also possible to represent the set of *all* weights occurring in M, i.e., the character of M, by a polynomial, which is then called the *character polynomial* for M. Since W permutes the weights 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 the *dominant character polynomial* for M.

On the set of g-modules a number of operations can be defined, such as formation of Cartesian product, tensor products, tensor powers and (as was discussed in Section 3.4) symmetrised tensor powers. Also, if a Lie group homomorphism $f: h \to g$ 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. Forming Cartesian and tensor products corresponds to addition and multiplication of the character polynomials, and taking tensor powers obviously corresponds to exponentiation of the character polynomials.

The character polynomials of symmetrised tensor powers can also be described in this way, but to explain the general case we would need the theory of symmetric functions (see [Macd]); here we just give the fully symmetric and alternating cases. To this end write out the character polynomial of M in fully expanded form, expressing any multiplicities merely by repetition of the monomial. Then the character polynomial of the alternating tensor square of M is obtained by summation of all products of two distinct monomials of that expanded polynomial; similarly the character polynomial of the n-th alternating tensor power is obtained by summation of all products of n distinct monomials. For the symmetric tensor square of M we take a similar summation, but include into it the square of each of the monomials in the character polynomial of M; for the n-th symmetric tensor power we take the sum of all products of n not necessarily distinct monomials (but unlike the case of polynomial exponentiation, each such product is taken only once). From the theory of symmetric functions it follows that these operations, and indeed the formation of any symmetric function in the monomials of the character polynomial of M, can be expressed (as linear combination with rational coefficients) in terms of polynomial exponentiation together with one additional operation, called the *Adams operator* or formation of power sums. The n-th Adams operator has the effect on the character polynomial of simply replacing each monomial by its n-th power; in general the resulting character polynomial belongs to a *virtual* module. As an example the character of the alternating tensor square of M is obtained from the character χ of M by computing $\frac{1}{2}(\chi^2 - \chi^{*2})$, where the asterisk indicates Adams operator; for the symmetric tensor square we may take $\frac{1}{2}(\chi^2 + \chi^{*2})$.

Branching from the group g to a (reductive) subgroup h amounts to applying to all the exponents occurring in the character polynomial a linear transformation, which describes the transition from weights for the maximal torus of g to those for 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 need

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 description of these operations for character polynomials, it is awkward to have to compute the character of any module to which one would like to apply them, since the character polynomial of a module is usually very much larger than its decomposition polynomial (this is mostly due to the fact that characters must be invariant under the action of the Weyl group W, which is often quite large). Therefore some of the most powerful built-in functions of LiE deal with the computation of these operations on the level of decomposition polynomials. (There can still be a problem due to the size of the characters, since the formulae appropriate for computing the operations for decomposition polynomials often involve some character polynomial as intermediate result; examples are tensor product and branching computations. In such cases LiE will generate and traverse these character polynomial dynamically, rather than computing and storing the whole thing at once.)

Adams operator For each $n > 1$ there is an operator, called the n-th Adams operator, defined on the set of virtual g-modules, which has the effect on the characters of scaling each occurring weight by a factor n (while retaining its multiplicity). In general the result is a virtual module even if the original module was actual. The n-th Adams operator is the 'weight analog' of the operator that, given a character χ of a finite group g, computes the decomposition of the class function $\gamma \mapsto \chi(\gamma^n)$ as an integral linear combination of irreducible characters. The operator is useful for computing *symmetrised tensor powers*.

Adjoint representation Each Lie group g acts on its *Lie algebra* by conjugation, which defines a *representation* of the group, the so-called adjoint representation. The non-zero weights of this representation (all of which occur with multiplicity 1) are called the *roots* of g.

Alternating Weyl sum Let $\rho = \sum_{i=1}^{s} \omega_i = \frac{1}{2}\sum_{\alpha\in\Phi^+} \alpha$, and let W act linearly on the set of polynomials with exponents in $\Lambda(T)$ by putting $(X^\lambda)\cdot w = X^{\lambda\cdot w}$. Then for a given polynomial p the following expression will be of interest (mainly for representation theoretic purposes):

$$\mathcal{J}(p) = \sum_{w\in W} (-1)^{l(w)}((X^\rho p)\cdot w)X^{-\rho}.$$

This expression is called the alternating Weyl sum of p; one interesting property of it is that it gives the same result when applied to a decomposition polynomial as when applied to the corresponding character polynomial. Note that the expression above suggests an alternative action of W on polynomials, where the i-th generator of W (as a Coxeter group) does not act on exponents by reflection in the hyperplane perpendicular to α_i, but rather in that plane shifted by $-\omega_i$ (or equivalently by $-\rho$), and meanwhile also changes the sign of the coefficients. For this action $\mathcal{J}(p)$ is just the sum of the W-images of p. However, since this "shifted alternating action" plays no rôle *except* via the operator $\mathcal{J}$, we will not introduce any further terminology or notation relating to it.

Branching Branching is another word for restricting a g-module M to another reductive group h. Suppose h is a closed reductive Lie subgroup of 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 $\Lambda(T_g) \to \Lambda(T_h)$. In fact we can define such a 'restriction' transformation in the more general setting of an arbitrary Lie group morphism $f: h \to g$ (not just for embeddings); consequently we can consider branching for such situations as well. The matrix m which describes this transformation on the respective bases of fundamental weights, is called the *restriction matrix* for h in g, and plays a crucial rôle in the function *branch*. The function *res_mat* helps to find the restriction matrix in cases where h is a *fundamental Lie subgroup*; LiE has also access to a table of precomputed restriction matrices for cases where h is a maximal subgroup in g but not 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. This character can be conveniently encoded as a *character polynomial*. LiE provides several ways to compute character polynomials, see Chapter 5.

Character polynomial The (formal) character of a representation of g can be expressed as a polynomial, which records each weight λ occurring with multiplicity m in the character as a term mX^λ of the polynomial.

Contragredient representation For each representation of g on a vector space V there is a corresponding representation, called its contragredient representation, on the dual vector space V^*. Here a group element a acts on an element $f: V \to \mathbf{C}$ of V^* by mapping it to $fa: v \mapsto f(va^{-1})$. As an example where the contragredient occurs, consider the space of homogeneous polynomial functions of degree n on V; this is a finite dimensional space on which g acts (by the same formula as above, but with a polynomial function replacing f). This representation of g is isomorphic to the n-th symmetric tensor power of the contragredient representation of the

original representation.

Decomposition polynomial The decomposition of a g-module M into irreducible modules may be represented by a decomposition polynomial d. Each term mX^λ of d represents a dominant weight λ such that the highest weight module V_λ occurs in M with multiplicity m. In certain circumstances we allow m 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.

Demazure operator For each simple root α_i a linear operator M_{α_i}, called Demazure operator, is defined on the set of polynomials with exponents in $\Lambda(T)$. Let X^λ be a monomial and let $\lambda[i] = \langle \lambda, \alpha_i \rangle$ be the coefficient of ω_i in λ. Then

$$M_{\alpha_i}(X^\lambda) = \frac{X^{\lambda \cdot r_i} - X^{\lambda+\alpha_i}}{1 - X^{\alpha_i}} = \begin{cases} X^\lambda + X^{\lambda-\alpha_i} + \cdots + X^{\lambda \cdot r_i} & \text{if } \lambda[i] \geq 0, \\ 0 & \text{if } \lambda[i] = -1, \\ -X^{\lambda+\alpha_i} - \cdots - X^{(\lambda+\alpha_i)\cdot r_i} & \text{if } \lambda[i] \leq -2. \end{cases}$$

Note that $\lambda \cdot r_i = \lambda - \lambda[i]\alpha_i$, and that $M_{\alpha_i}(p)$ is symmetric with respect to the fundamental reflection r_i while the *alternating Weyl sum* is unchanged: $\mathcal{J}(M_{\alpha_i}(p)) = \mathcal{J}(p)$. A discussion of the mathematical significance of this operator is beyond the scope of this manual, as it involves the representation theory of parabolic subgroups, which are not reductive. We refer to [Litt] for more information and references.

Dominant character polynomial Since *character polynomials* are invariant under W, and each W-orbit of weights contains a unique dominant element, the information of a character polynomial can be more compactly represented by omitting all terms whose exponents are not dominant. The polynomial obtained in this way from the character polynomial of a g-module M is called the dominant character polynomial of M. LiE provides functions *filter_dom* and *W_orbit* for going from the character polynomial to the dominant character polynomial and back again.

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 λ 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*; 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 by 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*.

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 (in *partition coordinates*), then the resulting representation of g is called the plethysm, or symmetrised tensor power, of M with respect to λ. See also Section 3.4 for an alternative description of plethysms.

Representation An action by linear transformations of a group g on a finite dimensional vector space V (where the representing matrices depend in a polynomial way on the coordinates of the group element), is called a (rational) 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 of reductive Lie groups, are determined (up to equivalence) 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 parametrised by *partitions*, see Section 3.4.

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. A similar matrix can be defined for an arbitrary Lie group morphism $f: h \to g$. 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*.

Symmetrised tensor power See *plethysm*.

Virtual module A formal sum of irreducible g-modules with integer coefficients corresponds to an actual g-module only if all the coefficients are non-negative (the module can then be constructed by interpreting the formal additions as direct sums). However, when negative coefficients occur, it is still possible to perform all the operations defined for modules (addition, computing characters etc.), although there are no corresponding actual representations. In such cases we say that the objects of computation correspond to virtual g-modules. Computations with virtual modules can be useful even if one is primarily interested in actual modules, as in the computation of *plethysms*.

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 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 described in Section 3.3,
- *permutation*, denoting a permutation as a sequence numbers $\{1, 2, \ldots, n\}$ in permuted order,
- *partition*, denoting a partition as a weakly decreasing sequence of numbers, possibly with trailing zeros, or
- *tableau*, denoting a Young tableau, encoded by row numbers as described in Section 3.4.

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,
- *vectors*, *roots*, *weights*, *Weyl words*, *torals*, *partitions*, or *tableaux*, 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,
- *character*, representing the character of a representation of a symmetric group by its character polynomial,
- *decomposition*, representing a g-module by its decomposition polynomial,
- *dominant*, representing a g-module by its dominant character polynomial,
- *weights*, or representing a set of weights with multiplicities without imposing a fixed interpretation of those weights.

The terms used here are described in more detail in Chapter 3. The notation V_λ will be used throughout 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 toral elements or one parameter subgroups generating the center of g. The center of a semisimple Lie group g (always assumed to be simply connected in LiE) 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 $det_Cartan(g)$ (which order appears in the last column of the result); however, 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). The labeling of the vertices of the Dynkin diagram prescribes the order of the coordinates of root- and weight vectors used in LiE.

dim ((**grp** g)): **int**. Returns the dimension of the Lie group g, which is equal to $dim(adjoint(g), g)$. *Algorithm:* We compute $2 * n_pos_roots(g) + Lie_rank(g)$.

Lie_code (**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 $Lie_group(t, n) = g$.

Lie_group (**int** t, **int** n): **grp**. Returns a torus or a simple group according to the following rule:

$$\begin{array}{llll} Lie_group(0,n) = T_n & & Lie_group(4,n) = D_n & (n \geq 3) \\ Lie_group(1,n) = A_n & (n \geq 1) & Lie_group(5,n) = E_n & (6 \leq n \leq 8) \\ Lie_group(2,n) = B_n & (n \geq 2) & Lie_group(6,4) = F_4 & \\ Lie_group(3,n) = C_n & (n \geq 2) & Lie_group(7,2) = G_2 & \end{array}$$

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.

Lie_rank ((**grp** g)): **int**. Returns the Lie rank of g; for simple groups and tori this equals $Lie_code(g)[2]$, 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 weight vector. The labeling of the fundamental roots is as indicated by *diagram*(g). When g is semisimple, the (i,j)-entry of the Cartan matrix is $\langle \alpha_i, \alpha_j \rangle$. If g contains a central torus, so that the semisimple rank s of g is differs from the Lie rank r, then the Cartan 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 α to be a root, we stick to the root basis for α as well as for β]. See also *inprod* and *norm*.

Cartan_type (**mat** R, (**grp** g)): **grp** [R: *roots*]. Returns the 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 *centr_type*. *Algorithm:* The same algorithm as *fundam* is performed, but only the type of the root system is returned.

cent_roots (**vec** t, (**grp** g)): **mat** [t: *toral*, *result*: *roots*]. Returns the matrix whose rows form the set of all positive roots centralising the toral 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 $T \to \mathbf{C}^*$) maps $t \mapsto 1$. *Algorithm:* Let n be the final entry of t, and t' the vector of remaining entries. First all positive roots are obtained by *pos_roots*, from which those roots α are selected for which $\alpha * \mathit{Cartan}(g) * {t'}^{\top} \equiv 0 \pmod{n}$ (note that the transpose ${t'}^{\top}$ of t' is naturally interpreted as a linear function on the weight lattice with values in $\mathbf{Z}/n$).

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

centr_type (**vec** t, (**grp** g)): **grp** [t: *toral*]. Returns the centraliser $C_g(t)$ of the toral element $t \in T$ (or of the specified one parameter subgroup); effectively only the type is computed. See also *cent_roots*. [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).]

centr_type (**mat** S, (**grp** g)): **grp** [S: *torals*]. Returns the (universal cover of the) centraliser of the toral elements and/or one parameter subgroups of T represented by the rows of S, i.e., the intersection of the groups *centr_type*(t, g) for t traversing the rows of S. *Algorithm:* The set *cent_roots*(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 alternatively be computed as *Cartan_type*(*cent_roots*$(S, g), g)$, which provides a useful check, since in that case the result is obtained by analysing the Cartan matrix of 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 [CoGr].)

closure (**mat** R, (**grp** g)): **mat** [R, *result*: *roots*]. Returns a basis of fundamental roots of the minimal closed subsystem of the root system of g that contains all the roots in R; the basis consisting of positive (for g) roots only is chosen. *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. Afterwards the function *fundam* is applied once more to make the result dominant.

det_Cartan ((**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 *i_Cartan*.

dom_weights (**vec** λ, (**grp** g)): **mat** [λ: *weight*, *result*: *weights*]. Returns the set of dominant weights lying under λ, i.e., the set $\{\, \mu \in \Lambda^+(T) \mid \mu \prec \lambda \,\}$. This is equal to the set of weights occurring in *dom_char*(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 dominant weights would then only be reachable via weights that are not dominant.

fundam (**mat** R, (**grp** g)): **mat** [R, *result*: *roots*]. Returns a basis of fundamental roots of the minimal subsystem of the root system of g that contains all the roots in R; the basis consisting of positive (for g) roots only is chosen. The order in which the fundamental roots are returned is compatible with the standard labeling for a root system of type *Cartan_type*(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 *unique*. This is repeated until no more changes occur.

high_root ((**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 *pos_roots*(g). See also *adjoint*.

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

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 in such a way that for each simple component of g the short roots α have $inprod(\alpha, \alpha) = 2$.

norm (**vec** α, (**grp** g)): **int** [α: *root*]. Returns the norm $inprod(\alpha, \alpha)$ of the root vector α (it would be more accurate, but less convenient, to call this the "squared norm"). When α is a root, the value 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 [Bour4] in the case of groups of type B_n, as the short roots are given norm 1 there.

n_pos_roots ((**grp** g)): **int**. Returns the number of positive roots of the root system of g, which is equal to *n_rows*(*pos_roots*(g)). The number of all roots is twice as much, and can also be computed as $dim(g) - Lie_rank(g)$.

pos_roots ((**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 *n_pos_roots*(g), is *high_root*(g).

4.3. The Weyl group

Bruhat_desc (**vec** w, (**grp** g)): **mat** [w: *Weyl word*, *result*: *Weyl words*]. Returns the set of Bruhat descendents of w, each one represented by a reduced Weyl word. The Weyl word chosen for a Bruhat descendent is the unique one which is obtainable by omitting one of the fundamental reflections occurring in the Weyl word *reduce*(w). *Algorithm:* For each fundamental reflection in w it is tested whether leaving it out decreases the length by exactly 1; if so a row is included in the result. In particular this function does not use *Bruhat_leq*.

Bruhat_desc (**vec** v, w, (**grp** g)): **mat** [v, w: *Weyl word*, *result*: *Weyl words*]. Returns the set of Bruhat descendents w' of w which satisfy $v \leq w'$ in the Bruhat ordering. This is useful in generating all elements between v and w in the Bruhat ordering.

Bruhat_leq (**vec** v, w, (**grp** g)): **int** [v, w: *Weyl word*]. Returns the value 1 if $v \leq w$ in the Bruhat order, and 0 otherwise.

canonical (**vec** w, (**grp** g)): **vec** [v, *result*: *Weyl word*]. Returns the canonical Weyl word representing the same Weyl group element as w, which is the lexicographically first reduced expression for that element. *Algorithm:* We take a strictly dominant weight ρ, and compute $W_word(W_action(\rho, \tilde{w}, g), g)$, where $\tilde{w}$ denotes the reverse Weyl word of w.

canonical (**mat** m, (**grp** g)): **mat** [m, *result*: *Weyl words*]. Returns the matrix obtained by replacing each row w by $canonical(w, g)$, filling out the row with zeros if necessary. This is useful in combination with *unique* when handling sets of Weyl words.

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 matrix obtained by replacing each row of m by the unique dominant weight in its Weyl group orbit.

dominant (**pol** p, (**grp** g)): **pol** [p, *result*: *weights*]. Returns the polynomial obtained by replacing each exponent of p 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.

filter_dom (**mat** m, (**grp** g)): **mat** [m, *result*: *weights*]. Returns the matrix obtained by casting away all rows of m that are not dominant weights.

filter_dom (**pol** p, (**grp** g)): **pol** [p, *result*: *weights*]. Returns the polynomial obtained by casting away all terms of p whose exponents are not dominant weights.

KL_poly (**vec** x, y, (**grp** g)): **pol** [x, y: *Weyl word*, *result*: *polynomial*]. Returns the Kazhdan-Lusztig polynomial $P_{x,y}$. *Algorithm:* The recursion given in Section 3.3 is used, with a few minor improvements.

length (**vec** w, (**grp** g)): **int** [w: *Weyl word*]. Returns the length of the Weyl group element w. We have $length(w) \leq size(w)$, with equality if and only if $w == reduce(w, g)$. *Algorithm:* The computation of $reduce(w, g)$ is simulated, but recording only the length changes.

long_word ((**grp** g)): **vec** [*result*: *Weyl word*]. Returns a Weyl word for longest element of the Weyl group. *Algorithm:* We compute $W_word(-\rho, g)$, where ρ is a strictly dominant weight.

l_reduce (**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 set of fundamental reflections $\{ r_i \mid i \in l \}$. 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.

lr_reduce (**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 sets of fundamental reflections $\{ r_i \mid i \in l \}$ respectively $\{ r_i \mid i \in r \}$. 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 *l_reduce*(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. If any of the matrices has an eigenvalue of absolute value > 1, then the existence of a finite orbit is unlikely; it may happen however that an apparent orbit is still found, due to undetected arithmetic overflow in the vector entries. For larger orbits, see *orbit*(n, v, M), for Weyl group orbits see *W_orbit*.

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. A warning is in its place here: *orbit* uses a rather simple minded algorithm, which in particular allocates space at the beginning for the maximal number of vectors allowed in the orbit; therefore one shouldn't go overboard on choosing the limit n

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 *canonical*, *l_reduce*, *r_reduce* and *lr_reduce*. *Algorithm:* We apply the reflections in the word w from left to right to a strictly dominant weight ρ, 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 the matrix of the reflection of the weight lattice in the hyperplane perpendicular to the root α, expressed on the basis of fundamental weights. See also *W_action*.

R_poly (**vec** $x, y, (\mathbf{grp}\ g)$): **pol** [x, y: *Weyl word*, *result*: *polynomial*]. Returns the value of the R-polynomial $R_{x,y}$. *Algorithm:* The recursion given in Section 3.3 is used.

r_reduce (**vec** $w, r, (\mathbf{grp}\ g)$): **vec** [w, *result*: *Weyl word*, r: *ints*]. The set r determines a subgroup W_r of W generated by the set of fundamental reflections $\{\, r_i \mid i \in r \,\}$. 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. *Algorithm:* The function *l_reduce* is called, reversing the Weyl word before and after the call.

W_action (**vec** $w, (\mathbf{grp}\ g)$): **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*, *W_rt_action*, and *W_word*.

W_action (**vec** λ, **vec** $w, (\mathbf{grp}\ g)$): **vec** [λ, *result*: *weight*, w: *Weyl word*]. Returns the weight that is the image $\lambda{\cdot}w$ of the weight λ underaction of the Weyl group element $w \in W$.

W_action (**mat** m, **vec** $w, (\mathbf{grp}\ g)$): **mat** [m, *result*: *weights*, w: *Weyl word*]. Returns the matrix obtained by replacing each row λ of m by $W_action(\lambda, w, g)$; this matrix is equal to $m * W_action(w, g)$, while conversely $W_action(w, g)$ equals $W_action(id(Lie_rank(g)), w, g)$.

W_action (**pol** p, **vec** $w, (\mathbf{grp}\ g)$): **pol** [p, *result*: *weights*, w: *Weyl word*]. Returns the polynomial obtained by replacing each exponent λ of p by $W_action(\lambda, w, g)$; this polynomial is equal to $p * W_action(w, g)$.

W_orbit (**vec** $\lambda, (\mathbf{grp}\ g)$): **mat** [$\lambda$: *weight*, *result*: *weights*]. 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 *next_perm*. 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*.

W_orbit (**pol** $p, (\mathbf{grp}\ g)$): **pol** [p, *result*: *weights*]. Returns the polynomial obtained by summing over all terms nX^λ of p the polynomial $n * X\ W_orbit(\lambda, g)$; the latter polynomial contains each weight in the W-orbit of λ exactly once and with coefficient n. This operation can be used for instance to compute the full character polynomial of a module from its dominant character module.

W_orbit_size (**vec** $\lambda, (\mathbf{grp}\ g)$): **int** [$\lambda$: *weight*, *result*: *weights*]. Returns the size of the orbit of the weight λ under the Weyl group of g. This size can also be computed as $W_order(g)/W_order(I, g)$, where I is a vector whose entries indicate the positions at which the vector $dominant(\lambda)$ has zero entries.

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

W_order (**vec** I, (**grp** g)): **int** [I: *ints*]. Returns the order of the subgroup W_I of the Weyl group of g generated by the set of fundamental reflections $\{ r_i \mid i \in I \}$. This subgroup is the stabiliser subgroup of any weight vector that has zero entries precisely at positions i for which $i \in I$. *Algorithm:* The set of roots $R = \{ \alpha_i \mid i \in I \}$ is constructed, and $W_order(Cartan_type(R, g), g)$ is computed.

W_rt_action (**vec** w, (**grp** g)): **mat** [w: *Weyl word*, *result*: *lin*(*root*, *root*)]. (Weyl root action) 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.

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

W_rt_action (**mat** m, **vec** w, (**grp** g)): **mat** [m, *result*: *roots*, w: *Weyl word*]. Returns the matrix obtained by replacing each row α of m by $W_rt_action(\alpha, w, g)$; this matrix is equal to $m{*}W_rt_action(w, g)$, while conversely $W_rt_action(w, g)$ equals $W_rt_action(id(Lie_rank(g)), w, g)$.

W_rt_orbit (**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.

W_word (**vec** λ, (**grp** g)): **vec** [λ: *weight*, *result*: *Weyl word*]. Returns a Weyl word for a Weyl group element w whose action sends λ to a dominant weight. In fact, the canonical Weyl word for w is returned, while w is the distinguished representative of its right coset wW_S, where W_S is the stabiliser of $\lambda' = dominant(\lambda, g)$. (Here $S = \{ i \mid \lambda'[i] = 0 \}$, the set of indices of fundamental reflections which stabilise λ'.) *Algorithm:* Starting with the weight λ, and while the weight is not yet dominant, the fundamental reflection r_i is applied for the smallest index i for which the coefficient of ω_i in the weight is negative; the sequence of reflections used is recorded as the result.

W_word (**mat** m, (**grp** g)): **vec** [m: *lin*(*weight*, *weight*), *result*: *Weyl word*]. Returns the canonical Weyl word for the Weyl group element w, if it exists, whose action on the weight lattice is given by the square matrix m, i.e., such that $W_action(w, g) = m$. *Algorithm:* $W_word(\rho * m, g)$ is computed for a strictly dominant weight ρ, and reversed. Then it is checked whether applying *W_action* to the result indeed gives back m.

4.4. Operations related to the Symmetric group

class_ord (**vec** λ): **int** [λ: *partition*]. Returns the order of the conjugation class of S_n of permutations of cycle type λ (for $n = |\lambda|$, the sum of the parts of λ).

from_part (**vec** λ): **vec** [λ: *partition*, *result*: *weight*]. Let n be the number of parts of λ (trailing zeros are significant here) then the function returns the weight for a group of type A_{n-1} (i.e., for SL_n) corresponding to λ, expressed on the basis of fundamental weights. See also *to_part*.

from_part (**mat** m): **mat** [m: *partitions*, *result*: *weights*]. Replaces each row λ of m by *from_part*(λ).

from_part (**pol** p): **pol** [p: *partitions*, *result*: *weights*]. Replaces each exponent λ occurring in p by *from_part*(λ).

next_part (**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*.

next_perm (**vec** p): **vec** [p, *result*: *ints*]. Returns the next permutation of the entries of p, in lexicographical order. If p is the last such permutation, i.e., if the entries of p are decreasing, 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 *next_perm* to go from the weakly increasing order to the weakly decreasing order. See also *sym_orbit*.

next_tabl (**vec** T): **vec** [T, *result*: *tableau*]. Returns the lexicographically next Young tableau of the same shape as T.

n_tabl (**vec** λ): **int** [λ: *partition*]. Returns the number of Young tableaux of shape λ. *Algorithm:* The hook length formula (see [Macd]) is used.

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 *next_part*.

print_tab (**vec** T): **vid** [T: *tableau*]. Displays the Young tableau encoded by T in 2-dimensional form.

RS (**vec** π): **mat** [π: *permutation*, *result*: *tableaux*]. Returns the pair of Young tableaux corresponding to the permutation π by the Robinson-Schensted correspondence; the result is represented as a 2-row matrix.

RS (**vec** P, **vec** Q): **vec** [P, Q: *tableau*, *result*: *permutation*]. Returns the permutation corresponding to the pair of Young tableaux P, Q (which must have the same shape) by the Robinson-Schensted correspondence.

sign_part (**vec** λ): **int** [λ: *partition*]. Returns the sign ($+1$ or -1) of permutations of cycle type λ.

shape (**vec** T): **vec** [T: *tableau*, *result*: *partition*]. Returns the shape of the Young tableau T.

sym_char (**vec** λ): **pol** [λ: *partition*, *result*: *character*]. (Symmetric group character) Let $n = |\lambda|$; the function returns the character polynomial of the character χ_λ of the symmetric group S_n corresponding to the partition λ. *Algorithm:* We use an implementation of the Murnaghan-Nakayama rule (see for instance [JaKe]), representing partitions by their edge sequences for fast detection and manipulation of rim hooks.

sym_char (**vec** λ, **vec** μ): **int** [λ, μ: *partition*]. We should have $|\lambda| = |\mu|$; the function returns the (integral) value $\chi_\lambda(\mu)$ of the character of the symmetric group $S_{|\lambda|}$

corresponding to λ on the conjugacy class with cycle type μ. *Algorithm:* A similar implementation of the Murnaghan-Nakayama rule, slightly adapted to avoid computations that do not contribute to the character value at μ.

sym_orbit (**vec** v): **mat** [*result*: *vectors*]. (Symmetric group orbit) Let $n = size(v)$. 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. The rows of the result are ordered lexicographically. See also *next_perm*.

tableaux (**vec** λ): **mat** [λ: *partition*, *result*: *tableaux*]. Returns matrix whose rows encode the set of all Young tableaux of shape λ, in lexicographic order.

to_part (**vec** v): **vec** [v: *weight*, *result*: *partition*]. Let $n = size(v)$, then v is interpreted as a weight for a group of type A_n (i.e., for SL_{n+1}); the expression of that weight in $n+1$ partition coordinates is returned. When v is dominant, this is a partition with $n+1$ parts. See also *from_part*.

to_part (**mat** m): **mat** [m: *weights*, *result*: *partitions*]. Replaces each row λ of m by *to_part*(λ).

to_part (**pol** p): **pol** [p: *weights*, *result*: *partitions*]. Replaces each exponent λ occurring in p by *to_part*(λ).

trans_part (**vec** λ): **vec** [λ, *result*: *partition*]. Returns the transpose partition of λ.

4.5. Representations

Adams (**int** n, **vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *decomposition*]. Returns the decomposition polynomial of the virtual module obtained by applying the n-th Adams operator to V_λ. *Algorithm:* The computation is effectively equivalent to *v_decomp*(*dom_char*(λ, g) $*$ n, g). This function is used in *plethysm*, *sym_tensor*, and *alt_tensor*.

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

adjoint ((**grp** g)): **pol** [*result*: *decomposition*]. Returns the decomposition polynomial of the adjoint representation of g. For simple groups the adjoint representation is irreducible and the result therefore has a single term; the highest weight of the adjoint representation can then be obtained as *expon*(*adjoint*(g), 1). Since the non-zero weights of the adjoint representation are precisely the roots, this highest weight is equal to *high_root*(g) $*$ *Cartan*(g).

alt_tensor (**int** n, **vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *decomposition*]. (alternating tensor power) Returns the decomposition polynomial of $\bigwedge^n V_\lambda$, the n-th alternating tensor power (also called n-th exterior power) of V_λ. See also *sym_tensor* and *plethysm*.

alt_tensor (**int** n, **pol** p, (**grp** g)): **pol** [p, *result*: *decomposition*]. This is similar to *alt_tensor*(n, λ, g), but with the irreducible module V_λ replaced by the module with decomposition polynomial p.

alt_dom (**pol** p, **vec** w, (**grp** g)): **pol** [p, *result*: *weights*, w: *Weyl word*]. (alternating dominant) Starting with the polynomial p, the following operation is repeatedly applied, taking for i the successive entries of the Weyl word w, reading from left to right. For any term nX^λ let $\lambda[i] = \langle \lambda, \alpha_i \rangle$ be its coefficient of ω_i; the term is unaltered if $\lambda[i] \geq 0$, it is removed if $\lambda[i] = -1$, and it is replaced by $-nX^{(\lambda+\omega_i)\cdot r_i - \omega_i}$ if $\lambda[i] \leq -2$. (The exponent of the latter monomial could also have been written as $\lambda \cdot r_i - \alpha_i$ or as $\lambda - (\lambda[i]+1)\alpha_i$.) As a result of the operation for i, the coefficient $\lambda[i]$ is made non-negative without affecting the image $M_{\alpha_i}(p)$ under the Demazure operator, and hence also without changing the value of the alternating Weyl sum $\mathcal{J}(p)$. The final result of *alt_dom* should be the same when taking for w different *reduced* Weyl words for the same element of W.

alt_dom (**vec** λ, **vec** w, (**grp** g)): **pol** [λ: *weight*, w: *Weyl word*, *result*: *weights*]. Returns $alt_dom(1X^\lambda, w, g)$.

alt_dom (**pol** p, (**grp** g)): **pol** [p, *result*: *weights*]. This is equivalent to (but somewhat faster than) $alt_dom(p, long_word(g), g)$. The resulting polynomial p' can be charaterised as the unique polynomial with only dominant exponents which has $\mathcal{J}(p') = \mathcal{J}(p)$. If p is a character polynomial, then p' is the corresponding decomposition polynomial.

alt_dom (**vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *weights*]. Returns $alt_dom(1X^\lambda, g)$.

alt_W_sum (**pol** p, (**grp** g)): **pol** [p, *result*: *weights*]. (alternating Weyl sum) Returns the alternating Weyl sum $\mathcal{J}(p)$ of p. This function can be useful for demonstration purposes, e.g., to check Weyl's character formula (see Chapter 5), but the fact that the number of terms of the result is a multiple of $W_order(g)$ makes it impractical for groups like E_8. *Algorithm:* First *alt_dom* is applied, then the alternating orbits of the terms are generated in a straightforward way.

alt_W_sum (**vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *weights*]. Is $alt_W_sum(1X^\lambda, g)$.

branch (**vec** λ, **grp** h, **mat** m, (**grp** g)): **pol** [λ: *weight*, m: *lin*(*weight*, *weight*), *result*: *decomposition*]. Returns the decomposition polynomial 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 the weight $\lambda' * m$ (expressed on the basis of fundamental weights for h). In many cases the restriction matrix can be obtained by use of *res_mat*. See also *decomp* for a warning about memory overflow. *Algorithm:* For simple groups g, the computation is a dynamic version of

$$decomp(filter_dom(\,W_orbit(dom_char(\lambda, g), g) * m, h), h).$$

The dynamic nature lies in the fact that no intermediate result between the application of *W_orbit* and *filter_dom* is generated in memory. The function *branch* is a very general one, and in specific situations there may be alternative methods which compute the answer much more efficiently; some examples are given in Chapter 5. Note that the call of *decomp* might result in an error due to "failure of non-virtual

decomposition" being reported, this is almost always due to supplying a matrix m which is not the correct restriction matrix for any morphism $h \to g$ (but we do not know any certain way of testing whether an arbitrary matrix m is a valid restriction matrix or not). When g is composite, then λ and m are split up accordingly, and the results for the individual factors of g are combined by calling *tensor* (for the subgroup h). If g has a central torus, then m should be such that any coefficients of λ on this torus can only contribute on the central torus of h; for this part of the weight *branch* reduces simply to multiplication by m.

branch (**pol** p, **grp** h, **mat** m, (**grp** g)): **pol** [p, *result*: *decomposition*, m: *lin*(*weight*, *weight*)]. This is like *branch*(λ, h, m, g), but with the irreducible module V_λ replaced by the module with decomposition polynomial p. *Algorithm:* Rather than applying the formula given above to p in place of λ, we simply call *branch*(λ, h, m, g) for every exponent λ of p and combine the results linearly. This means on one hand that the call of *tensor* mentioned in the description of that instance of *branch* can still be used to avoid generating complete Weyl orbits for composite groups; on the other hand it means that *branch* can handle virtual modules, despite the call of *decomp*. The only restriction is that each irreducible constituent of p—regardless of the sign of its multiplicity—should branch to an actual h-module; this is always the case for a proper restriction matrix m.

collect (**pol** p, **grp** h, **mat** l, (**grp** g)): **pol** [p, *result*: *decomposition*, l: *lin*(*weight*, *weight*)]. This function attempts to perform the inverse operation of *branch*, namely to reconstruct a g-module from its restriction to h. This is not generally possible unless the restriction matrix is invertible, and in particular g and h have the same Lie rank. When a restriction matrix m has an inverse l, and the h-module with decomposition polynomial p is equal to some restriction *branch*(q, h, m, g) of a g-module via m, then the decomposition polynomial q can be computed as *collect*(p, h, l, g). *Algorithm:* Remarkably, the computation is quite similar to that of branch: we compute *decomp*(*filter_dom*(*dom_char*$(p, h) * l, g), g)$; the fact that the operation is *inverse* to *branch* is mainly due to the inverse of the restriction matrix being supplied in the call. Apart from the switching of rôles of g and h, the main differences with *branch* are that no W-orbits are generated (since it is assumed that dominant weights for g correspond *a fortiori* to dominant weights of h), and that all irreducible constituents of p are considered together (because individually they would almost certainly give rise to problems with the non-virtual decomposition). It can be seen from this that if p does not correspond to the restriction of any g-module, this will result in failing of *decomp*, and also that contrary to *branch*, it is not possible to *collect* virtual modules.

collect (**pol** p, **grp** h, **mat** l, **int** n, (**grp** g)): **pol** [p, *result*: *decomposition*, l: *lin*(*weight*, *weight*)]. An obvious limitation of the previous version of *collect* is that it is only applicable for restriction matrices which are invertible over the integers; certain restriction matrices are invertible, but only over the rational numbers. For these cases this extended version is provided. Since LiE cannot handle matrices with rational entries, a common denominator n of all the entries of the inverse restriction

matrix has to be factored out and passed as a separate argument, so that the scaled inverse matrix l has only integer coefficients. For all weights to which l is applied the image should be divisible by n, or else an error will be reported; apart from this, the extended version of *collect* operates in the same way as the previous one.

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 is $dominant(-\lambda, g)$.

contragr (**pol** p, (**grp** g)): **vec** [p, *result*: *decomposition*]. Returns the decomposition polynomial of the contragredient representation of the module with decomposition polynomial p.

decomp (**pol** d, (**grp** g)): **pol** [d: *dominant*, *result*: *decomposition*]. Returns the decomposition polynomial of the g-module with dominant character polynomial d; it is the inverse operation of *dom_char*. *Algorithm:* The terms of d are sorted according to the decreasing height ordering, then repeatedly *dom_char* is applied to the leading term and the result subtracted from the polynomial, until no terms remain. The sequence of leading terms encountered in this way constitute the required decomposition polynomial. It is required that the coefficients of these leading terms are all positive, otherwise it is reported that the "non-virtual decomposition failed". Note that this function temporarily sets the sorting criterion to decreasing height order, and it may need to change the default group as well to achieve this; this is particularly likely when it is called from *branch*. If memory overflows within *decomp* or *branch*, then then warning in Section 2.7.4 is particularly relevant, and one should check the default group and the sorting criterion. See also *v_decomp*.

Demazure (**pol** p, **vec** w, (**grp** g)): **pol** [p, *result*: *weights*, w: *Weyl word*]. Starting with the polynomial p, repeatedly apply the Demazure operator M_{α_i}, taking for i the successive entries of the Weyl word w, reading from left to right. The final result of *Demazure* should be the same when taking for w different *reduced* Weyl words for the same element of W.

Demazure (**vec** λ, **vec** w, (**grp** g)): **pol** [λ: *weight*, w: *Weyl word*, *result*: *weights*]. Returns $Demazure(1X^\lambda, w, g)$.

Demazure (**pol** p, (**grp** g)): **pol** [p, *result*: *weights*]. This is an abbreviation for the call $Demazure(p, long_word(g), g)$. The resulting polynomial p' can be characterised as the unique W-invariant polynomial which has $\mathcal{J}(p') = \mathcal{J}(p)$. In fact, due to Demazure's character formula, p' is the character polynomial of the module with decomposition polynomial p (provided all exponents of p were dominant). This is not the most efficient way to compute characters, but it can be very useful in checking other algorithms, since nothing but the most elementary manipulations are involved, in which no information about g is used other than its Cartan matrix.

Demazure (**vec** λ, (**grp** g)): **pol** [λ: *weight*, w: *Weyl word*, *result*: *weights*]. Returns $Demazure(1X^\lambda, g)$.

dim (**vec** λ, (**grp** g)): **int** [λ: *weight*]. Returns the dimension of the representation V_λ. *Algorithm:* Weyl's dimension formula is applied.

dim (**pol** p, (**grp** g)): **int** [p: *decomposition*]. Returns the dimension of the g-module with decomposition polynomial p.

dom_char (**vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *dominant*]. (dominant character) Returns the polynomial representing the dominant part of the character of the g-module V_λ. *Algorithm:* Freudenthal's multiplicity formula is applied, see [Hum1] and [Kruse].

dom_char (**vec** λ, μ, (**grp** g)): **int** [λ, μ: *weight*]. Returns the coefficient of X^μ in the character polynomial of V_λ. The weight λ should be dominant, but μ may be any weight. *Algorithm:* The computation of $dom_char(\lambda, g)$ is performed, but it is terminated as soon as the coefficient of $dominant(\mu, g)$ is known.

dom_char (**pol** p, (**grp** g)): **pol** [p: *decomposition*, *result*: *dominant*]. This is like $dom_char(\lambda, g)$, but with the irreducible module V_λ replaced by the module with decomposition polynomial p.

dom_char (**pol** p, **vec** μ(**grp** g)): **pol** [p: *decomposition*, μ: *weight*, *result*: *dominant*]. Returns the coefficient of X^μ in the character polynomial of the module with decomposition polynomial p.

LR_tensor (**vec** λ, **vec** μ): **pol** [λ, μ: *partition result*: *decomposition*]. (Littlewood-Richardson tensor) The partitions λ and μ, which must have the same number of parts, say n, are interpreted as dominant weights for the group SL_n of type A_{n-1}, expressed in partition coordinates. The decomposition polynomial of the tensor product of the corresponding highest weight modules is computed using the Littlewood-Richardson rule, where the exponents in the result are again expressed in partition coordinates. Note that extending λ and μ by zeros can be significant: partitions with more than n non-zero parts may appear as exponents of new terms, while existing terms will reappear in zero-extended form. The total number of non-zero parts is bounded however by the number in λ and μ taken together, so eventually the number of terms will stabilise; the limiting case corresponds to the decomposition of the Young product of the representations corresponding to λ and μ in the representation theory of the symmetric groups. *Algorithm:* A description of the Littlewood-Richardson rule can be found in the literature, see for instance [JaKe] and [Macd]. Suffice it here to mention that the result is computed by a combinatorial counting process, which produces the irreducible constituents of the result one by one, in approximately constant time. This means on one hand that there are no cancellations, as are possible in Klymik's formula which is used for the ordinary *tensor* operation, but on the other hand no advantage is taken of any high multiplicities that may occur in the characters of V_λ and V_μ. Experimental evidence suggests that *LR_tensor* is slightly less efficient than *tensor* when n is small but λ and μ are high weights, but it can be much more efficient when n (and hence the Weyl group) is large, and the weights are relatively small.

LR_tensor (**pol** p, **pol** q): **pol** [p, q, *result*: *decomposition*]. Returns the decomposition polynomial of the tensor product of the SL_n-modules with respective decomposition polynomials p and q, computed using the Littlewood-Richardson rule; all

polynomials have their exponents in partition coordinates.

max_sub ((**grp** g)): **tex**. Returns the types of the maximal proper subgroups of g, represented textually as comma separated list; the list is obtained from a small database. The group g must be simple and of rank ≤ 8. Types for which more than one conjugacy class of subgroups exist have repeated occurrences in the list. See also *res_mat*.

max_sub (**int** i, (**grp** g)): **grp**. Returns the type of the i-th maximal proper subgroup of g in the list $max_sub(g)$. The group g must be simple and of rank ≤ 8. See also *res_mat*.

plethysm (**vec** λ, μ, (**grp** g)): **pol** [λ: *partition*, μ: *weight*, *result*: *decomposition*]. Returns the decomposition polynomial of the plethysm of V_μ corresponding to the partition λ. *Algorithm:* We use the classical Frobenius Formula (cf. [And] and [JaKe])

$$plethysm(\lambda,\mu) = \frac{1}{n!}\sum_{\kappa\in\mathcal{P}_n} class_ord(\kappa) * sym_char(\lambda,\kappa) * \bigotimes_{i=1}^{l(\kappa)} Adams(\kappa_i,\mu),$$

where $n = |\lambda|$, $\mathcal{P}_n$ denotes the set of partitions of n, and $l(\kappa)$ denotes the number of non-zero parts κ_i of κ. Hence the algorithm uses *partitions*, *class_ord*, *sym_char*, *Adams*, and *tensor*.

plethysm (**vec** λ, **pol** p, (**grp** g)): **pol** [λ: *partition*, p, *result*: *decomposition*]. This is similar to $plethysm(\lambda, \mu, g)$, but with the irreducible module V_μ replaced by the module with decomposition polynomial p.

p_tensor (**int** n, **vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *decomposition*]. Returns the decomposition polynomial of the n-th tensor power $V_\lambda^{\otimes n}$ of V_λ.

p_tensor (**int** n, **pol** p, (**grp** g)): **pol** [p, *result*: *decomposition*]. Returns the decomposition polynomial of the n-th tensor power of the g-module with decomposition polynomial p.

res_mat (**mat** R, (**grp** g)): **mat** [R: *roots*, *result*: *lin*(*weight*, *weight*)]. (restriction matrix) 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 fundamental Lie subgroup of g with root system Φ'. *Algorithm:* Let R consist of m roots, and let r be the Lie rank of g. The first m columns of the restriction matrix are easily computed: the (i,j)-entry is equal to $\langle \omega_i, R[j] \rangle$ for $1 \leq i \leq r$ and $1 \leq j \leq m$. The remaining $r - m$ columns define a map from the weight lattice to $\mathbf{Z}^{r-m}$ which vanishes on the roots in R and yet is surjective; it is computed by an extended Euclidean type algorithm. This does not completely specify these last columns, which is due to the absence of a canonical choice of fundamental weights for the part of the central torus of the fundamental Lie subgroup which lies in the semisimple part of the original group. Although the function checks whether the

rows of R are indeed roots, and whether they are linearly independent, it does not test whether they are positive roots and whether their mutual inner products are non-positive; these conditions should be met however in order to obtain a result suitable for use with *branch* and *collect*. Note that if one is in fact interested in the *semisimple* subgroup with the given root system Φ', then it suffices to simply discard the final $r - m$ columns.

res_mat (**grp***h*, (**grp** *g*)): **mat** [*result*: *lin*(*weight*, *weight*)]. Returns the restriction matrix for the maximal proper subgroup with type h of g, which is obtained from a small database. The group g must be simple and of rank ≤ 8. In case more than one non-conjugate subgroups of type h exist, the restriction matrix for the first one in the list is returned; in case no such subgroup exists, an error is reported. See also *max_sub*.

res_mat (**grp***h*, **int** *i*, (**grp** *g*)): **mat** [*result*: *lin*(*weight*, *weight*)]. Returns the restriction matrix for the i-th maximal proper subgroup with type h of g, which is obtained from a small database. The group g must be simple and of rank ≤ 8. See also *max_sub*.

spectrum (**vec** λ, t, (**grp** *g*)): **pol** [λ: *weight*, *t*: *toral*]. Let n be the last entry of t, then the toral element $t \in T$ will act in any representation of g as a diagonalisable transformation all of whose eigenvalues are n-th roots of unity. The function returns a polynomial in one indeterminate, in which the coefficient of the monomial X^i is the multiplicity of the eigenvalue ζ^i in the action of the toral element t on the irreducible g-module V_λ, where ζ is the complex number $e^{2\pi \mathbf{i}/n}$. The result can also be obtained by calling *branch* to compute the restriction to a one parameter subgroup containing the toral element, see Section 5.7.3. *Algorithm:* The element t naturally determines a linear function on $\Lambda(T)$ with values in $\mathbf{Z}/n$; this map is applied to the character polynomial of V_λ, which is generated using using *dom_char* and a dynamic version of *W_orbit*. In fact this algorithm is a simplified version of *branch*. (A pre-LiE version of this function, implemented only for E_n, has been used for [CoGr].)

spectrum (**pol** p, **vec** t(**grp** *g*)): **pol** [*p*: *decomposition*, *t*: *toral*]. This is similar to *spectrum*(λ, t, g), but with the irreducible module V_λ replaced by the module with decomposition polynomial p.

sym_tensor (**int** n, **vec** λ, (**grp** *g*)): **pol** [λ: *weight*, *result*: *decomposition*]. (symmetric tensor power) Returns the decomposition polynomial of $S^n(V_\lambda)$, the n-th symmetric tensor power of V_λ. See also *alt_tensor* and *plethysm*. *Algorithm:* We use the recursion

$$sym_tensor(n, \lambda) = \frac{1}{n} \sum_{k=1}^{n} tensor(sym_tensor(n-k, \lambda), Adams(k, \lambda)).$$

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

sym_tensor (**int** n, **pol** p, (**grp** g)): **pol** [p, *result*: *decomposition*]. This is similar to *sym_tensor*(n, λ, g), but with the irreducible module V_λ replaced by the module with decomposition polynomial p.

tensor (**vec** λ, μ, (**grp** g)): **pol** [λ, μ: *weight*, *result*: *decomposition*]. Returns the decomposition polynomial of the tensor product $V_\lambda \otimes V_\mu$. *Algorithm:* We use Klimyk's formula, see [Hum1], Exerc. 24.9. In terms of other LiE functions this formula can be expressed as *alt_dom*(*W_orbit*(*dom_char*$(\lambda, g), g) * X^\mu, g)$; for efficiency the weights λ and μ are possibly interchanged, in order to make V_λ the lower dimensional module of the two. Like in other functions, a dynamic version of *W_orbit* is used to prevent storage of a complete Weyl group orbit. For groups of type A_n, see also *LR_tensor*.

tensor (**vec** λ, μ, ν, (**grp** g)): **int** [λ, μ, ν: *weight*]. Returns the coefficient of the monomial X^ν in *tensor*(λ, μ, g). *Algorithm:* The same computational steps as in *tensor*(λ, μ, g) are performed, but the computed weights unequal to ν are ignored.

tensor (**pol** p, q, (**grp** g)): **pol** [p, q, *result*: *decomposition*]. Returns the decomposition polynomial of the tensor product of the g-modules with respective decomposition polynomials p and q.

tensor (**pol** p, q, **vec** ν, (**grp** g)): **pol** [p, q, *result*: *decomposition*, ν: *weight*]. Returns the coefficient of the monomial X^ν in *tensor*(p, q, g).

v_decomp (**pol** d, (**grp** g)): **pol** [d: *dominant*, *result*: *decomposition*]. (virtual decomposition) Returns the virtual decomposition polynomial of the virtual g-module with dominant character polynomial d. *Algorithm:* Although the algorithm of *decomp* could be used in this case as well, we have chosen a different approach: we compute *alt_dom*$(W_orbit(d, g), g)$, as always using a dynamic version of *W_orbit*. The validity of this formula is most easily verified by comparing with the formula given under *tensor*, taking for μ the zero weight. Experimental evidence suggests that in cases where the character polynomial to be decomposed is highly virtual, as in the application to *Adams*, this algorithm is significantly more efficient than that of *decomp*.

v_decomp (**vec** λ, (**grp** g)): **pol** [λ: *weight*, *result*: *decomposition*]. This is equivalent to *v_decomp*$(1X^\lambda, g)$.

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. For conciseness we have assumed that a default group has been chosen, so that we do not have to pass around an extra **grp** parameter all the time (although this would be a good thing to do when writing a library of functions; remember that one could then still omit the parameter when making a call to such a function). Some functions given in this chapter use other functions defined before them, which will not always be explicitly mentioned. A file containing all the function definitions of this chapter is supplied as part of the LiE distibution.

5.1. General

5.1.1. Reversing the ordering

For sorting the entries of a vector the is no choice of sorting criterion, unlike with sorting of matrices and polynomials. However, the only likely alternative to the decreasing ordering which *sort* gives is the increasing ordering, and it can be obtained by the call '$-sort(-v)$'. The same trick works for matrices as well, if one just wants to reverse the ordering for one matrix without affecting the global sorting criterion.

5.1.2. Union and intersection 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 '$unique(a\hat{}b)$'. For the intersection of two sets a little more work is needed. An obvious approach is to sort both matrices and then systematically search for equal rows among the two. This approach is slightly complicated by the fact that there is no direct way to compare two given rows in the selected ordering for matrix rows. Here is a slicker solution using polynomials.

$union(\mathbf{mat}\ a, b) = unique(a\hat{}b)$
$intersection(\mathbf{mat}\ a, b) = \ support(X\ unique(a) + X\ unique(b) - X\ union(a, b))$;

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** *s* = 0; **for** *i* **in** *v* **do** *s* += *i* **od;** *s*
prod(**vec** *v*) = **loc** *p* = 1; **for** *i* **in** *v* **do** *p* = *p* ∗ *i* **od;** *p*

Incidentally, there is another solution for the first case, namely to form the inner product with the all-one vector:

sum(**vec** *v*) = *v* ∗ *all_one*(*size*(*v*))

The latter solution is more efficient than the former one, because built-in operations (such as the standard inner product) are executed much more efficiently than programs executed by the interpreter.

5.1.4. The factorial function

There are many ways to compute the factorial $n!$ of a natural number n. An obvious recursive definition

fac(**int** *n*) = **if** *n* == 0 **then** 1 **else** *n* ∗ *fac*(*n* − 1) **fi**

will do fine, but there are faster and less obvious methods, such as

fac(**int** *n*) = *class_ord*([*n* + 1])

which surprisingly even gives a value for $(-1)!$.

5.1.5. Evaluating polynomials

LiE has no built-in operation for evaluating a polynomial in a point; such an operation is scarcely needed in the context in which LiE generally interprets polynomials (i.e., for decomposition polynomials and the like). It is not difficult however to write a function that performs such an evaluation, provided that no negative exponents occur (for otherwise the result cannot be computed over the integers).

```
eval_pol(pol p; vec pt) = loc s = 0;
    for i = 1 to length(p)
    do loc m = 1; for j = 1 to size(pt) do m = m * pt[j]^expon(p, i)[j] od;
        s += coef(p, i) * m
    od; s
eval_pol(pol p; int n) = eval_pol(p, [n])
```

5.1.6. The sum of the coefficients of a polynomial

There are several ways to compute the sum of all the coefficients of a polynomial p. First, one may consider this as a special case of polynomial evaluation: the call *eval_pol*(*p*, *all_one*(*n_vars*(*p*))) will return the desired answer, unless there are any negative exponents (since LiE will refuse to compute even 1^a if a is negative). A function written specifically for this purpose could of course avoid the exponentiation. A better method still is to use the fact that any polynomial can be interpreted as a (virtual) decomposition polynomial for a torus, and then the dimension is the desired answer; the value can therefore be obtained as *dim*(*p*, *Lie_group*(0, *n_vars*(*p*)). The probably fastest way is to use multiplication by a matrix to obtain a polynomial in 0 indeterminates (LiE provides automatic coercion of such polynomials to and from integers); the matrix in question has 0 columns: *p* ∗ *null*(*n_vars*(*p*), 0). Thus even multiplication by empty matrices can have its use.

5.1.7. Accessing the default group

Although the default group is implicitly provided (whenever necessary) as an additional parameter in function calls (independently of whether built-in or user defined functions are involved), its value can not be explicitly accessed within expressions. There is a way out of this problem however, by means of the following little trick: define a function on groups that returns its parameter, and call it without arguments.

$$default(\textbf{grp}\ g) = g$$

Now one can for instance assign the default group to a variable by the statement $g = default$, or obtain the central torus of the default group as $default[0]$.

5.1.8. Gaussian elimination over $\mathbf{Z}/p$

Here is a practical example of a program executed entirely by the interpreter, with no use of built-in functions. It performs Gaussian elimination over a prime field $\mathbf{Z}/p$. The input consists of a matrix m whose rows represent linear functions in $n_cols(m)$ unknowns with coeffients in $\mathbf{Z}/p$; the goal is to describe the subspace on which all these functions vanish. The algorithm performs row operations (adding one row to another, multiplying a row by a non-zero scalar and discarding null rows) until the first non-zero entry each row is unique in its column. The resulting rows are returned, sorted reverse lexicographically (which makes rows with non-zero entries at low indices come first). This result is easily interpreted as the solution for the homogeneous system of linear equations: each row expresses the unknown corresponding to its first non-zero entry as a linear combination of later unknows, and any unknown for which no row gives such an expression is a free parameter of the solution. Inhomogeneous systems can also be treated by tacitly assuming the final unknown to be 1, so that the last column can be used for the inhomogeneous parts of the equations. Then after performing the elimination, the final unknown should be a free parameter (if not the system is inconsistent), and it can be set to 1 in the solution.

A first ingredient needed is a division procedure in $\mathbf{Z}/p$. We use a slight variation of the extended gcd computation that was given in Section 2.6.1.

```
p = 1831 # or any other prime number #
div(int a, b) = a * inv(b) % p
inv(int x) =
{  loc m = [[x % p, 1, 0], [p, 0, 1]];
   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;
   if m[2,1] != 1 then error("division by 0") fi; m[2,2]
}
```

The Gaussian elimination is now a relatively simple matter.

```
Gauss(mat m) =
{  m = m % p; loc i = 1;
   while i ≤ n_rows(m)
   do if wipe(i) # true unless m[i] was null # then i += 1
      else m = m − i # cast away a null row #
      fi
   od;
   savestate; on − lex; sort(m); restorestate; m
}
wipe(int row_nr) =
{  loc v = m[row_nr];  # the row to wipe with #
   loc j = 1; while j ≤ size(v) && !v[j] do j += 1 od;
   if j > size(v) then return 0 # failure # fi;
   v = inv(v[j]) * v % p; # normalise, making leading coefficient 1 #
   for i = 1 to n_rows(m) do if i != row_nr then m[i] = m[i]−m[i,j]*v fi od;
   m[row_nr] = v; 1 # success #
}
```

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() = loc m = pos_roots; 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. The following appears to be a correct computation

```
rho(grp g) =  loc sum = null(Lie_rank(g));
   for alpha row pos_roots(g) do sum = sum + alpha od; sum/2
```

but in fact this rounds off the result downwards to the nearest root vector, so that for instance $rho(A_1)$ turns out to be [0], which is clearly unsatisfactory. The obvious solution is to use the weight basis instead of the root basis, and in that case we have $\rho = all_one(Lie_rank(g))$ for semisimple groups g, as can be easily shown (to prove $\langle \rho, \alpha_i \rangle = 1$ use the fact that the fundamental reflection r_i sends every positive root except α_i to a positive root). For groups containing a central torus we have to be a bit more accurate:

```
toral_dim(grp g) = Lie_rank(g[0])
ss_rank(grp g) = Lie_rank(g) − toral_dim(g)
rho(grp g) = all_one(ss_rank(g))^null(toral_dim(g))
```

5.2.3. Positive roots made negative by w

It is known that for each Weyl group element w there are exactly $l(w)$ positive roots α such that $\alpha \cdot w$ is a negative root. Here is a function that computes this set of roots.

```
inverse(vec w) = loc l = size(w); loc wi = w;
    for i = 1 to l do wi[i] = w[l + 1 − i] od; wi
pos_neg(vec w) = intersection(pos_roots, W_rt_action(−pos_roots, inverse(w)))
```

We have used the function *intersection* that was defined in Section 5.1.2.

5.3. Weyl words

5.3.1. 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 way to compute it based on the definitions.

```
cox_mat() = loc m = id(ss_rank);
    for i = 1 to n_rows(m) − 1 do for j = i + 1 to n_rows(m)
    do m[i, j] = ord(W_action([i, j])); m[j, i] = m[i, j] od od; m

ord(mat m) = loc p = m; loc i = 1; loc idmat = id(n_rows(m));
    while p != idmat do p = p ∗ m; i += 1 od; i
```

A more efficient way to compute the Coxeter matrix is to use the information contained in the Cartan matrix directly:

```
cox_mat() = loc m = id(ss_rank); loc c = Cartan;
    for i = 1 to n_rows(m) − 1 do for j = i + 1 to n_rows(m)
    do m[i, j] = 2 + c[i, j] ∗ c[j, i]; if m[i, j] == 5 then m[i, j] = 6 fi;
        m[j, i] = m[i, j]
    od od; m
```

5.3.2. 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 *next_rewrite* forms a basic ingredient in the enumeration of all equivalent Weyl words: it produces the indicated replacement (if possible) in the Weyl word w for the subword that begins at the k-th entry of w.

```
# try rewriting reduce(w) at position k #
next_rewrite(vec w; int k) =
{   w = reduce(w); if k+1 > size(w) then return w fi;
    loc m = cox_mat[w[k], w[k+1]]; if k+m-1 > size(w) then return w fi;
    for i = k+2 to k+m-1 do if w[i] != w[i-2] then return w fi od;
    loc t = w[k+m-2]; for i = k to k+m-2 do w[i] = w[i+1] od;
    w[k+m-1] = t; w
}
```

The function *cox_mat* is as in the previous subsection. In order to find all reduced expressions for a given Weyl group element, we may use a standard closure algorithm.

```
# produce a list of all reduced expressions for a given w ∈ W #
all_rewrites(vec w) =
{   w = reduce(w); loc l = size(w); loc m = [w]; loc i = 0;
    while i < n_rows(m)
    do i += 1;
       for j = 1 to l-1
       do loc next = next_rewrite(m[i], j); loc found = 0;
          for k = 1 to n_rows(m)
          do if m[k] == next then found = 1; break fi od;
          if !found then m = m + next fi
       od
    od; m
}
```

Incidentally there is a completely different way to solve this problem, which is analogous to the way *W_word* is implemented. We first produce a weight x that is made strictly dominant by the action of w, namely $x =$ *W_action*(*rho*, *inverse*(w)). Then those fundamental reflections that may start a Weyl word for w are the ones whose action brings x closer to the dominant chamber, and these are immediately recognisable by the fact that x has a negative entry at the corresponding position. Having found such a position we may apply the reflection to x and repeat the procedure. The routine implementing *W_word* always chooses the first possible fundamental reflection, but the routine *all_W_words* below tries every possibility and proceeds recursively to obtain all possible continuations.

```
all_W_words(vec prefix, x) =
{   loc m = null(0, l);
    for i = 1 to ss_rank
    do if x[i] < 0 then m = m^all_W_words(prefix+i, W_action(x, [i])) fi od;
    if n_rows(m) > 0 then m else  #  x is dominant # [prefix] fi
}
alt_rewrites(vec w) =
    loc l = length(w); all_W_words([], W_action(rho, inverse(w)))
```

This routine is faster that *all_rewrites*, and it produces the Weyl words in lexicographic

order.

5.3.3. The Bruhat ordering

Let us first reproduce from a previous version of this manual—as an example of how certain functions were implemented in the interpreter before they were built in—a function that computes the Bruhat descendents of a Weyl group element, which is now available as *Bruhat_desc*. The default group is assumed to be semisimple.

```
bruhat(vec w) =
{ loc w = reduce(w); loc m = null(0, size(w) − 1);
  for i = 1 to size(w)
  do loc v = w − i; if length(v) == size(w) − 1 then m = m + v fi od;
  # it remains to check whether two rows represent words #
  # corresponding to the same Weyl group element #
  m = unique(m); loc rho = all_one(Lie_rank);
  for i = 1 to n_rows(m) − 1 do for j = i + 1 to n_rows(m) do
    if W_action(rho, m[i]) == W_action(rho, m[j]) then m[j] = m[i] fi
  od od;
  unique(m)
}
```

Note how thanks to the (new) function *canonical* the second part can be made more efficient: it suffices to return *unique*(*canonical*(*m*)). Better still, the whole second part can be omitted, as it is not difficult to show that any two Bruhat descendents produced by the initial **for** loop cannot represent the same Weyl group element: they would contradict the reducedness of the initial expression. These considerations have now been incorporated in the built-in algorithm for *Bruhat_desc*; in particular the Weyl words it returns are *not* necessarily canonical ones.

Now let us consider using the built-in functions for the Bruhat order to traverse the interval in the Bruhat order between to given Weyl group elements x and y. To be concrete about what we do with the elements found, let us just count them at each level, producing a polynomial in one indeterminate with a term nX^i if there are n elements of length i between x and y in the Bruhat order. We shall use the version of *Bruhat_desc* with two Weyl word arguments.

```
inter_pol(vec x, y) =
{ loc result = poly_null(1);  if ! Bruhat_leq(x, y) then return result fi;
  loc l = length(y);  loc lx = length(x);  loc level = [y];
  while l ≥ lx
  do result += n_rows(level)X l;
    loc nl = canonical(Bruhat_desc(x, level[1]));
    for i = 2 to n_rows(level)
    do nl = unique(nl^canonical(Bruhat_desc(x, level[i])) od;
    level = nl; l = l − 1
  od; result
}
```

Note that *canonical* is always immediately applied to the result of *Bruhat_desc*, and that *unique* is called each time new words are added, in order to limit the size of the intermediate values as much as possible. Nevertheless this way of traversing parts of the Weyl group may lead to memory problems for large Weyl groups and elements x and y which differ considerably in length, since the majority of the elements will occur near the middle levels.

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 [BrCo].

5.4.1. Right cosets

Suppose S is a subset of $\{1,\ldots,r\}$ and W is a Weyl group of rank r; let W_S be the subgroup of W generated by $\{\, r_i \mid i \in S \,\}$. When enumerating cosets of W_S a natural choice of representatives is that of the *distinguished coset representatives*, i.e., the representatives of minimal length. We may use the natural bijection between the (Weyl) orbit of a vector and the set of right cosets of its stabiliser in order to enumerate the right cosets for W_S, using the fact that W_S is the stabiliser of any dominant weight that has zeros precisely at those positions whose index occurs in S. Calling such a weight having ones on all other positions the characteristic vector for S, we can write the following functions.

```
char_v(vec s) = loc y = rho; for i = 1 to size(s) do y[s[i]] = 0 od; y
r_cosets(vec r) = for x row W_orbit(char_v(r)) do print(W_word(x)) od
```

For example, the right coset representatives for the subsystem $A_1A_1A_1$ in D_4 can now be found as follows:

```
setdefault D4; s = [1,3,4]; r_cosets(s)
```

The Weyl words printed as a result of this all end (with the exception of the empty word) with the simple reflection 2, which is reassuring: obviously a Weyl word which is right reduced for S shouldn't end with any of the reflections contained in S. Note that $[1,3,4]$ represents the nodes of the Dynkin diagram corresponding to a subsystem $A_1A_1A_1$ of D_4, which can be verified by calling *diagram*(D_4). Another—more elaborate—way of verifying this is to ask for the Cartan type of the subsystem generated by the fundamental roots with indices 1, 3, and 4, i.e., by computing *Cartan_type*$(id(4)-2, D_4)$.

5.4.2. Left cosets

Using the fact that a Weyl word w is left reduced with respect to the subset S if and only if its inverse 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; we use the function *inverse* defined above in Section 5.2.3.

```
l_cosets(vec l) =
    for x row W_orbit(char_v(l)) do print(inverse(W_word(x))) od
```

5.4.3. Double cosets

We now construct a function *double_cosets* 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 *r_cosets* to the effect that it only prints those Weyl words (already right reduced for R) that are left reduced for L.

double_cosets(**vec** l, r) =
  for x **row** $W_orbit(char_v(r))$ **do loc** $w = W_word(x)$;
    if $w == l_reduce(l, w)$ **then** $print(w)$ **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. A good upper bound for the number of columns needed is $lr_reduce(v, long_word, w)$. An alternative efficient way of storing the coset representatives is to encode the Weyl word w by the weight $W_action(rho, reverse(w))$, which always has the same size *Lie_rank* (this is generally much less then the upper bound for $size(w)$), and from which w can be retrieved by calling *W_word*. This idea applies in many other situations as well, in which one has to deal with subsets of the Weyl group.

5.5. Traversing the Weyl group

For some applications it may be necessary to perform a certain operation upon all the elements of the Weyl group. Basically this is not difficult: if *action* is a function performing the desired operation, then in analogy to the example of enumerating right cosets one may write

for r **row** $W_orbit(all_one(Lie_rank))$ **do** $action(W_word(r))$ **od**

to obtain the desired effect. However, this solution has one drawback, rendering it useless for many applications, which is that it needs an amount of memory proportional to the order of the Weyl group to store the Weyl orbit. What we would like to have is a way of traversing the Weyl group dynamically, without storing any information permanently for each element. The built-in functions have access to a routine that generates Weyl group orbits, but this routine is not available from the interpreter level; there is no such construction as '**Weyl** w **do** $action(w)$ **od**'. We will present a set of functions that enable traversing the Weyl group without excessive storage requirements. A price has to be paid in terms of speed, since the interpreter has to perform the major part of the computation (as an indication, for the group E_6 the approach first mentioned is still feasible on large computers, and is about 7 times as fast as the procedure below, while the orbit generation proper (without the **for** loop) is more than 100 times as fast).

The basic idea we use is to find a large subgroup of the kind W_S of Section 5.4, to enumerate its right cosets using *W_orbit* as described in that section (this requires only a small amount of memory) and then for each such coset to recursively enumerate the

elements of W_S, appending the resulting Weyl words to the distinguished representative of the current right coset. The largest proper subgroups of W of the kind W_S are obviously those for which only one fundamental reflection is absent from S; moreover it is usually best to omit a reflection at an end of the Dynkin diagram. To inspect the orbit sizes involved one can use

inspect(**grp** *g*) = **for** *r* **row** *id*(*Lie_rank*(*g*)) **do** *print*(*W_orbit_size*(*r*, *g*)) **od**

We will not fix some specific choice of reflection to omit, but rather allow the user to specify a function *choose* which determines for each group the index of the reflection to omit; a reasonable choice for simple groups is

choose(**grp** *g*) = **if** *Lie_code*(*g*)[1] == 5 **then** *Lie_rank*(*g*) **else** 1 **fi**

Having chosen a reflection to omit, we have to determine the subgroup generated by the remaining fundamental reflections. This subgroup can be obtained by applying *Cartan_type* to the corresponding set of fundamental roots. We are not interested in the central torus, however, so we make the subgroup semisimple. The following function uses a "feature" which would have been considered a bug if it weren't so handy for this purpose: the function *Lie_group* can produce tori of negative rank.

semisimple(**grp** *g*) = *g* ∗ *Lie_group*(0, −*toral_dim*(*g*)) # kill the central torus #

We also need to know the correspondence of the fundamental reflections of the subgroup to those of the containing group. This can be achieved by calling *fundam* with the same set of roots as *Cartan_type*: the effect is not so much to make the set fundamental (which it already is) but to reorder the roots in such a way that their ordering corresponds to the standard labeling of the Dynkin diagram of the subgroup. The resulting matrix is almost a permutation matrix: its i-th row is the j-th unit vector, where j is the index of the reflection in the whole group corresponding to the i-th fundamental reflection of the subgroup. This information is more compactly represented by a vector with j as its i-th entry, which can be obtained by multiplying the matrix to the right by the vector $[1, 2, 3, \ldots]$. If we call the resulting vector v, then Weyl words for the subgroup can be translated to Weyl words for the larger group by replacing any entry i by $v[i]$. Since we are using recursion, we must maintain a cumulative translation vector for translating directly to Weyl words for the original group; this is accomplished by using a previous translation vector instead of the vector $[1, 2, 3, \ldots]$ above. The remaining details can be found in the following program; we revert to the method mentioned at the beginning of this section as soon as the rank of the group has been brought down to 3. The only further subtle point is the function *translate* used with **make**, which accesses the vector *trans* as non-local variable; the identification rules of LiE ensure that it always uses the most recent instance of the parameter of that name to the recursive function *trav*.

```
translate(int i) = trans[i]
trav(grp g; vec prefix, trans) =
{  loc s = ss_rank(g);
   if s ≤ 3 then
      for r row W_orbit(rho(g), g)
```

```
        do action(prefix^make(translate, W_word(r, g))) od;
    else loc c = choose(g); loc roots = id(s); loc sub_roots = roots − c;
        loc h = semisimple(Cartan_type(sub_roots, g));
        loc new_trans = fundam(sub_roots, g) ∗ trans;   # cumul. translation #
        for r row W_orbit(roots[c], g) # orbit of vector with stabiliser h #
        do trav(h, prefix^make(translate, W_word(r, g)), new_trans) od
    fi
}

ii(int n) = n
traverse(grp g) =  g = semisimple(g); trav(g, [ ], make(ii, Lie_rank(g)))

action(vec w) = print(w) # a useful initial setting #
```

We leave it as an exercise to write a similar program for traversing the orbits of weights with small but non-trivial stabilisers.

5.6. Kazhdan-Lusztig polynomials

LiE provides a basic routine for computing Kazhdan-Lusztig polynomials. It has not been optimised very much for speed (in particular no attempt is made to remember previously computed polynomials), and difficult cases in large Weyl groups may prove to be problematic. For a Weyl group like that of type B_3 however there are no problems. The following routines compute all relevant values for that group, and present the results compactly in the format used in [Shi], p. 23; this format is based on the observation that only very few different values occur for Kazhdan Lusztig polynomials in small groups. For the pairs of Weyl group elements which are not listed the Kazhdan-Lusztig polynomials are either zero or one (i.e., X^0), depending on whether *Bruhat_leq* does or does not hold for the pair. For the pairs that are listed the kind of bracketing indicates the value: no bracketing signifies $X^1 + X^0$, parentheses $X^2 + X^0$, square brackets $X^2 + X^1 + X^0$, and inverted angle brackets any other value, but these don't occur for B_3 (for other groups other encodings would be more appropriate).

The routines below taken care to list all Weyl group elements in canonical form, and to order then by increasing length and lexicographically within the same length class. The routine *list_KL* is a variation of *inter_pol* shown earlier, which spends some extra effort to generate the elements in order of *increasing* length: the order reversing involution of multilplying with the longest element of W is used for this purpose. The function *table_KL* illustrates an alternative way to traverse the Weyl group, useful when a sensible order of traversal is more important than time or memory considerations. It uses the fact that any initial segment of a canonical expression for a Weyl group element is again a canonical expression (in fact this holds for any subsegment). The remaining routines mainly concern themselves with formatting the output.

```
# table of Kazhdan-Lusztig polynomials, most suited for B_3 #
str(vec w) =
    if w == [] then "e"
    else loc s = ""; for i = 1 to size(w) do s = s + w[i] od; s
    fi

buf = ""; first = 1
output(tex t) = if length(buf + t) > 70 then print(buf + ","); buf = "   " fi;
    buf = buf + if first || buf == "   " then t else buf + ",  " + t fi; first = 0
flush() = if !first then print(buf + "."); first = 1 fi

list_KL(vec x) =
{   loc level = [reduce(x^long_word)]; loc l = length(level[1]);
    buf = str(x) + ":  ";
    while l > 0 # case l = 0 would not produce any output #
    do loc nl = null(0, l - 1);
        for r row level
        do nl = unique(nl^canonical(Bruhat_desc([], r)));
            loc y = reduce(r^long_word); loc p = KL_poly(x, y);
            if p != X0 then output
            (   if p == X1 + X0 then str(y) else
                if p == X2 + X0 then "(" + str(y) + ")" else
                if p == X2 + X1 + X0 then "[" + str(y) + "]" else
                    ">" + str(y) + "<" fi fi fi
            ) fi
        od; level = nl; l += -1
    od; flush
}
table_KL() =
{   loc level = [[]]; loc l = 0;
    while n_rows(level)
    do l += 1; loc nl = null(0, l);
        for w row level
        do list_KL(w);
            for i = 1 to ss_rank do if w+i == canonical(w+i) then nl += w+i
            fi od
        od; level = nl
    od
}
```

5.7. Toral elements

In LiE, a toral element is represented by a vector $v = [a_1, \ldots, a_r, d]$. This vector corresponds to the element t of the maximal torus T with $t^{\omega_i} = e^{2\pi \mathbf{i} a_i/d}$ for $1 \leq i \leq r$. The representation of the toral element t by the vector v is not unique, but it can be

made so by the following function.

toral(**vec** v) = $v = v/gcd(v)$; **loc** $s = size(v)$; $(v - s)$ % $v[s] + v[s]$

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

For the special linear group $SL(n, \mathbf{C})$ there is a much more familiar way to describe a toral 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 toral element whose diagonalised form has entries $[\zeta^{b_1}, \ldots, \zeta^{b_n}]$ along the main diagonal, where $\zeta = e^{2\pi \mathbf{i}/d}$ 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 *mk_toral* (an abbreviation for make toral element), to the vector $[b_1, \ldots, b_n]$ and the number d:

mk_toral(**vec** b; **int** d) = **loc** $n = size(b)$;
for $i = 2$ **to** $n - 1$ **do** $b[i] = (b[i - 1] + b[i])$ % d **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. One might wonder if the functions *to_part* and *from_part* could be of any use for our present purpose, since they are concerned with coordinate transformations to and from "diagonal entry" coordinates for groups of type A_n. However they deal with weights whereas our current problem is stated for toral elements, which are in a sense dual to weights; therefore they cannot be directly applied. To be sure, the second line of *mk_toral* could be replaced by *to_part*$(-b)$ % $d - 1 - n + d$, (recall that the operators '$-$' and '$+$' associate to the left!) provided that *sum*(b) is indeed divisible by d, but this solution seems to be rather artificial.

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

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 inner product $(\,\cdot\,, \cdot\,)$ fixed by $SO(12, \mathbf{C})$ satisfies $(e_i, e_j) = (f_i, f_j) = 0$ and $(e_i, f_j) = \delta_{i,j}$ 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 $\zeta = e^{2\pi \mathbf{i}/d}$. Then the following function, using the given matrix m, transforms the vector $[a_1, a_2, \ldots, a_6]$ into the form used by LiE to represent t.

$m = [[2,2,2,2,1,1],[0,2,2,2,1,1],[0,0,2,2,1,1],$
$[0,0,0,2,1,1],[0,0,0,0,1,1],[0,0,0,0,-1,1]]$
mk_tor_d6(**vec** *a*; **int** *d*) = *toral*((*a* ∗ *m*) % (2 ∗ *d*) + 2 ∗ *d*)

Some explanation may be in its place as to how we obtained this function. Let $\varepsilon_1, \ldots, \varepsilon_6$ be weights for the maximal torus T of $SO(12,\mathbf{C})$, where ε_i assigns to a diagonal matrix its i-th diagonal entry. Then it is well known that the fundamental roots of $SO(12,\mathbf{C})$ are the weights $\varepsilon_i - \varepsilon_{i+1}$ for $i = 1, \ldots, 5$, together with $\varepsilon_5 + \varepsilon_6$, and that the weights ε_i form an orthonormal system in $\Lambda(T)$. From this one can deduce what the fundamental weights are (we should have $(\omega_i, \alpha_j) = \delta_{i,j}$ since all fundamental roots have norm 2), and these turn out to be $\omega_i = \sum_{j=1}^{i} \varepsilon_j$ for $1 \leq i \leq 4$, together with $\omega_5 = \frac{1}{2}(\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_4 + \varepsilon_5 - \varepsilon_6)$ and $\omega_6 = \frac{1}{2}(\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_4 + \varepsilon_5 + \varepsilon_6)$. The fractional factors in the last two expressions may seem a bit surprising; indeed, as a weight for T we would have for instance that ω_6 corresponds to taking the square root of the product of the first 6 diagonal entries, which is not well defined. The explanation is that ω_5 and ω_6 are only defined as weights for the maximal torus of the twofold covering $Spin(12,\mathbf{C})$ of $SO(12,\mathbf{C})$; they do not occur in the weight lattice of T. Nevertheless we have to use these fundamental weights, since LiE deals only with the simply connected simple groups.

To obtain the entries of the toral element t, it now suffices to apply each of the fundamental weights ω_i to t; each ω_i accounts in this way for a column of m given above. To avoid the fractions that might result from ω_5 and ω_6 we have multiplied all entries, including the final denominator entry, by 2; in fact this corresponds to lifting t from T to the maximal torus of $Spin(12,\mathbf{C})$ which covers it twofold. Afterwards any common divisor present in all entries are removed, which leaves the meaning of a toral element invariant. For example we have *mk_tor_d6*([0,1,0,0,0,0],7) = [0,2,2,2,1,1,14] and *mk_tor_d6*([0,0,1,0,1,0],7) = [0,0,1,1,1,1,7]. The fact that the first element turns out to have order 14 rather than 7, is due to the lifting to $Spin(12,\mathbf{C})$. The 7-th power of the lift of this element is [0,0,0,0,7,7,14], which is equal to [0,0,0,0,1,1,2]; this is an element in the center of $Spin(12,\mathbf{C})$ (which can be checked by calling *center*(D_6)), which moreover lies in the kernel of the morphism $Spin(12,\mathbf{C}) \to SO(12,\mathbf{C})$ (this can be checked by calling *spectrum*, see the next subsection).

5.7.3. Spectrum

The function *spectrum* provides a means to recognise the toral element specified in a more natural form. For instance, we perform the following computation for a toral 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 $3X[0] + 2X[1]$, showing that t has 3 eigenvalues 1, and 2 eigenvalues -1 in the standard representation. It is therefore conjugate to the element *mk_toral*([0,0,0,1,1],2) (with *mk_toral* as in Section 5.7.1), which equals [0,0,0,1,2]. The element t itself can be obtained as an image of *mk_toral* by an appropriate permutation of the eigenvalues: we have t == *mk_toral*([1,1,0,0,0],2). By replacing the

final entry of t by 0 we obtain a vector representing a one parameter subgroup that contains t; information about this whole subgroup may be obtained using the function *branch*. The restriction matrix needed for such a 1-dimensional torus is essentially obtained by transposition of the vector leaving out the final entry; in the current case this restriction matrix can be expressed as $*[t-5]$. Computing $branch(sr, T_1, *[t-5])$ we find the polynomial $1X[-1] + 3X[0] + 1X[1]$, indicating that the element of that one parameter subgroup parametrised by some $z \in \mathbf{C}^*$ has one eigenvalue z^{-1}, three eigenvalues 1, and one eigenvalue z in the standard representation; 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 *centr_type*(t), which returns $A_1A_2T_1$, and the centraliser *centr_type*$(t - 5 + 0)$ of the one parameter subgroup containing it is A_2T_2. On the other hand we may compute $spectrum(adjoint, t) \mid [0]$, which will return the dimension of the Lie subalgebra fixed (eigenvalue $(-1)^0$) by t in the adjoint representation. Since the adjoint action is by conjugation this number should be equal the dimension of (the Lie algebra of) the centraliser of t. The call returns 12, which is indeed equal to $dim(A_1A_2\,T_1)$. If we want to verify the dimension of the centraliser of the one parameter subgroup containing t in a similar way, then we may again use *branch*: the call $branch(adjoint, T_1, *[t-5])$ returns $1X[-2]+6X[-1]+10X[0]+6X[1]+1X[2]$, which shows that this centraliser 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(**pol** p; **vec** t) = **loc** $r = size(t)$; $branch(p, T_1, *[t-r])$ % $[t[r]]$

The built-in function *spectrum* is slightly more efficient than this way of using branch. As a final example of its use, let $c = [0, 0, 0, 0, 1, 1, 2]$ be the central toral element of the group $Spin(12, \mathbf{C})$ which was considered in the previous subsection; we claimed that its image in $SO(12, \mathbf{C})$ was the identity. To verify this we first of all need to know which highest weight corresponds to the (standard) representation of $Spin(12, \mathbf{C})$, acting as $SO(12, \mathbf{C})$ on a 12-dimensional space. From the descriptions above it is obvious that $\varepsilon_1 = \omega_1$ is a weight of the standard representation (whose weight space is spanned by the first basis vector), and in fact it is its highest weight. To check our claim we set **setdefault** D_6 and compute $spectrum([1, 0, 0, 0, 0, 0], c)$; the result $12X[0]$ shows that indeed c acts trivially on the 12-dimensional space. We may perform this calculation for all 6 fundamental representations of $Spin(12, \mathbf{C})$ in one go by typing

for r **row** $id(6)$ **do** $print(spectrum(r, c))$ **od**

and find in this way (try it) that c acts non-trivially only in the last two fundamental representations (with highest weights ω_5 and ω_6); these representations are called *spin representations*, and there is no corresponding representation of $SO(12, \mathbf{C})$ for them.

As an extra result of our computations we easily read off the respective dimensions of the 6 fundamental representations.

5.7.4. Branching to a centraliser

We continue with the toral element of the preceding subsection. We wish to compute how the standard representation decomposes when restricted to the centraliser of the toral element t, which we have already seen to be of type $A_1A_2T_1$. We start with computing the centraliser more explicitly by calling *cent_roots*(t) which 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 $h =$ *Cartan_type*(f), which give, respectively,

$$f = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \text{and} \qquad h = A_2A_1T_1.$$

Note that the order of the simple factors of h differs from that in *centr_type*(t); this is due to the fact that, contrary to *centr_type*, *fundam* and *Cartan_type* sort their roots before grouping them according to the simple factors (whence their result may differ when another sorting criterion is selected, but is independent of the initial ordering of the roots). In order to branch to the centraliser we need the restriction matrix $m =$ *res_mat*(f); this gives

$$m = \begin{pmatrix} 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 6 \\ 0 & 1 & 0 & 4 \\ 1 & 0 & 0 & 2 \end{pmatrix}.$$

Finally we compute *branch*(sr, h, m) which returns $1X[0,0,1,3] + 1X[0,1,0,-2]$. Note that it is important to supply the group h here as second argument, and not *centr_type*(t) which has its simple components in the wrong order: the order of these components should match the order of the columns of the restriction matrix. In our case the restriction matrix was obtained via *res_mat* from the matrix f of roots, and the order of those roots correspond the order of the factors in h (as a rule, if the restriction matrix is obtained by fres_mat preceded by either *fundam* or *closure*, and the subgroup is computed by *Cartan_type* from the same set of roots, the orderings match).

It is interesting to attempt to interpret the result of branching the standard representation of SL_5 to the centraliser of t. To that end, we must first find the individual factors A_2, A_1, and T_1 of that centraliser; they can be seen to be respectively

the group SL_3 with its non-trivial entries in the lower right hand 3×3 block, the group SL_2 living similarly in the upper left hand 2×2 block, and the one parameter subgroup of diagonal matrices in SL_5 of the form

$$\begin{pmatrix} a & 0 & 0 & 0 & 0 \\ 0 & a & 0 & 0 & 0 \\ 0 & 0 & b & 0 & 0 \\ 0 & 0 & 0 & b & 0 \\ 0 & 0 & 0 & 0 & b \end{pmatrix} \quad \text{or equivalently,} \quad \begin{pmatrix} z^3 & 0 & 0 & 0 & 0 \\ 0 & z^3 & 0 & 0 & 0 \\ 0 & 0 & z^{-2} & 0 & 0 \\ 0 & 0 & 0 & z^{-2} & 0 \\ 0 & 0 & 0 & 0 & z^{-2} \end{pmatrix}.$$

The decomposition of the restriction to this centraliser of the standard representation is obviously that into the spaces spanned by the first two and by the last three standard basis vectors. Grouping the exponents in the decomposition polynomial $p = 1X[0,0,1,3] + 1X[0,1,0,-2]$ of this restriction according to the factors of $h = A_2A_1T_1$, we see that the highest weight $[0,0,\quad 1,\quad 3]$ corresponds to the space spanned by the first two basis vectors (with SL_3 acting trivially, SL_2 in its standard representation, and the one parameter group T_1 by third powers), while the highest weight $[0,1,\quad 0,\quad -2]$ corresponds to the space spanned by the last three basis vectors. Another way to identify the two components of the restriction is by computing $dim(p[1],h)$ and $dim(p[2],h)$, which return 2 and 3, respectively.

The paradoxical fact that the polynomial p claims that SL_3 acts on the latter component by its *second* fundamental representation ($[0,1]$), which is the contragredient of its first (standard) representation, while our "decomposition by hand" indicates a standard representation of SL_3, can be explained as follows. The matrix f of roots lists them in the reverse order to what is most natural (the fourth fundamental root coming before the third), and we have consequently used an embedding of the group h into SL_5 which applies the "diagram automorphism" of A_2 (taking transpose inverses) to that factor of h. The reader who wishes to find out exactly where these orderings play a rôle is urged to try the same computations after selecting a different sorting criterion, e.g., by `on - lex`.

5.8. Computing and decomposing characters

5.8.1. The character polynomial

A first matter of interest for a given irreducible representation is to find out which weights occur in it, and with which multiplicities, in other words, to find the character of the representation. Somewhat surprisingly there is no function *char* built into LiE to compute characters. A basic way to compute the character of a representation is to use *branch*: if r is the Lie rank of the group then for any dominant weight λ, the call $branch(\lambda, T_r, id(r))$ will compute the decomposition of the restriction to the maximal torus T of the irreducible representation with highest weight λ, and this is by definition the character of that representation. A simpler way however is to use the function *Demazure*: it was mentioned in Chapter 4 that calling this function without a Weyl word argument (which implies the default *long_word*), can be used to compute characters. We therefore define

char(**vec** *lambda*) = *Demazure*(*lambda*)

as a way to compute character polynomials. Neither *branch* nor *Demazure* provide the most efficient way of computing characters (a better method is discussed below), but for our rather simple examples this is of little concern.

We use this function to compute the character of the standard representation of SL_4:

setdefault A_3; *char*([1, 0, 0])

which returns the polynomial $1X[-1,1,0] + 1X[0,-1,1] + 1X[0,0,-1] + 1X[1,0,0]$, indicating the 4 weights, all occurring with multiplicity 1. To interpret these weights—as we have done before—in terms of 'diagonal entries', let ε_i be the weight of the torus T of diagonal matrices in SL_4 which assigns to any $t \in T$ its i-th diagonal entry. Then the fundamental weights are given by $\omega_1 = \varepsilon_1$, $\omega_2 = \varepsilon_1+\varepsilon_2$ and $\omega_3 = \varepsilon_1+\varepsilon_2+\varepsilon_3$, while $\varepsilon_1+\varepsilon_2+\varepsilon_3+\varepsilon_4 = 0$ since the elements of T have determinant 1. Using this it is not difficult to see that the weights in the character computed above are equal to ε_2, ε_3, ε_4, and ε_1, which is as it should be since the weight spaces for T in the standard representation of SL_4 are obviously those spanned by the 4 standard basis vectors.

As the ordering produced by sorting the weights lexicographically is somewhat unnatural, we change it to the decreasing height ordering, which puts the highest weight in front:

`on - height; $`

which gives $1X[1,0,0] + 1X[-1,1,0] + 1X[0,-1,1] + 1X[0,0,-1]$. The ε-coordinates are in fact just what was called 'partition coordinates' in Section 3.4, so we may convert to these coordinates by means of the function *to_part*: calling *to_part*(\$) returns $1X[1,0,0,0] + 1X[0,1,0,0] + 1X[0,0,1,0] + 1X[-1,-1,-1,0]$, which is not exactly what we computed by hand, but equivalent to it modulo the relation on the ε_i's.

If we want to have weights which are uniquely expressible in partition coordinates, we should consider representations of GL_n rather than of SL_n; extra information will then be contained in the coefficient of the n-th fundamental weight of the group of type $A_{n-1}T_1$, which coefficient equals the sum of the partition coordinates. The coordinate transformations from and to ε-coordinates can then be given by

```
from_eps(vec lambda) = from_part(lambda) + sum(lambda)
from_eps(pol p) = loc s = poly_null(n_vars(p));
    for i = 1 to length(p) do s += coef(p,i)X from_eps(expon(p,i)) od; s
to_eps(vec wt) = loc n = size(wt);  loc v = to_part(wt - n);
    v + (wt[n] - sum(v))/n * all_one(n)
to_eps(pol p) = loc s = poly_null(n_vars(p));
    for i = 1 to length(p) do s += coef(p,i)X to_eps(expon(p,i)) od; s
```

We now repeat the computations above for GL_4 instead of SL_4, keeping in mind that the standard representation is now encoded by the highest weight $[1,0,0,1]$:

setdefault $A_3\,T_1$; $char([1,0,0,1])$

returns $1X[1,0,0,1]+1X[-1,1,0,1]+1X[0,-1,1,1]+1X[0,0,-1,1]$; we then call $to_eps(\$)$, which yields $1X[0,1,0,0]+1X[1,0,0,0]+1X[0,0,1,0]+1X[0,0,0,1]$ as desired. A slightly larger representation of GL_4 is the irreducible representation with highest weight $[0,1,0,2]$ (corresponding to ω_2 for SL_4), which is the second exterior power of the standard representation (this fact can be verified by calling $alt_tensor(2,[1,0,0,1])$). Its character $char([0,1,0,2])$ is equal to

$$1X[0,1,0,2]+1X[1,-1,1,2]+1X[1,0,-1,2]+1X[-1,0,1,2]+\\ 1X[-1,1,-1,2]+1X[0,-1,0,2]$$

which is converted by *to_eps* to

$$1X[1,1,0,0]+1X[0,1,1,0]+1X[1,0,1,0]+1X[0,1,0,1]+1X[1,0,0,1]+1X[0,0,1,1],$$

which is in accordance with the fact that the weights of this representation are $\varepsilon_i+\varepsilon_j$ for $1\leq i<j\leq 4$, where the corresponding weight space is spanned by the wedge $e_i\wedge e_j$ of standard basis vectors.

Note that for both representations considered so far the set of weights forms a single W-orbit, which is evident in the partition coordinates since W acts by permutation of these; for the fundamental weight coordinates it can be checked by calling for instance $W_orbit([0,1,0,2])$. Only few representations, called miniscule representations, have this property; it is more typical that there are several W-orbits of weights, some of them with non-trivial multiplicity, as is exemplified by $char([1,1,0,3])$:

$$1X[\ 1,\ 1,\ 0,3]+1X[\ 2,-1,\ 1,3]+1X[-1,\ 2,\ 0,3]+1X[\ 2,\ 0,-1,3]+\\ 2X[\ 0,\ 0,\ 1,3]+1X[\ 1,-2,\ 2,3]+2X[\ 0,\ 1,-1,3]+1X[-2,\ 1,\ 1,3]+\\ 2X[\ 1,-1,\ 0,3]+1X[-1,-1,\ 2,3]+1X[-2,\ 2,-1,3]+1X[\ 1,\ 0,-2,3]+\\ 2X[-1,\ 0,\ 0,3]+1X[\ 0,-2,\ 1,3]+1X[-1,\ 1,-2,3]+1X[\ 0,-1,-1,3]$$

which is converted by *to_eps* to

$$1X[1,2,0,0]+1X[0,2,1,0]+1X[2,1,0,0]+2X[1,1,1,0]+1X[0,1,2,0]+\\ 1X[2,0,1,0]+1X[1,0,2,0]+1X[0,2,0,1]+2X[1,1,0,1]+2X[0,1,1,1]+\\ 1X[2,0,0,1]+2X[1,0,1,1]+1X[0,0,2,1]+1X[0,1,0,2]+1X[1,0,0,2]+\\ 1X[0,0,1,2].$$

5.8.2. The dominant character

Obviously characters get rather large pretty soon. An indication of the size of the module with highest weight λ can be obtained by computing $dim(\lambda)$, which returns the sum of the multiplicities of all the weights in that character, so for instance since $dim([1,2,3,14])$ equals 630, it should be clear that we do not want to print

the result of $char([1,2,3,14])$ here. However, because characters are invariant under the W action, there is no need to print the whole character in order to describe it completely: one representative weight from each W-orbit with its multiplicity would be sufficient. Since each orbit contains a unique dominant weight, that weight is an obvious candidate. The function *filter_dom* can be used to select only the terms with a dominant weight as exponent from a polynomial, so we have for instance $filter_dom(char([1,1,0,3])) = 1X[1,1,0,3]+2X[0,0,1,3]$; the value of $char([1,1,0,3])$ can still be reconstructed from this by applying the function *W_orbit*. However, a much more efficient way to obtain the dominant part of the character is to call $dom_char([1,1,0,3])$, which computes the same result without building the whole character as an intermediate value. In fact this function is more efficient even if eventually we do want to compute the whole character, so a slightly more efficient definition of *char* would have been

$char(\textbf{vec}\ lambda) = W_orbit(dom_char(lambda))$

Using *dom_char* it is feasible to obtain information about the characters of much larger modules than with *char*, for instance for the GL_4-module with highest weight $[1,2,3,14]$ mentioned above the result of $dom_char([1,2,3,14])$ is the polynomial

$$1X[1,2,3,14] + 1X[2,0,4,14] + 1X[1,3,1,14] + 2X[2,1,2,14] + 2X[0,1,4,14] + 2X[2,2,0,14] + 4X[0,2,2,14] + 3X[3,0,1,14] + 5X[1,0,3,14] + 4X[0,3,0,14] + 7X[1,1,1,14] + 9X[2,0,0,14] + 9X[0,0,2,14] + 11X[0,1,0,14]$$

(the reader may apply *to_eps* to this result and verify that all exponents then become partitions of 14, as they should). If one is only interested in the set of dominant weights occurring in the character, then one may call *dom_weights* instead of *dom_char*; as an example we may compute the exact number of terms in $char([1,2,3,14])$ by entering

$s = 0;$ **for** r **row** $dom_weights([1,2,3,14])$ **do** $s \mathrel{+}= W_orbit_size(r)$ **od**; s

and find that this number is 'only' 188.

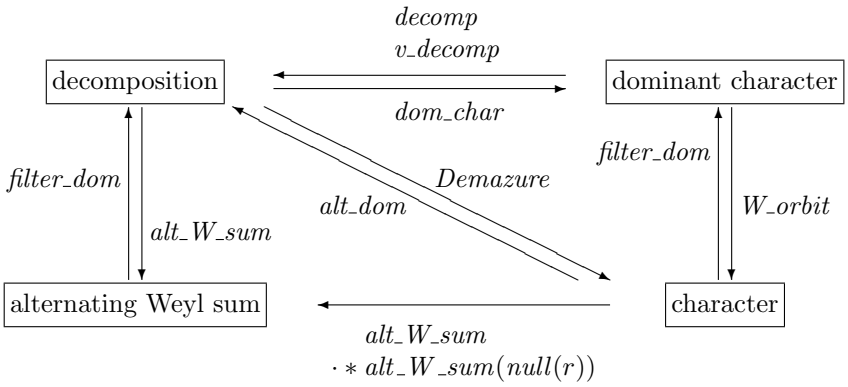
It is possible to compute characters, or their dominant parts, for reducible representations as well as for the (irreducible) highest weight modules that we discussed so far. For the built-in functions *Demazure* and *dom_char* (and indeed all other built in functions that accept a highest weight specifying an irreducible representation) it suffices to call them with a decomposition polynomial for the reducible module in place of the highest weight. For user defined functions like *char* this does not automatically work, but for either of the definitions of *char* an additional definition with '**vec**' replaced by '**pol**' would suffice to make it available for reducible representations as well as for irreducible ones. The functions *from_eps* and *to_eps* exemplify a simple way to linearly extend—from vectors to polynomials—functions which do more than merely call upon built in functions.

5.8.3. Decomposition

For the computation of characters of reducible representations the inverse operation is also of interest. Several operations are available as inverses to *char* and *dom_char*. For

reconstructing the decomposition polynomial of a representation from its dominant character one may use either *decomp* or *v_decomp*; the former will only succeed if all terms of the decomposition polynomial have positive coefficients (as they have for any actual representation), while the latter always succeeds (given enough time and memory), possibly returning a virtual decomposition polynomial. Apart from the wider applicability of *v_decomp*, it should also be noted that it uses an algorithm which is quite different from that of *decomp*. If we are given the full character of a representation, its decomposition polynomial can be most easily found by applying *alt_dom*: for any W-invariant polynomial p the dominantly supported polynomial $alt_dom(p)$ satisfies $char(alt_dom(p)) = p$.

Of course one may also choose to switch from dominant character polynomial to character polynomial or back (using *W_orbit* respectively *filter_dom*) before computing the decomposition polynomial. In fact it is possible to make an arbitrary sequence of transitions between the three forms of polynomial without loosing the essential information about the representation. We summarise the transition functions in a diagram, to which we add a fourth form of polynomial, the alternating Weyl sum, mainly because it adds a number of interesting commuting paths to the diagram.



For instance, *Weyl's character formula* essentially states that starting with a decomposition polynomial one obtains the same result by computing the alternating Weyl sum directly as by computing the character first and then multiplying with the alternating Weyl sum of the zero weight (the lower label of the bottom arrow; this second method is much less efficient). Also one sees that the alternating Weyl sum of decomposition polynomial as the same as that of the character polynomial. A similar property holds for the functions *alt_dom* and *Demazure*, so that the diagram could be extended by circular arrows at the corners "decomposition polynomial" and "character polynomial", respectively labeled *alt_dom* and *Demazure* (more such 'identity arrows' could in fact be added, but they would not be very illuminating). Finally note that the application of *branch* mentioned in Section 5.8.1 could be added as an alternative to *Demazure* along the diagonal arrow, and similarly *collect* is an alternative for *alt_dom* along the opposite arrow.

It is not possible (and it would make little sense) to apply *decomp* to a single weight; however it is possible to apply *v_decomp* to single weights. In fact, somewhat surprisingly, *v_decomp* can be effectively applied to weights λ which are so large that even computing $dom_char(\lambda)$ would cost excessive time and space, provided only that the Weyl group has a tractable size. For instance, still assuming the default group A_3T_1 from earlier examples, we have for $\lambda = ([34, 19, 52, 68])$ that $dim(\lambda) = 1340608500$, and $dom_char(\lambda)$ has 26860 terms, yet calling $v_decomp([34, 19, 52, 68])$ promptly returns

$$\begin{aligned}
&1X[34,19,52,68] - 1X[35,17,53,68] - 1X[34,20,50,68] - 1X[32,20,52,68] + \\
&1X[32,21,50,68] + 1X[36,16,52,68] + 1X[35,19,49,68] + 1X[34,16,54,68] + \\
&1X[31,19,53,68] - 1X[36,17,50,68] - 1X[32,17,54,68] - 1X[35,15,53,68] + \\
&-1X[31,21,49,68] - 1X[34,19,48,68] - 1X[30,19,52,68] + 1X[35,17,49,68] + \\
&1X[32,20,48,68] + 1X[31,17,53,68] + 1X[30,20,50,68] + 1X[34,15,52,68] + \\
&-1X[34,16,50,68] - 1X[32,16,52,68] - 1X[31,19,49,68] + 1X[32,17,50,68]
\end{aligned}$$

An explanation for this phenomenon can be deduced from the diagram above, and from the algorithm used to implement *v_decomp* (the algorithm of *decomp* would not produce this simple answer using a reasonable amount of time and space).

5.9. Checks

Numerous checks are possible to verify the consistency between results produced by different functions. We have already mentioned many of them in Chapter 4, and in the previous examples (several such checks are implicitly contained in the diagram above). In addition to those, we give below a number of other checks that can be made.

5.9.1. Checking Kazhdan-Lusztig polynomials

One may use the R-polynomials for a test of the Kazhdan-Lusztig polynomials. We should have for any $x, w \in W$:

$$X^{l(w)-l(x)}\overline{P_{x,w}} - P_{x,w} = \sum_{x<y\leq w} R_{x,y}P_{y,w}$$

where the bar indicates a sign change of all the exponents. The following function performs the test; it should always return 1 for $x, w \in W$.

```
test_KL(vec x, w) =
{  loc result = poly_null(1);
   loc l = length(w); loc lx = length(x); loc dl = l − lx; loc level = [w];
   while l > lx
   do l += −1; loc nl = null(0, l);
      for y row level
      do result += R_poly(x, y) ∗ KL_poly(y, w);
         nl = unique(nl^canonical(Bruhat_desc(x, y)))
      od; level = nl
   od; result == { loc pxw = KL_poly(x, w); X dl ∗ (pxw ∗ [[−1]]) − pxw }
}
```

This function also exemplifies a useful application of a local block as an operand in a (relational) formula.

5.9.2. Dimension checks

A dimension check can be made for the result of *dom_char*; we should have

$$\sum_{\mu\in\Lambda^+(T),\mu\prec\lambda} dom_char(\lambda,\mu) * W_orbit_size(\mu) = dim(\lambda).$$

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

```
check_dim(vec lambda) =  loc p = dom_char(lambda); loc d = 0;
   for i = 1 to length(p) do d += coef(p, i) ∗ W_orbit_size(expon(p, i)) od;
   d == dim(lambda)
```

We may similarly check for *branch* that the dimension of the resulting sum of h-modules equals that of the original g-module V_λ. In other words, we should have

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

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). The test can be performed with the following function

```
chk_branch(vec wt; grp h; mat m) =  dim(branch(wt, h, m), h) == dim(wt)
```

Similar dimension checks can be made for most representation theoretic operations. For instance, the dimension of the tensor product of representations should be the product of their dimensions, and the (virtual) dimension of the n-th Adams operator applied to a representation should be the same as the original dimension of that representation (because the corresponding character polynomials have the same number of terms, with the same multiplicities). The dimension of plethysms for a fixed partition is a polynomial expression (with rational coefficients) in the dimension of the starting representation. The following functions compute this polynomial and use it to predict the dimension of plethysms.

```
l(vec mu) =  # number of non-zero parts # loc n = size(mu);
    for i = 1 to n do if ! mu[i] then return i - 1 fi od;  n
pleth_pol(vec lambda) =
{   loc p = 0X0;
    for mu in partitions(sum(lambda))
    do p += sym_char(lambda, mu) * class_ord(mu)X l(mu) od;
    p
}
pleth_dim(vec lambda; int orig_dim) =
    eval_pol(pleth_pol(lambda), orig_dim)/fac(sum(lambda))
```

5.9.3. Checks using characters

Using character polynomials even more precise checks are possible than using dimensions. For instance we may write

```
char(pol p; grp g) = W_orbit(dom_char(p, g), g)
check_tensor(vec x, y) =  char(tensor(x, y)) == char(x) * char(y)
check_branch(vec x; grp h; mat m) =
    char(branch(x, h, m), h) == char(x) * m
```

The practical applicability of these tests is limited to small cases, however, due to the size of the character polynomials.

5.9.4. The functions *sym_tensor*, *alt_tensor*, and *plethysm*

It has already been mentioned that *sym_tensor* and *alt_tensor* should coincide with special cases of *plethysm*. Here is a way to check that the second tensor power of a module decomposes into a symmetric and alternating part:

```
check_sq(vec wt) = alt_tensor(2, wt) + sym_tensor(2, wt) == p_tensor(2, wt)
```

Such a simple relation does not hold for higher tensor powers, since one needs all plethysms to decompose the tensor power, moreover $plethysm(\lambda, \mu)$ occurs a number of times in $p_tensor(n, \mu)$ with $n = |\lambda|$. The number of times it occurs is the dimension of the representation S_n corresponding to λ. This dimension is equal to the value of the symmetric group character χ_λ at the identity element, which can be computed as $sym_char(\lambda, all_one(sum(\lambda)))$, but it is also equal to the number of Young tableaux of shape λ, so it is more efficiently computed as $n_tabl(\lambda)$. We can now set up the following test.

```
chk_p_tensor(int n; vec wt) =  loc d = poly_null(size(wt));
    for lambda row partitions(n)
    do d += n_tabl(lambda) * plethysm(lambda, wt) od;
    d == p_tensor(n, wt)
```

5.9.5. Intrinsic tests

In the course of the computation of certain built-in functions a number of tests are automatically performed to check the validity of assumptions about the results obtained from subsidiary calls. We mention here two such tests which have been useful

in detecting subtle coding errors during the development of LiE. (The tests are still present in the production versions of LiE, and although of course they should "stay forever silent", it is still conceivable that they confront an unsuspecting user with a harsh error message. Users getting such error messages, apparently not indicating a transgression on their part, are requested to report this to the developers of LiE.)

For each orbit of weights generated by *W_orbit* the size is first predicted by *W_orbit_size* and a corresponding amount of memory is allocated. Then the actual orbit is generated using the same procedure as used by *tensor*, *branch* and other functions. If during this process the allocated memory runs out, or if at the end it is not completely filled, then an error will be reported. In this way both the orbit generation routines and the routines involved in *W_orbit_size* (for instance *Cartan_type*) are tested. A single call of *W_orbit* with a polynomial argument may perform this test for orbits of many different sizes, and therefore provides a rather thorough check of the used routines.

The formulae used for the routines *sym_tensor*, *alt_tensor*, and *plethysm* involve a summation of polynomials which is to be divided by an integer, where the resulting polynomial should have integer coefficients. It is checked that the divisions involved leave no remainder. Since *plethysm* in particular uses many of the main routines of LiE, and since the denominator occurring in its formula can be substantial, the successful completion of a call of *plethysm* with a moderately large partition provides in itself some confirmation of the correctness of the computation.

5.10. Branching

The function *branch* is very general, and many other functions can be considered to be special cases of it, for instance *spectrum* (branching to a 1-dimensional torus), *tensor* (branching from $g \times g$ to g), *plethysm* (branching a representation of some GL_n to g via the morphism $g \to GL_n$ determined by a representation of g), and if we were to allow an irreducible representation to branch to a *virtual* representation, even *Adams* (branching with a multiple of the identity as restriction matrix). Being so general, the algorithm used for *branch* is not always the most efficient one in a specific situation. The following subsections discuss alternative ways of computing *branch*; except for the first of these they deal with improvements for specific cases.

5.10.1. Branching from composite groups

Many of the examples given in the earlier version of this manual have become obsolete since their functionality has now been built into LiE; this is also the case with the following example which shows how branching in composite groups can be expressed in terms of branching in simple groups. We have retained the example because it is a somewhat larger function than most other examples, and as a documentation of the algorithm used by the built-in function *branch*. The example has been adapted to the conventions of the current version of LiE, for instance by using the polynomial data type; it reflects rather precisely the algorithm which has been built in. The function has been given a different name, since using *branch* with the same parameter types

would conflict with the built-in function.

```
branch_comp(vec wt; grp h; mat m; grp g) =
{  loc c = n_cols(m);
   if Lie_rank(h) != c || Lie_rank(g) != n_rows(m)
      then error("wrong size restriction matrix") fi;
   loc r = toral_dim(g); loc wk = null(r); loc mk = null(r, c);
   loc i = ss_rank(g);
   for j = 1 to r do mk[j] = m[i + j]; wk[j] = wt[i + j] od;
   loc res = alt_dom(wk * mk, h);  # central torus part, ensure dominant #
   i = 0;
   for k = 1 to n_comp(g)
   do r = Lie_rank(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 += r
   od;
   res
}
branch_comp(pol p; grp h; mat m; grp g) =
{  loc s = coef(p, 1) * branch_comp(expon(p, 1), h, m, g);
   for i = 2 to length(p)
   do s += coef(p, i) * branch_comp(expon(p, i), h, m, g) od;
   s
}
```

5.10.2. Branching from F_4 to B_4

Recall that the algorithm of *branch* generates the full character, then applies the restriction matrix, filters the dominant part and decomposes. Generating the full (not just the dominant) character is necessary since non-dominant weights may become dominant (for the subgroup) upon applying the restriction matrix (but conversely dominant weights should always stay dominant, if the restriction matrix is constructed for compatible systems of fundamental weights; this fact is used by *collect*). We cannot improve on this in the general case, as can be seen by considering branching to the maximal torus of g: the result should then in fact be equal to the full character (filtering and decomposition are null operations in this case). However, in specific cases we may be able to see beforehand that only a few Weyl chambers (images under some $w \in W$ of the dominant chamber) will map into the dominant chamber of the subgroup under applying the restriction matrix; in such cases it is only necessary to generate that part of the W-orbit which lies in these chambers.

As a concrete example let us study branching from F_4 to B_4. We start with some considerations for finding the restriction matrix, The root system of F_4 contains 24 short roots and 24 long roots, each of which subsets form a subsystem of type D_4; the set of long roots is a closed subsystem, but the closure of the short roots is the full root system of F_4. To check these statements we compute as follows (we do not

print the results; the reader should use LiE to obtain them).

setdefault F_4; $short = null(0,4)$; $long = short$;
for r **row** pos_roots **do if** $norm(r) == 2$ **then** $short \mathrel{+}= r$ **else** $long \mathrel{+}= r$ **fi od**
$print(n_rows(short))$; $Cartan_type(short)$
$print(n_rows(long))$; $Cartan_type(long)$
$short = fundam(short)$; $Cartan_type(closure(short))$
$long = fundam(long)$; $Cartan_type(closure(long))$

Adjoining any one short root to the set of long roots and then forming the closure will result in a subsystem of type B_4, which contains 8 short roots and of course all 24 long roots (consequently there are 3 such subsystems, which are conjugate under the Weyl group of F_4). We choose such a system and compute its basis of fundamental roots

$b4_roots = closure(long + short[1])$; $print(b4_roots)$; $Cartan_type(b4_roots)$

The full set of positive roots of this subsystem can be obtained by mapping the positive roots of B_4 into the F_4 system: $pos_roots(B_4) * b4_roots$. We can now obtain the restriction matrix for the fundamental Lie subgroup of type B_4 with this subsystem as set of roots:

$m = res_mat(b4_roots)$; m

Now that we know the restriction matrix, we would like to know which Weyl chambers for F_4 are dominant for B_4. We expect that there are three such chambers, since $W_order(F_4)/W_order(B_4)$ equals 3. In terms of our restriction matrix we can state the problem as: for which $w \in W$ are the entries of the matrix $W_action(w) * m$ all positive (meaning that the images of all fundamental weights are dominant in B_4)? The answer can be found with a little experimentation (obviously the identity is one of the requested w's, and the other two will not be far removed from it), or by the following reasoning. Since the first three fundamental roots of F_4 occur in $b4_roots$ (which is a bit of a coincidence; it is not true for the other two conjugate subgroups) the chambers obtained applying any of the first three fundamental reflections to the dominant Weyl chamber of F_4 will certainly not be dominant for B_4. Since one of the chambers bordering the dominant one must be included, the fourth fundamental reflection (encoded as [4]) must be one of the desired w's. The third chamber must be a neighbor of the one for $w = [4]$, whence its Weyl group element is obtained by multiplying [4] to the *left* by a fundamental reflection, and moreover it must be right reduced with respect to the subset $S = \{1, 2, 3\}$ of fundamental reflections; therefore the only possibility is $w = [3, 4]$. So for branching from F_4 to this instance of a subgroup of type B_4, one may replace the full W-orbits by the images under the action of $w \in \{[\,], [4], [3, 4]\}$ (for the other two instances of such subgroups the corresponding sets are $\{[\,], [3], [4, 3]\}$ and $\{[\,], [3], [4]\}$ respectively).

Our next step is to generate these 3-chamber "orbits". It is convenient to do it for the whole dominant character at once, in analogy to *W_orbit* applied to a polynomial. We should not simply compute $p + W_action(p, [4]) + W_action(p, [3, 4])$, since that would incorrectly duplicate weights at the 'walls' separating our chambers.

We therefore need to be able to remove from p the terms whose exponents lie on a given wall of the dominant chamber:

shave(**int** i; **pol** p) = **loc** $v = id(n_vars(p))[i]$; *filter_dom*$(X(-v) * p) * X\ v$

(note how a unit vector is obtained as a row of the identity matrix). Then the "orbit" of a dominantly supported polynomial over the indicated three chambers is

spread(**pol** p) $= p +$ *W_action*$(shave(4, p), [4]) +$ *W_action*$(shave(3, p), [3, 4])$

and we can now write our special branching function as

branch_f4_b4(**pol** p) = *decomp*$(spread(dom_char(p)) * m, B_4)$

Experimentation with this function shows that indeed it delivers the same results as a corresponding call of branch, and it is marginally faster (about 10%). Apparently generating and discarding many weights in each W-orbit is not a limiting factor for the speed of *branch* (for this case); the amount of time spent by the interpreter executing our little program cannot be of much influence since it only involves a limited number of instructions.

5.10.3. Branching from G_2 to A_2 using a rational function

The polynomials computed by LiE's representation theoretic functions usually cannot be expressed by simple closed formulae in terms of the input data. However, the method of generating functions often allows us to construct closed expressions for power series in newly introduced indeterminates, containing the polynomials computed by LiE as the coefficient of some power of the new indeterminates. Although these power series are infinite structures and can therefore not be computed fully in explicit form, the closed expressions (which are in fact *rational functions* in the old and new indeterminates) often enable an efficient algorithm for computing the (finite) polynomials we are after.

As an example of this technique consider branching from G_2 to its fundamental Lie subgroup of type A_2 (whose root system is the closed subsystem of G_2 of the long roots). In analogy to the examples already given, the reader should have no difficulty to verify that the restriction matrix for this branching is $\binom{0\,1}{1\,1}$ (the columns might be interchanged depending on the chosen ordering, because of the symmetry of the Dynkin diagram of type A_2). If m is this matrix, then we are considering an alternative way to compute $branch(\lambda, A_2, m, G_2)$. Let P_λ denote this polynomial for arbitrary dominant (for G_2) weights λ; it is an ordinary polynomial (since all exponents are dominant for A_2) in two indeterminates. To form the power series we want to discuss, we introduce a pair of new indeterminates; to indicate monomials in these indeterminates we use the symbol Y (as opposed to X which is used in the P_λ) with a vector of size two as exponent (in fact a weight for G_2). The power series then is

$$P = \sum_{\lambda \in \Lambda^+(G_2)} P_\lambda Y^\lambda$$

It has been shown in [CoRui] that this power series can be given by the following

rational function:

$$P = \frac{1 - X^{[1,1]}Y^{[1,1]}}{(1 - X^{[1,0]}Y^{[1,0]})(1 - X^{[0,1]}Y^{[1,0]})(1 - X^{[0,0]}Y^{[1,0]}) * \\ (1 - X^{[1,0]}Y^{[0,1]})(1 - X^{[0,1]}Y^{[0,1]})(1 - X^{[1,1]}Y^{[0,1]})}$$

(in the notation of [CoRui], u corresponds to $X^{[1,0]}$, v to $X^{[0,1]}$, y to $Y^{[1,0]}$, and x to $Y^{[0,1]}$). To deduce from this rational function its coefficient P_λ of Y^λ one could use general power series expansion techniques, but given the special form of the function this is not necessary. Let Q be the rational function obtained by replacing the numerator of P by 1, and Q_λ its coefficient of Y^λ, then $P_\lambda = Q_\lambda - X^{[1,1]}Q_{\lambda-[1,1]}$ if $\lambda - [1,1]$ is dominant, and otherwise $P_\lambda = Q_\lambda$. For the computation of Q_λ note that the denominator of P factors as product of a term A not involving $Y^{[0,1]}$, and a term B not involving $Y^{[1,0]}$; consequently $Q_{[a,b]}$ is the product of the coefficient R_a of $Y^{[a,0]}$ in $\frac{1}{A}$ and the coefficient S_b of $Y^{[0,b]}$ in $\frac{1}{B}$. Furthermore both $\frac{1}{A}$ and $\frac{1}{B}$ are a product of factors of the form $(1 - X^\mu Y^\nu)^{-1}$ for dominant weights μ, ν (this is quite typical for this kind of generating functions); such a factor can be expanded to a geometric series $\sum_{n=0}^{\infty} X^{n\mu}Y^{n\nu}$. It is not difficult to find combinatorial descriptions of the coefficients R_a and S_b occurring in the product of these series, they are respectively

$$R_a = \sum_{\substack{i,j,k \in \mathbf{N} \\ i+j+k=a}} X^{[i,j]} \qquad \text{and} \qquad S_b = \sum_{\substack{i,j,k \in \mathbf{N} \\ i+j+k=b}} X^{[i+k,j+k]}.$$

These formulae can be straightforwardly translated into functions for LiE. Since the generation of the very regular polynomials R_a and S_b is likely to consume a large portion of the time, we use a little trick to avoid two nested loops.

```
r(int a) = loc p = X[0,0]; loc sum = p;
    for i = 1 to a do p = X[1,0] * p + X[0,i]; sum += p od; sum
s(int b) = loc p = X[b,b]; loc sum = p;
    for i = b - 1 downto 0 do p = X[-1,0] * p + X[b,i]; sum += p od; sum
q(vec lambda) = r(lambda[1]) * s(lambda[2])
p(vec lambda) =
    if lambda[1] > 0 && lambda[2] > 0
    then q(lambda) - X[1,1] * q(lambda - [1,1]) else q(lambda)
    fi
```

Experimentation shows that this function p is considerably more efficient than *branch*: speedup factors exceeding 20 have been found for weights of the order of magnitude of $[13, 7]$. Clearly the approach of using rational generating functions is an attractive one, in particular since the existence of appropriate rational functions has been proved for all cases of branching, tensor products and plethysms. The main problem with this approach is the determinatation of these rational functions: in the literature several other generating functions for small rank cases are known (cf. [McKPa]), but there seems to be no practical algorithmic way to compute the generating function for an

arbitrary problem in the mentioned categories. (An additional problem might be that other rational functions could turn out to be much more difficult to handle than the one in this example.)

5.11. Overflow

Due to the fact that matrix and vector entries are stored as machine size integers, rather than as arbitrary size integers, and that arithmetic overflow is not tested for during vector and matrix arithmetic, one should not completely trust vector and matrix operations leading to big integer entries. In the example below, we found an 'orbit' of length 34, apparently due to the computer's arithmetic modulo 2^{32}.

$orbit([1,0,0,0], reflection([1,1,1,0]) + reflection([1,1,1,1]))$

Incidentally, the real error that led to this example was using the operator '+' instead of '^', implying addition of matrices rather than the intended concatenation.

5.12. Branching to Levi subgroups

In order to be able to compute branching to a subgroup, one needs to specify the subgroup within the containing group; in LiE this is done by means of the type of the subgroup together with a restriction matrix. If one has another sort of description of the subgroup, or if one is looking for *all* subgroups with certain properties, then extra work has to be done to determine the type and the restriction matrix. Whether this can be done in an algorithmic way depends on the details of the situation; generally speaking one must depend on classification in the form of tables (such as provided by *max_sub* and instances of *res_mat*) for subgroups of lower Lie rank than the full group, while for groups of equal rank (the fundamental Lie subgroups) algorithms dealing with root systems can be applied. For one class of subgroups the whole computation can be done completely mechanically, namely for the Levi subgroups, i.e., those fundamental Lie subgroups of with a system of fundamental roots that is a subset of the set of fundamental roots of the whole group. The most interesting of these are the maximal (proper) Levi sugroups, for which only one fundamental root is removed (the other Levi subgroups can clearly be reached by repeating this process). We give here some functions that compute the ingredients necessary for branching to them.

$Levi_mat(\mathbf{int}\ i) = fundam(id(Lie_rank) - i)$ # remove i-th row and reorder #
$Levi_type(\mathbf{int}\ i) = Cartan_type(Levi_mat(i))$
$Levi_diagram(\mathbf{int}\ i) = diagram(Levi_type(i))$
$Levi_res_mat(\mathbf{int}\ i) = res_mat(Levi_mat(i))$
$Levi_branch(\mathbf{vec}\ v;\ \mathbf{int}\ i) =$
  $\mathbf{loc}\ m = Levi_mat(i);\ branch(v, Cartan_type(m), res_mat(m))$

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

LIE Manual

Chapter 6. SYNTAX

In this chapter the complete formal syntax accepted by the interpreter is given for reference. It is as a context free grammar, using the well known BNF formalism. Literally represented symbols are given in `typewriter type`, and every rule is terminated by a period. The grammar has a few ambiguities, but these only involve expressions with multiple operators, and they are resolved by the priorities assigned in Section 2.3 to operators, and the rule that all operators are left-associative. Note however that the fact that relational and Boolean operators have lower priorities than the arithmetic ones is not essential, since it explicitly follows from the given grammar.

The majority of the rules presented are derived in an almost mechanical way from the syntax used to generate the parser of LIE (using the program `yacc`); therefore we are quite confident that, apart from the mentioned ambiguities, these rules give truthful descriptions of the language accepted by LIE (even if they should have some strange consequences). The lexical part of the grammar is given at the end (starting with the rule for ⟨ empty ⟩), and describes the operation of the lexical analyser; in this part white space between the various components of a production is not allowed. This part is somewhat more informal, and some lexical matters are purposely not indicated, like under which circumstances an end of line is taken to terminate a command (for this see Section 2.1.1); a grammar is not an appropriate means to specify such details.

The syntax includes a few cases such as '`exec` ⟨ tail⟩' that are not described anywhere in this manual, since they are of little or no interest to the average user. The special identifiers `read`, ..., `exec` that start various alternatives of ⟨ command ⟩ are recognised as keywords *only* at the beginning of a command; in other positions they are treated as ordinary identifiers, and they may be used for instance as local variables.

⟨ command ⟩ ::= ⟨ series ⟩ | ⟨ function definition ⟩ | `read` ⟨ filename option ⟩
| `edit` ⟨ filename option ⟩ | `write` ⟨ filename ⟩ | `monfil` ⟨ filename option ⟩
| `maxobjects` | `maxobjects` ⟨ number ⟩ | `maxnodes` | `maxnodes` ⟨ number ⟩
| `listvars` | `listfuns` | `listops` | ⟨ help ⟩ ⟨ subject ⟩ `>` ⟨ filename ⟩
| ⟨ help ⟩ ⟨ subject ⟩ `>>` ⟨ filename ⟩ | ⟨ help ⟩ ⟨ subject ⟩ | `learn` ⟨ tail option ⟩
| `:` ⟨ tail ⟩ | ⟨ quit ⟩ | `type` ⟨ arithmetic expr ⟩ | `exec` ⟨ tail ⟩ | ⟨ empty ⟩ .

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

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

⟨ statement ⟩ ::= ⟨ assignment ⟩ | ⟨ expression ⟩ | `return` ⟨ expression ⟩
| `break` ⟨ expression ⟩ | `return` | `break` | `setdefault`
| `setdefault` ⟨ expression ⟩ | ⟨ on ⟩ ⟨ identifier ⟩ | `off` ⟨ identifier ⟩ | ⟨ on ⟩
| `off` | `savestate` | `restorestate` .

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

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

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

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

⟨ arithmetic expr ⟩ ::= ⟨ arithmetic expr ⟩ ⟨ operator ⟩ ⟨ arithmetic expr ⟩
| ⟨ monadic expr ⟩ .

⟨ monadic expr ⟩ ::= ⟨ monadic operator ⟩ ⟨ monadic expr ⟩ | ⟨ secondary ⟩ .

⟨ secondary ⟩ ::= ⟨ selection ⟩ | ⟨ primary ⟩ .

⟨ selection ⟩ ::= ⟨ secondary ⟩ `[` ⟨ expression ⟩ `]`
| ⟨ secondary ⟩ `[` ⟨ expression ⟩ `,` ⟨ expression ⟩ `]` | ⟨ secondary ⟩ `|` ⟨ primary ⟩ .

⟨ primary ⟩ ::= ⟨ variable ⟩ | ⟨ number ⟩ | ⟨ string ⟩ | ⟨ group ⟩
| ⟨ identifier ⟩ `(` ⟨ list option ⟩ `)` | `make` `(` ⟨ variable ⟩ `,` ⟨ arithmetic expr ⟩ `)`
| `make` `(` ⟨ variable ⟩ `,` ⟨ arithmetic expr ⟩ `,` ⟨ arithmetic expr ⟩ `)`
| ⟨ apply ⟩ `(` ⟨ variable ⟩ `,` ⟨ arithmetic expr ⟩ `,` ⟨ arithmetic expr ⟩ `)`
| `[` ⟨ list option ⟩ `]` | `(` ⟨ arithmetic expr ⟩ `)` | ⟨ block ⟩
| ⟨ conditional clause ⟩ | ⟨ loop clause ⟩ .

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

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

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

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

⟨ loop clause ⟩ ::= `while` ⟨ expression ⟩ `do` ⟨ series ⟩ `od`
| `for` ⟨ identifier ⟩ `=` ⟨ arithmetic expr ⟩ `to` ⟨ arithmetic expr ⟩ `do` ⟨ series ⟩ `od`
| `for` ⟨ identifier ⟩ `=` ⟨ arithmetic expr ⟩ `downto` ⟨ arithmetic expr ⟩ `do` ⟨ series ⟩ `od`
| `for` ⟨ identifier ⟩ `in` ⟨ arithmetic expr ⟩ `do` ⟨ series ⟩ `od`
| `for` ⟨ identifier ⟩ `row` ⟨ arithmetic expr ⟩ `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 ⟩ ::= ⟨ empty ⟩ | ⟨ list ⟩ .

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

⟨ empty ⟩ ::= .

⟨ help ⟩ ::= `help` | `?` .

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

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

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

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

⟨ upper case letter ⟩ ::= `A` | `B` | ... | `Z` .

⟨ letter ⟩ ::= ⟨ lower case letter ⟩ | `_` | ⟨ upper case letter ⟩ .

⟨ sysident ⟩ ::= `$` | `$` ⟨ number ⟩ .

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

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

⟨ monadic operator ⟩ ::= `+` | `-` | `*` | ⟨ X ⟩ .

⟨ X ⟩ ::= `X` | `Y` .

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

⟨ Boolean operator ⟩ ::= `&&` | `||` .

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

⟨ tail ⟩ ::= { non-empty sequence of any characters but newline } .

⟨ string ⟩ ::= ⟨ basic string ⟩ | ⟨ string ⟩ { white space } ⟨ basic string ⟩ .

⟨ basic string ⟩ ::=
`"` { sequence of characters unequal to “`"`” and newline, or “`"""`” } `"` .

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

⟨ filename ⟩ ::= { non-empty sequence of non-whitespace characters } .

⟨ subject ⟩ ::= ⟨ empty ⟩
| { non-empty sequence of any non-whitespace characters but “`>`” } .

⟨ quit ⟩ ::= `quit` | `exit` | `@` .

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

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

LiE Manual

Chapter 7. REFERENCES

We give a list of the main books and papers that have been of use and/or influence to us while preparing LiE, and which 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] C. M. Andersen, "Clebsch-Gordan series for symmetrized tensor products", *J. Math. Phys.* **8**, (1977), 988–997.

[BeKo] R. E. Beck & B. Kolman (eds.), *Computers in Nonassociative Rings and Algebras*, Acad. Press, New York, 1977.

[Bour4] N. Bourbaki, *Groupes et algèbres de Lie, Chap 4, 5, et 6*, Hermann, Paris, 1968.

[Bour7] N. Bourbaki, *Groupes et algèbres de Lie, Chap 7 et 8*, Hermann, Paris, 1975.

[BMP] 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.

[BrCo] A. E. Brouwer & A. M. Cohen, "Computation of some parameters of Lie geometries", *Annals of Discrete Math.* **26**, (1985), 1–48.

[CoGr] 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]" (P. Fong, ed.), Amer. Math. Soc., Providence, 1987.

[CoRui] A. M. Cohen & G. C. M. Ruitenburg, "Generating functions and Lie groups", *CWI tract* **84**, (1991), 19–28.

[Deodh] V. V. Deodhar, "Some Characterisations of Bruhat Ordering on a Coxeter group and determination of the Relative Möbius Function", *Inventiones Math.* **39**, (1977), 178–198.

[Hum1] Humphreys, J. E., *Introduction to Lie algebras and representation theory*, Springer, New York, 1972.

[Hum2] J. E. Humphreys, *Reflection groups and Coxeter groups*, Cambridge University Press, 1990.

[Jac] N. Jacobson, *Lie algebras*, Wiley & Sons, New York, 1962.

[JaKe] G. James & A. Kerber, *The Representation Theory of the Symmetric Group*, Addison-Wesley, Reading MA, 1981.

[KaLu] D. Kazhdan and G. Lusztig, "Representations of Coxeter groups and Hecke algebras", *Inventiones Math.* **53**, (1979), 165–184.

[Knuth] D. E. Knuth, *The art of Computer programming, Vol. III: Sorting and searching*, Addison-Wesley, Reading MA, 1975.

[Kruse] M. I. Krusemeyer, "Determining multiplicities of dominant weights in irreducible Lie algebra representations, using a computer", *BIT* **11**, (1971), 310–316.

[Litt] P. Littelmann, "A generalisation of the Littlewood-Richardson rule", *Journal of Algebra* **130, #2**, (1990), 328–368.

[Macd] I. G. Macdonald, *Symmetric functions and Hall polynomials*, Clarendon Press, Oxford, 1979.

[McKPa] 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.

[MoPa] R. V. Moody & J. Patera, "Characters of elements of finite order in Lie groups", *SIAM J. Alg. Discr. Meth.* **5**, (1984), 359–383.

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

[Spri] T. A. Springer, *Linear algebraic groups*, Birkhäuser, Basel, 1981.

[Shi] Shi J.-Y., *The Kazhdan-Lusztig Cells in Certain Affine Weyl Groups*, Springer Verlag, Berlin, 1986.

[Tits] 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, with the pages on which they are referred to. When a term coincides with the name of a function, references to both the term and the function are listed after the function name. Underlined numbers refer either to the listings of operators and functions (Chapter 2 and Chapter 4), to the listings of terms(Chapter 3), or to occurrences in the syntax(Chapter 6).

$(\cdot,\cdot)$ 61
$\langle \cdot, \alpha \rangle$ 41, 59
Λ 40
+ 10, 15, 106, 107
+= 22, 106
− 10, 16, 106, 107
∗ 10, 16, 107
/ 16, 107
% 17, 107
^ 17, 107
= 21, 106
< 17, 107
<= 17, 107
> 17, 105, 107
>= 17, 107
! 17, 18, 106
!= 17, 107
== 17, 107
&& 18, 107
|| 18, 107
| 106
`>` 4, 10, 32
\ 10
`?` 10, 31, 107
`@` 4, 107
`#` 11
`$` 11, 107
$\prec$ 41
ρ 78
abort 4
abs 19
Adams 67
Adams operator 52, 53
adjoint 67
adjoint representation 40, 43, 53
all_one 19
alternating Weyl sum 53, 55
alt_dom 68
alt_tensor 67
alt_W_sum 68
`bigint` 32
block 22
block_mat 19
branch 68, 69
branching 52, 54, 56
break 24, 106
Bruhat descendent 45
Bruhat order 45, 46, 47, 48
Bruhat_desc 61
Bruhat_leq 61
canonical 62
canonical Weyl word 46, 47
Cartan 59
Cartan matrix 38, 41

Cartan product 59
Cartan type 41
Cartan_type 59
center 58
central torus 38, 39
centraliser 89
centr_type 59, 60
cent_roots 59
character 50, 54, 56, 58
character polynomial 40, 50, 52, 54, 55
classical groups 36
class_ord 65
clauses 21
closed subsystem 38, 41, 42
closure 60
coef 20
coefficient 13
collect 69
n_comp 20
contragr 70
contragredient representation 54
coset 41, 46, 82
Coxeter group 45, 46, 47
Coxeter matrix 46, 79
decomp 70
decomposition 58
decomposition polynomial 51, 55
default group 25
Demazure 70
Demazure operator 55
det_Cartan 60
diag 19
diagram 38, 58
dim 58, 70, 71
direct sum 55
distinguished coset representative 46, 82
do 23, 106
dominant 44, 58, 62
dominant character polynomial 52, 55
dominant weight 46
dom_char 71
dom_weights 60
downto 23, 106
`edit` 30, 105
else 22, 106
error 20
exceptional groups 37
exec 105
`exit` 4, 107
expon 13, 20
exponent 13, 20
exponents 46, 62
factor 19
fi 22, 106
filter_dom 62
for 23, 24, 106
frequency 51
from_part 65, 66
fundam 60
fundamental domain 44
fundamental Lie subgroup . 38, 42, 54
fundamental reflection . . . 42, 44, 48
fundamental root 41, 42, 43
fundamental weight 41, 42, 44, 46, 56
garbage collector 31
`gc` 32
gcol 20, 31
General Linear group 38, 39
grp 107
`help` 10, 107
higher than 41
highest root 42
highest weight 3, 42, 55, 56
highest weight module 55
high_root 61
iapply 29, 107
id 19
identifier 10
if 22, 106
in 23, 106
`initfile` 31
inner product 61
inprod 61
int 11, 57, 107
ints 11, 57, 107
irreducible 51
irreducible representation 55
i_Cartan 61

Kazhdan-Lusztig polynomial . . 45, 47
KL_poly 62
Laurent polynomial 7, 13
`learn` 31, 105
length 20, 45, 46, 47, 62
level 27
Levi subgroup 42
lexicographic ordering 13
Lie algebra 38, 53
Lie group 39
Lie rank 38, 39
lies under 41
Lie_code 58
Lie_group 58
Lie_rank 58
lin 57
`listfuns` 10, 105
`listops` 10, 105
`listvars` 10, 105
Littlewood-Richardson rule 71
loc 21, 106
local variable 27
Longest element 47
long_word 62
`lprint` 32
lr_reduce 63
LR_tensor 71
l_reduce 62
make 28, 106
mapply 29, 107
mat 12, 107
matrix 35
mat_vec 19
maximal torus 38, 39
`maxnodes` 33, 105
`maxobjects` 33, 105
module 51, 56
`monfil` 9, 32, 105
`monitor` 32
multiplicity 51
next_part 66
next_perm 66
next_tabl 66
norm 61
null 19
n_cols 19
n_pos_roots 61
n_rows 19
n_tabl 66
n_vars 20
od 23, 106
`off` 21, 32, 106
`on` 21, 32, 106
one parameter subgroup 42, 43
orbit 47, 63
orbit matrix 47
partitions 48, 50, 51, 56, 57, 66
partition coordinates 49, 50, 56
parts 48
permutation 57
plethysm 50, 56, 72
pol 13, 107
polynomial 36, 58
poly_null 20
poly_one 20
positive root 43, 55
pos_roots 61
print 20
print_tab 66
`prompt` 4, 10, 32
p_tensor 72
`quit` 4, 107
rational function 102
`read` 30, 105
real Lie groups 37
reduce 63
reduced expression 45
reduced Weyl word 47
reductive 39
reductive group 39
reflection 47, 63
representation 53, 56
`restorestate` 21, 33, 106
restriction matrix 52, 54, 56
res_mat 72, 73
return 25, 106
Robinson-Schensted correspondence 50
root 39, 40, 43, 53, 57

root lattice 40, 43
root matrix 43
root system 40, 43
root vector 42, 43
row 24, 106
R-polynomial 47
R_poly 64
r_reduce 64
RS 66
`runtime` 32
`savestate` 21, 33, 106
semisimple element 39, 40
semisimple group 39
semisimple Lie rank 38
semisimple part 38
semisimple rank 39
series 22
setdefault 25, 106
shape 51, 66
sign_part 66
size 19
sort 19
Special Linear group 36, 39
spectrum 73, 88
Spin group 36
statements 21
symmetric group 51
symmetrised tensor power . 50, 53, 56
symplectic group 37
sym_char 66
sym_orbit 67
sym_tensor 73, 74
tableaux 50, 51, 57
tensor 74
term 13
tex 15, 107
then 22, 106
to 23, 106
toral 57
toral element 37, 42, 43, 86
torus 39, 40
total degree ordering 13
to_part 67
transposition 16
trans_part 67
type 10, 105
used 20, 32
vapply 29, 107
variable 27
vec 12, 107
vec_mat 19
vector 36, 57
vid 10, 21
virtual module 56
void 20
v_decomp 74
weight . . 39, 40, 42, 43, 54, 56, 57, 58
weight lattice 42, 43, 44, 48
weight vector 42, 44
Weyl chamber 44
Weyl group 40, 42, 48
Weyl word 45, 46, 48, 57
while 23, 24, 106
`write` 31, 105
W_action 64
W_orbit 64
W_orbit_size 64
W_order 65
W_rt_action 65
W_rt_orbit 65
W_word 65
X 17, 107
Young tableau 50, 51, 57

LiE MANUAL

Table of Contents

1 Introduction **1**
- 1.1 About the content of this manual 2
- 1.2 Theoretical aspects 2
- 1.3 The authors 3

2 The Interpreter **4**
- 2.1 A first look 4
  - 2.1.1 Command prolongation 10
  - 2.1.2 Getting help 10
  - 2.1.3 Identifiers 10
  - 2.1.4 File management 11
  - 2.1.5 Comments 11
  - 2.1.6 Escape to the shell 11
- 2.2 Values 11
  - 2.2.1 Integer 11
  - 2.2.2 Vector 12
  - 2.2.3 Matrix 12
  - 2.2.4 Polynomials 13
  - 2.2.5 Group 14
  - 2.2.6 Text 15
- 2.3 Operators 15
- 2.4 Using functions 18
  - 2.4.1 Function call 18
  - 2.4.2 Basic functions 19
- 2.5 Statements and clauses 20
  - 2.5.1 Assignment statements 21
  - 2.5.2 Series 22
  - 2.5.3 Blocks 22
  - 2.5.4 Conditional clauses 22
  - 2.5.5 Loop clauses 23
  - 2.5.6 Break and return 24
  - 2.5.7 Setdefault 25
- 2.6 User defined functions 25
  - 2.6.1 Function definition 26
  - 2.6.2 Local variables and blocks 27
  - 2.6.3 Make and apply 28

2.7 Global commands . . . 30
2.7.1 File management . . . 30
2.7.2 Information retrieval . . . 31
2.7.3 Memory management . . . 31
2.7.4 System parameters . . . 32

3 Terminology . . . 35
3.1 Lie groups and algebras . . . 36
3.2 Roots and weights . . . 40
3.3 The Weyl group and its action . . . 44
3.4 The Symmetric groups and related matters . . . 48
3.5 Representation theory . . . 51

4 Built-in mathematical functions . . . 57
4.1 Lie groups . . . 58
4.2 Root systems . . . 59
4.3 The Weyl group . . . 61
4.4 Operations related to the Symmetric group . . . 65
4.5 Representations . . . 67

5 Examples . . . 75
5.1 General . . . 75
5.1.1 Reversing the ordering . . . 75
5.1.2 Union and intersection of sets of vectors . . . 75
5.1.3 Sum and product of vector entries . . . 75
5.1.4 The factorial function . . . 76
5.1.5 Evaluating polynomials . . . 76
5.1.6 The sum of the coefficients of a polynomial . . . 76
5.1.7 Accessing the default group . . . 77
5.1.8 Gaussian elimination over $\mathbf{Z}/p$ . . . 77
5.2 Roots . . . 78
5.2.1 All roots . . . 78
5.2.2 The half sum of the positive roots . . . 78
5.2.3 Positive roots made negative by w . . . 79
5.3 Weyl words . . . 79
5.3.1 The Coxeter matrix . . . 79
5.3.2 All reduced Weyl words of a given element . . . 79
5.3.3 The Bruhat ordering . . . 81
5.4 Cosets in The Weyl group . . . 82
5.4.1 Right cosets . . . 82
5.4.2 Left cosets . . . 82
5.4.3 Double cosets . . . 83

5.5 Traversing the Weyl group . . . 83
5.6 Kazhdan-Lusztig polynomials . . . 85
5.7 Toral elements . . . 86
5.7.1 $SL(n, \mathbf{C})$ . . . 87
5.7.2 $SO(12, \mathbf{C})$ . . . 87
5.7.3 Spectrum . . . 88
5.7.4 Branching to a centraliser . . . 90
5.8 Computing and decomposing characters . . . 91
5.8.1 The character polynomial . . . 91
5.8.2 The dominant character . . . 93
5.8.3 Decomposition . . . 94
5.9 Checks . . . 96
5.9.1 Checking Kazhdan-Lusztig polynomials . . . 96
5.9.2 Dimension checks . . . 97
5.9.3 Checks using characters . . . 98
5.9.4 The functions *sym_tensor*, *alt_tensor*, and *plethysm* . . . 98
5.9.5 Intrinsic tests . . . 98
5.10 Branching . . . 99
5.10.1 Branching from composite groups . . . 99
5.10.2 Branching from F_4 to B_4 . . . 100
5.10.3 Branching from G_2 to A_2 using a rational function . . . 102
5.11 Overflow . . . 104
5.12 Branching to Levi subgroups . . . 104

6 Syntax . . . 105

7 References . . . 108

8 Index . . . 110