Stichting Computer Algebra Nederland (CAN)

LiE MANUAL

LiE is a software package for Lie group theoretical computations

developed by the

Computer Algebra Group
of the
Centre for Mathematics and Computer Science
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 LiE
```

LiE MANUAL

LiE is a software package for Lie group theoretical computations

developed by the

Computer Algebra Group
of the
Centre for Mathematics and Computer Science
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 LiE
```

LiE MANUAL

LiE MANUAL

Chapter 1. INTRODUCTION

a software package for Lie group theoretical computations
by the
Computer Algebra Group
Centre for Mathematics and Computer Science
Amsterdam

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. The present release should be considered as a beta version. It focuses on the representation theory of complex semisimple (reductive) Lie groups and on the structure of the Weyl groups of these Lie groups.

The basic objects of computation are vectors and matrices with integer entries representing weights, roots and the like rather than Lie algebras or Lie groups themselves. Our primary goal in realising the present version has been to cover (online) the mathematical content of the following three books:

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

The package creates an interactive environment from which commands - involving a few standard operations and certain mathematical standard functions - can be invoked. (Although, of course, 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.

The present beta version is available for the following computers: IBM RT (under AIX), SUN 3, SUN 4 and SparcStation (under SunOS 4.0), VAX 11/750 and 11/780 (under BSD UNIX 4.3). Should you want to order the package, please contact: Computer Algebra Group, c/o Mrs. W. van Eijk, Centre for Mathematics and Computer Science, P.O.Box 4079, 1009 AB Amsterdam, The Netherlands, email: wilma@cwi.nl.

1.1. About the content of this manual

For complex reductive groups, specified by type, LiE provides root systems, Weyl groups (and possibilities to compute with these objects), multiplicities and degrees of highest weight modules, tensor product decompositions and branching (i.e., restrictions of modules) to reductive subgroups, the centralizer of a semisimple element, the spectrum of such an element on a module. The full set of mathematical functions can be found in Chapter 4 of this report. In Chapter 2, we define the main terms needed to understand how the mathematics has been implemented. Chapter 3 explains how the interactive shell operates: a beginner needs to know very little. But more advanced users may actually make their own programs on this level, using the mathematical functions of Chapter 4 as their primitives.

1.2. A few technical aspects

The package is written in C, and can be made available on any system running UNIX or comparable operating systems, and (with a little more effort) probably on many other machines with a C-compiler. The interpreter has been set up with the help of the UNIX program `yacc`. The interpreter recognizes the following types of objects.

types	name	example	comment
integer	**int**	1235	up to machine length
bigint	**bin**	−12344321267	arbitrary length
vector	**vec**	[1,2,−7,6,9,8]	integer entries
matrix	**mat**	[[1,2],[3,−4]]	fixed row length
group	**grp**	A6A6E8F4T4	T4 is a 4-dim. torus
text	**tex**	"anystring"	"s required
void	**vid**		to use functions as routines

Entering LiE yields access to about 75 standard mathematical functions. They can be used directly, or in a more sophisticated way through user-defined functions in the interpreter language. The latter option makes it possible to customize and extend the package with more mathematical functions; see Chapter 5 for a few examples.

1.3. Theoretical aspects

The package is mainly intended for computations concerning semi-simple Lie groups. Since reductive groups provide a more general and at the same time more convenient setting, they form the class of groups we have chosen as objects. For notational convenience, we only allow for the simply connected group to enter the picture. Since all other reductive groups are quotients by central subgroups, we feel that this is not a major limitation anyway.

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 and matrices corresponding to various relevant objects in Lie group theory. For instance, representations are parametrized 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 its image in the reflection representation).

The emphasis has been on the development of routines for basic mathematical operations that work in greatest generality. Thus, it is quite likely that greater speed could have been achieved in specific cases with more specialised programs; to mention one major instance, we have not yet implemented Young tableaux techniques, which are known to be provide quite fast implementations for the general linear groups, but for which not even the theory fully extends to all simple Lie groups.

1.4. The authors

Arjeh M. Cohen developed the idea, wrote some mathematical functions and this manual (TEXnical typesetting help from André Heck, Marc van Leeuwen and Jan van der Steen is gratefully acknowledged), and is the project leader; Bert Lisser made the interpreter and provided a description in the form of Chapter 3 of this manual; Bert Ruitenburg tested the package under development, and edited the texts prompted by the commands `?` and `learn`; Ron Sommeling has realized the first version of the package, is the author of quite a few basic mathematical programs, the editor of many more, and the main trouble shooter; Bart de Smit wrote some of the key mathematical programs. We acknowledge the support of Guido van Rossum in the form of some useful discussions concerning the interpreter and integers of arbitrary length.

It is our intention to improve the present version and to extend it into several directions. For more information beyond what this manual has to offer, bug reports, interesting algorithms you may want us to know, or any other helpful comments, contact: Arjeh M. Cohen, CWI, Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, email: marc@cwi.nl.

LiE MANUAL

Chapter 2. TERMINOLOGY

In LiE , various mathematical notions are encoded by means of the same type (*viz.* integer, vector, matrix). This chapter helps to sort out how to interpret an object of given type as a mathematical element. It is done as follows: names of mathematical notions that are represented in LiE are listed as well as how they should be viewed as vectors, matrices, integers, or otherwise. For example, if the vector $v = [v_1, ..., v_r]$ is a *root vector* with respect to a semi-simple Lie group g of rank r, it represents the element $\sum_{i=1}^{r} v_i\alpha_i \in \bigoplus_{1\leq i\leq r} \mathbb{Z}\alpha_i$, where α_i $(1 \leq i \leq r)$ are fundamental roots of the root system of g.

It may be clear from this very example that some basic notions and ideas should be clarified. This is done in the first section of this chapter. The rest of the chapter consists of a list of alphabetically listed mathematical notions with descriptions of how they are represented in LiE .

2.0. Mathematical set up

To begin, let's look at the class of Lie groups LiE deals with: the connected reductive complex Lie groups. This class of groups comprises the essential semi-simple Lie groups, but also the well-known non-semisimple group $GL(n, \mathbb{C})$ (the group of all invertible $n \times n$-matrices). The choice of class is quite convenient: firstly, because of their classification, secondly because of their representation theory. We shall now go into these two observations into some greater depth, thereby displaying the basic approach to groups in LiE .

Due to the classification of these groups (cf. [Bourb 1975]), each connected reductive complex group is a homomorphic image with finite kernel of a product of $GL(1, \mathbb{C})$'s and simply connected simple complex Lie groups. Given the group, the product group can be obtained in a standard way. The identity component of the center of the group is a torus, i.e. a product of $GL(1, \mathbb{C})$'s, and the universal cover of the commutator subgroup is a product of simply connected simple complex Lie groups. Together these groups form the product group. The group is semi-simple when the identity component of the center is trivial (i.e.$\{1\}$). Every simply connected simple group is isomorphic to one of the classical groups $SL(n, \mathbb{C})$ ($n \geq 2$, the Special Linear group, consisting of all $n \times n$ matrices with determinant 1), $Spin(n, \mathbb{C})$ ($n \geq 5$, the Spin group, covering the Orthogonal group: the group of all invertible $n \times n$ matrices m with $m^{-1} = m^{\top}$), $Sp(2n, \mathbb{C})$ ($n \geq 3$, 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 one of the exceptional groups of type E_n $(6 \leq n \leq 8)$, F_4, G_2. See the next chapter how we use this fact to encode the isomorphism type of a Lie group as a string. Since any connected reductive complex group g is the quotient of a group $\hat{g}$ by a finite central subgroup, where $\hat{g}$ is the direct product of a torus and simply connected simple groups, g can be described by specifying the central elements of $\hat{g}$ that are in the kernel of the morphism $\hat{g} \to g$. For example, if $g = GL(2, \mathbb{C})$, then $\hat{g}$ can be taken to be A1T1, the direct product of a 1-dimensional torus T1=$GL(1, \mathbb{C})$ and A1=$SL(2, \mathbb{C})$ and g is isomorphic to the image of the canonical morphism $\hat{g} \to g$ with kernel $\{[1, 1],\ [-1, -1]\} \subset$ A1T1, where $-1 \in A1$ and $-1 \in$ T1 stand for the central elements.

The second reason for the choice of reductive groups is that every representation (a representation is always assumed to be a Lie group morphism into $GL(n, \mathbb{C})$ for some $n \in \mathbb{N}$) of g decomposes as a direct sum of irreducible representations and that the irreducible representations can be described by vectors with integral entries, the so-called weights. Consequently, a representation (also called *module*) M is fully determined by the *multiplicity* or *frequency* of M of each irreducible representation (that is, the number of times it occurs in a direct sum decomposition). Assume from now on that g is the direct product of a torus S, the so called central torus of g, and simply connected simple groups and fix a maximal torus T in g. (Since all maximal tori are conjugate, there is no harm in fixing one. LiE assumes this has been done throughout.) Then T is the direct product of S and the *semi-simple part* (i.e. the product of maximal tori of the simply connected simple components). The *Lie rank* of g is the dimension of T, we shall denote it by r; the *semi-simple Lie rank* of g is the Lie rank of the derived group g', we shall denote it by s. Consider the group $\Lambda(T)$ of all linear (rational, irreducible, 1-dimensional) characters of T; its elements are called *weights*. If $\lambda \in \Lambda(T)$ and $h \in T$, we write h^λ to denote the image of h under λ. This way $\Lambda(T)$ can be seen as an additive group; it is isomorphic to $\mathbb{Z}^r$, and affords a linear action of $W = N_g(T)/T$, the *Weyl group* of g (with respect to T). This additive group $\Lambda(T)$ naturally decomposes into a direct sum $\Lambda(T \cap Z(g)) \oplus \Lambda(T \cap g')$. The group T is diagonizable in any g-representation. In other words, if M is g-module, the restriction of M to T is a direct sum of 1-dimensional T-modules, and so described by a set of weights (with multiplicities). The *adjoint module* of g is its representation on the *Lie algebra* of g. The set of nonzero weights of T occurring in the adjoint representation is called the *root system* of g, and (often) denoted by Φ. The elements of Φ, the so-called *roots*, span the sublattice of $\Lambda(T)$ known as the *root lattice*. Up to scalar multiples, the roots are precisely those weights that occur as eigenvectors of reflections in W with eigenvalues -1.

There is a natural W-invariant inner product on the root lattice $\mathbb{Z}\Phi$ with values in $\mathbb{Z}$, which we extend to a bilinear symmetric positive definite form $(\cdot, \cdot)$ on $\Lambda(T)$ in such a way that $\Lambda(T \cap Z(g))$ is perpendicular to $\Lambda(T \cap g')$, and the restriction to the former is integral and unimodular. Choosing (and fixing) a hyperplane H through the origin (but not through any root), and fixing a so-called 'positive' half space with respect to H, there is a unique system of *fundamental roots*, that is, a set $\{\alpha_1, ..., \alpha_s\} \subset \Lambda(T \cap g')$ of s linearly independent roots with $(\alpha_i, \alpha_j) \leq 0$ for $i \neq j$ such that any other root is *positive*, i.e., a linear combination with non-negative integral coefficients of these roots or *negative*, that is, the negative of a positive root. Choose, and fix, a basis $\omega_1, ..., \omega_r$ of $\Lambda(T)$ such that $\omega_{s+1}, ..., \omega_r$ is an orthonormal basis of $\Lambda(T \cap Z(g))$, and $\omega_1, ..., \omega_s$ is the 'dual basis' of $\alpha_1, ..., \alpha_s$ in $\Lambda(T \cap g')$ with respect to $< \cdot, \cdot >$ given by $< \lambda, \mu >= 2(\lambda, \mu)/(\mu, \mu)$ (that is $< \omega_i, \alpha_j >= \delta_{i,j}$ for all $i, j \in \{1, ..., s\}$).

Any weight can be transformed by W into a unique element of shape $\sum_{i=1}^r a_i\omega_i$ with $a_i \geq 0$ for all $i \in \{1, ..., s\}$. Denoting the set of all these elements by $\Lambda^+(T)$ we obtain a so-called *fundamental domain* for the action of W on $\Lambda(T)$, usually referred to as the *Weyl chamber*. A weight is called *dominant* if it lies in $\Lambda^+(T)$. This cone also provides a partial ordering of weights: a weight v' is said to *lie under* a weight v if $v - v'$ is a linear combinations of $\alpha_1, ..., \alpha_s$ with non-negative coefficients; we also say that v is *higher than* v'. Since W permutes the linear characters of T on an irreducible g-module M according to its action on $\Lambda(T)$, it suffices for the determination of all T-characters occurring in the restriction of the g-module to T, to list those the dominant ones. It is a well known fact and crucial fact that the unique highest (dominant) weight determines M up to isomorphism.

The package also deals with Weyl groups. If g is a reductive group, its Weyl group W is (at least abstractly) the same as the Weyl group of the commutator subgroup g' of g. The group W is generated by s *fundamental reflections*, that is, the reflections r_i of $\Lambda(T)$ with -1 eigenvectors α_i $(1 \leq i \leq s)$. Thus $\beta r_i = \beta - <\beta, \alpha_i> \alpha_i$. Denote the order of $r_i r_j$ by m_{ij} (that is, m_{ij} is the least number $m > 0$ such that $(r_i r_j)^m = 1$), then $(\alpha_i, \alpha_j) = \sqrt{(\alpha_i, \alpha_i)(\alpha_i, \alpha_i)} \cos(2\pi/m_{ij})$. We include the case $i = j$, that is, we set $m_{ii} = 1$. Then W has the following abstract presentation:

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

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

Once the group g is specified, LiE fixes a maximal torus, a set of roots Φ, a subset of positive roots, whence the fundamental roots $\alpha_1, ..., \alpha_s$ (and their ordering), and an ordered basis $\omega_1, ..., \omega_r$ of fundamental weights. The function **posroots** in LiE displays all positive roots in Φ. This is an example of how some of the standard facts can be extracted from the LiE package. Other examples are the functions **diagram** and **cartan**,... but now we are running ahead; these functions will be dealt with in Chapter 4. The reader is well advised to skip the rest of this chapter when browsing/reading this manual for the first time.

2.1. branching

is another word for restriction of a g-module M to a subgroup h of g. Suppose h is a closed reductive Lie subgroup of the Lie group g. The branching problem concerns the finding of the decomposition into highest weight modules of M viewed as an h-module. Since the maximal torus T of g is unique up to conjugacy, and similarly for h, the fixed (by LiE !) maximal torus S of H may be thought of as part of T. As h also has a fixed (by LiE) set of fundamental weights, there is a unique matrix m describing the restriction $\Lambda(T) \to \Lambda(S)$ of weights on T to S on the fundamental weights. This matrix plays a crucial rôle in the function **branch**. The command **resmat** helps to find the restriction matrix in cases where h is a *fundamental Lie subgroup*. See Chapter 5 for further examples of restriction matrices.

2.2. Cartan matrix

the matrix $(<\alpha_i, \alpha_j>)_{1 \leq i,j \leq s}$ extended by zeroes to an $r \times r$ matrix.

2.3. Cartan type

of a closed subsystem Ψ of roots of Φ is the Lie group whose diagram is that of the fundamental Lie subgroup of g corresponding to Ψ.

2.4. character matrix

for the symmetric group. A group character is the trace function of a representation of the group. Such a function is constant on conjugacy classes. For the symmetric group on n letters, the conjugacy classes are parametrized by *partitions* of n. A character matrix corresponding to a character v of the symmetric group is a matrix with $n+1$ columns in which the first n entries of each row represent a partition of n and the last entry the value of the character v on the conjugacy class belonging to that partition.

2.5. closed subsystem of roots

in Φ is a subset Ψ of roots that is 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 closed Lie subgroup of g.

2.6. decomposition matrix

is a matrix representing a module M. Each row of m represents a pair v, n consisting of the dominant weight v and the multiplicity n of the highest weight module V_v with highest weight v in the g-module M. If n is allowed to be negative, then M can be thought of as a formal sum (with integral scalar coefficients) of irreducible modules. In this case M is called a *virtual module*, and the matrix a *virtual decomposition matrix*.

2.7. degree

of a representation is another word for the dimension of the underlying vector space.

2.8. diagram

a graph indicating the fundamental roots and their inner products.

2.9. distinguished coset representative

is the unique element of the Weyl group W of smallest length in its coset (be it a double, left or right coset) with respect to (a) subgroup(s) generated by fundamental reflections.

2.10. dominant weight

of a Lie group g is a linear combination of the *fundamental weights* $\omega_1, ..., \omega_r$ such that the first s scalar coefficients (corresponding to the semisimple part of the weight lattice $\Lambda(T)$) are non-negative integers.

2.11. fundamental Lie subgroup

of g is a closed Lie subgroup containing a maximal torus (which we may take to be T up to conjugacy again). Thus, if the subgroup is reductive, it can be fully described by the roots of the root system Φ of g that are also roots of the subgroup; these form a *closed subsystem of roots.*

2.12. fundamental reflection

a reflection with respect to a fundamental root (used only if a set of fundamental roots $\alpha_1, ..., \alpha_s$ has been fixed, in which case these reflections are often denoted by $r_1, ..., r_s$).

2.13. fundamental root

choosing a hyperplane H through the origin in the Euclidean space spanned by the root lattice of g with respect to T and an ordering (left/right) with respect to this hyperplane, there is a unique set of s roots closest to H in the right half space of H. The set obtained in this way is called a *set of fundamental roots.* In section 2.0 this set has been fixed and denoted by $\alpha_1, ..., \alpha_s$. It is the standard basis whenever we are looking at *root vectors.* Using the Cartan matrix, it is possible for semisimple groups to transform *root vectors* to *weight vectors.*

2.14. fundamental weight

a basis $\omega_1, ..., \omega_r$ of the *weight lattice* such that $< \omega_i, \alpha_j >= \delta_{i,j}$ for all $i, j \in \{1, ..., s\}$. See section 2.0. It is the standard basis for *weights.* The function **inprod** gives the natural Euclidean inner product for weights on this basis.

2.15. general linear group

of a vector space M, notation $GL(M)$, or if $M = \mathbb{C}^n$ also $GL(n, \mathbb{C})$ is the group of all linear transformations of M. See also *special linear group.*

2.16. highest root

is the unique positive root that is higher than any other root.

2.17. highest weight

If two weights λ and λ' are such that their difference $\lambda - \lambda'$ is a sum of *positive roots*, then λ is called higher than λ'. If M is an irreducible representation, there is a unique highest weight among the weights of T on M; it occurs with multiplicity 1. This is called the highest weight of M. Conversely, to every dominant weight, there corresponds a unique irreducible representation of g.

2.18. highest weight module

if λ is a dominant weight of T, the irreducible g-module with *highest weight* λ is called the highest weight module for λ.

2.19. length

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

2.20. Levi subgroup

is the semisimple part of the fundamental Lie subgroup corresponding to a *closed subsystem* whose *fundamental roots* are also *fundamental roots* of g. Thus, the (types of) maximal Levi subgroups are obtained by removing one node from the diagram of g.

2.21. Lie group

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

2.22. Lie algebra

is a finite-dimensional vector space V supplied with a bilinear operation $[\cdot,\cdot] : V \times V \to V$ with the following rules $[x,y] = -[y,x]$ and $[[x,y,],z] + [[y,z,],x] + [[z,x],y] = 0$ (anticommutativity and the Jacobi identity, respectively) for all $x, y, z \in V$. They play no rôle in this package. If V is a complex vector space, there is a notion of reductive algebras making semisimple Lie algebras correspond bijectively to simply connected reductive Lie groups, see [Hum 1972]. To name one consequence of this correspondence, representation theory of such Lie algebras coincides with the representation theory LiE deals with.

2.23. matrix

frequently viewed as a collection of rows, each row being a vector. Thus, for example a *root matrix* will be understood to be a matrix whose rows are *roots*. See also *decomposition matrix*, *character matrix*, *orbit matrix*, *multiplicity matrix*, and *restriction matrix*.

2.24. maximal torus

is a torus that is maximal with respect to inclusion. In a reductive Lie group, such tori exist and any two are conjugate. In LiE , the choice of *weights* and *roots* etc. is done once and for all with respect to a fixed maximal torus T.

2.25. multiplicity matrix

is a matrix whose rows have shape v, n, where v is a dominant weight (and so a vector with $Lierank(g)$ components, all entries positive) and n is an integer, called the *multiplicity* of v. If n is allowed to be negative, we refer to it as the *virtual multiplicity* of v, and to m as a *virtual multiplicity matrix.* These multiplicity matrices correspond to $N(T)$-modules.

2.26. orbit matrix

with respect to a group of matrices is the matrix whose rows form a single orbit of the group.

2.27. partition of n

is a vector $v = [v_1, ..., v_s]$ with $n = \sum_{i=1}^{s} v_i$ and $v_i \geq v_{i+1} \geq 0$ for all $i \in \{1, ..., s-1\}$. The LiE command **partitions**(n) produces a matrix whose rows represent the partitions (sufficiently many zeroes to make their sizes n). Partitions of n parametrize the conjugacy classes of the symmetric group on n symbols (the numbers v_i stand for the lengths of the cycles of the permutation on n symbols) and also their irreducible characters. See [JamKer 1981] for details. In LiE the irreducible character coming with partition v is given (upon the command **symchar**(v)) by a *character matrix.*

2.28. positive root

is a linear combination of fundamental roots all of whose coefficients are non-negative integers. Every root is either positive or negative (i.e., the negative of a positive root).

2.29. reduced Weyl word

a *Weyl word* $[a_1, ..., a_m]$ with $a_i \in \{1, ..., s\}$ such that corresponding element $r_{a_1} ... r_{a_m}$ of W cannot be expressed as a product of fewer than m fundamental reflections.

2.30. reductive group

is a connected complex Lie group g whose derived group g' is semisimple and with the property that it is the direct product of g' and a torus. In LiE , the type **group** always refers to a simply-connected reductive complex Lie group.

2.31. reflection

of a Weyl group is a linear (nonidentity) transformation (automorphism) of the weight lattice preserving $(\cdot,\cdot)$ and fixing (vectorwise) a sublattice of rank $r-1$. The -1 eigenvector, suitably normalized is a *root*.

2.32. representation

of a group g is a (Lie) group morphism into the general linear group $GL(n,\mathbb{C})$ for some n. The irreducible representations of finite groups as well as Lie groups are (up to equivalence) determined by their characters. For the symmetric groups, the characters are indexed by partitions and, provided the group permutes a small number of letters, can be found in LiE as a *character matrix*. For reductive Lie groups, the irreducible representations are indexed by *dominant weights*. For the general and special linear groups, the representations can alternatively be indexed by partitions (this is where the Young tableaux come in): in the special linear (simple Lie group if $n > 1$!) case, the representation with *partition* $v = [v_1, ... v_d]$ has dominant weight $[v_1 - v_2, v_2 - v_3, ..., v_{n-1} - v_n]$, where $v_i = 0$ for $i > d$. Thus, the partition $[d]$ corresponds to the d-th symmetric power of the standard module, i.e., the one with dominant weight $[d, 0, 0..., 0]$, and the partition $[1, 1, ..., 1]$ corresponds to the d-th alternating power of the standard module, i.e., the one with dominant weight $[0, 0.., 0, 1, 0.., 0]$ (the d-th component is 1). For the general linear case: think of $\mathrm{A}n-1\mathrm{T}1$ (the universal cover of $GL(n,\mathbb{C})$) as embedded in $SL(n+1,\mathbb{C})$ with the $\mathrm{A}n-1$ part in the upper $n \times n$ entries of the matrices representing $SL(n+1,\mathbb{C})$; and interpret the partition for $GL(n,\mathbb{C})$ as above for $SL(n+1,\mathbb{C})$.

2.33. restriction matrix

of a reductive subgroup h of g is the matrix whose i-th row $(1 \leq i \leq r)$ is the restriction of the i-th *fundamental weight* of g with respect to T to the (fixed) maximal torus of h, viewed as a *weight* of h.

2.34. root

a nonzero weight that occurs in the adjoint representation of g (alternatively: the suitably normalized eigenvector of a reflection in W with eigenvalue -1).

2.35. root lattice

the subgroup of the *weight lattice* generated by the roots of g with respect to T. Thus, it is a lattice in Euclidean space of rank s (i.e., isomorphic to $\mathbb{Z}^s$) and with the natural inner product $(\cdot,\cdot)$ discussed in section 2.0. The elements of the root lattice are *root vectors*. If g is semisimple, the converse is also true. See also *weight*.

2.36. root matrix

is a matrix whose rows represent *roots*.

2.37. root system

is the set of all roots of the Weyl group W of g. It is usually denoted by Φ. Since with every root, its negative is a root as well, the command that creates a matrix of roots (*viz.* **posroots**, see section 4.45) only lists the *positive roots.*

2.38. root vector

the vector $v = [v_1, ..., v_s]$ is interpreted as the sum $\sum_{i=1}^{r} v_i\alpha_i + \sum_{j=s+1}^{r} v_j\omega_j$. Thus, a root vector belongs to the root lattice if and only if its central torus part vanishes: $v_{s+1} = ... = v_r = 0$.

2.39. semisimple element

is a diagonizable element of g. (This notion is independent of the representation with respect to which the basis is chosen.) A semisimple element of g is conjugate to an element of T. Hence, its conjugacy class contains an element of shape $\prod_{i=1}^{r} h_i(\lambda^{a_i})$ of T for some $\lambda \in \mathbb{C}$, where $h_i(\lambda)$ is the unique element $h(\lambda) \in T$ with $h(\lambda)^{\omega_j} = \delta_{i,j}\lambda$ for all j. In LiE a vector $[a_1, ..., a_r, n]$ represents a semisimple element of T of the form $\prod_{i=1}^{r} h_i(\lambda^{a_i})$ with $\lambda = 1$ if $n = 0$ and $\lambda = e^{2\pi i/n}$ otherwise. Since this is not the usual (= diagonal matrix in the standard representation) presentation of a semisimple element in a Lie group like $GL(n, \mathbb{C})$, an example is given in Chapter 5 of how to transform from one presentation to another.

2.40. special linear group

on a vector space M is the closed Lie subgroup of the general linear group $GL(M)$ consisting of all transformations with determinant 1.

2.41. symmetric group

on n letters is the group of all permutations of the set $\{1, ..., n\}$. Its conjugacy classes are described by *partitions*, as well as its characters. They play a rôle in **plethysm**.

2.42. symmetrized tensor

of a module M with respect to the partition v of an integer n is the restriction to g of the irreducible module for $GL(M)$, the general linear group of M, corresponding to the partition v'. The resulting module for g is called the **plethysm** for g with respect to v.

2.43. torus

a group isomorphic to $GL(1, \mathbb{C})^n$ for some n; the number n is the dimension of this reductive Lie group. A *torus of g* is a subgroup of g all of whose elements are semi-simple. Every torus of g is contained in a maximal torus, and every maximal torus is conjugate to T, the fixed maximal torus (cf. section 2.0). See also *semi-simple* *elements.* A fundamental property of a torus is that all of its representations decompose into direct sums of 1-dimensional representations (also called *linear characters*). The 1-dimensional representations are the *weights* of g (with respect to T).

2.44. vector

used as a representative of a *semisimple element*, *root*, *weight*, *Weyl word*, or a *partition*.

2.45. virtual decomposition matrix

see *decomposition matrix.*

2.46. virtual multiplicity matrix

see *multiplicity matrix.*

2.47. weight

(of g with respect to T) is a 1-dimensional representation of T; in other words: a linear character of T, that is, a morphism $T \to \mathbb{C}^*$. The vector $v = [v_1, ..., v_r]$ represents the weight $\sum_{i=1}^{r} v_i \omega_i$, where ω_j is the j-th *fundamental weight* of g with respect to T.

2.48. weight lattice

is the set $\Lambda(T)$ of all weights of g with respect to T. If λ and μ are weights, $\lambda + \mu$ is the weight defined by $h^{\lambda+\mu} = h^{\lambda} \cdot h^{\mu}$ for all $h \in T$. This turns the set of all weights into an additive group isomorphic to $\mathbb{Z}^r$.

2.49. Weyl group

is the quotient of the normalizer $N_g(T)$ of the maximal torus T in g by T. It is a finite group, usually denoted by W, that has a linear representation on the *weight lattice* $\Lambda(T)$ in which the *fundamental reflections* serve as a canonical set of generators.

2.50. Weyl word

a vector $[a_1, ..., a_m]$ with $a_i \in \{0, 1, ..., s\}$ representing the product $r_{a_1} ... r_{a_m}$ of fundamental reflections r_{a_j} in W. Here r_i is the i-th fundamental reflection if $i > 0$ and r_0 is the identity.

LiE MANUAL

Chapter 3. THE INTERPRETER

In this chapter, the facts needed to run a successful LiE session are described. We discuss the features of the interactive shell, that interprets the commands entered during a session. After an introductory session, we give more details of the types of objects the interpreter recognizes. Then, in section 3.3, the operators defined in the package are listed. Section 3.4 digresses on the notion of functions. The mathematical standard functions in Chapter 4 are examples, but it is also possible for the user to define functions.

3.1. A first look

An interactive session of LiE starts by executing the command `LiE` on your machine (provided LiE has been installed; the leaflet accompanying the software package explains how to do that). You will then enter the *Lie shell.* In this mode, you can enter commands. Execution of a command will be performed once a carriage return (**CR**) has been entered. The command will be read by an *interpreter* built into LiE and, if necessary, will invoke some of the standard mathematical functions. The system will respond to the command by returning an answer if relevant. The prompt character > tells you it is ready to accept the next command. The command

```
    quit
```

enables you to terminate the LiE session. The interpreter possesses some features of a pocket calculator. For example, the commands

```
    2+a
    f(v[2])
    if 2<3 then 1 else 0 fi
```

are arithmetic expressions; the commands

```
    a == b && 2<3
    a != [1,2]
    f(3)<=7 || 2+3 == 5
```

are logical expressions; and the commands

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

are statements.

Once an expression has been entered, it will be evaluated and the result will appear on your screen.

There are commands to control input and output flow, like `quit` above; here are some more

```
    on monitor
    edit filename
    exit
```

(the last command has the same effect as **quit**). The command

```
    f(int x)=2*x
```

illustrates how to define functions. The core of LiE is a batch of standard functions, some examples being

```
    diagram(E8)
    partitions(6)
```
The former commands prompts:
```
             O 2
             |
             |
    O---O---O---O---O---O---O
    1   3   4   5   6   7   8
```
The latter returns a matrix whose rows represent partitions of 6 in a way to be explained in Chapter 2.

Finally, there are some features to help you out, such as
```
    listvars
    learn lie group
```
The second command leads to a text being printed with an indication of what the authors think a Lie group is.

The objects used for calculation are of the following types: **bin** (big integer), **int** (integer) , **vec** (vector) , **mat** (matrix) , **grp** (group) or **tex** (text), **vid** (void). There is no declaration of types for variables, the type is determined by the value. Change of type by assignments is permitted, as long as the type of variables are uniquely determined. (Change of type is for that reason not allowed in **if then** and **for** statements.)

We end this section with a few essential details.

3.1.1 Command ends As mentioned above, a command ends with a carriage return (**CR**). We have implemented this rule because, usually, one line suffices for a command. If more lines are needed, the command should be entered with the character \ at the end of each line of the command except for the final one. \ is then equivalent to a space. [However, this cannot be used after `?` or `help` or `:` or in a text or comment.]

3.1.2 Comment The character `#` occurring for the $(2i+1)^{st}$ time (for some $i \geq 0$) means that the string following that character and ending with the next occurrence of the character `#` will be viewed as comment.

3.1.3 Escape to login shell The character `:` as first character of a line means that the string which follows that character and ends with a `CR` will be considered as a command to the login shell. [This feature only applies to UNIX implementations of LiE .]

3.1.4 Exit Exit the program by typing `quit`, `stop` or `exit`.

3.1.5 Help Use `?` `help` to make enquiries. More text following `?` prompts what you might want to know about the specified text. For example, `?functions` returns the list of mathematical standard functions available. The command `?`*name* returns information about the variable, functions or operators with the name *name*. Thus `?lierank` will return information on the mathematical function **lierank**. By the way, for mathematical functions, this is usually a summary of the information to be found in Chapter 4 of this manual.

The command `listvars` returns a list of variables defined in the session. The command `listfuns` returns a list of functions defined in the session. The command `listops` returns a list of operators (cf. the next section).

3.1.6 File management Predefined functions or variables from a file named *filename* can be read with command `read` *filename* . A file named *filename* can be edited after issuing the command `edit` *filename* . When the editing session has been completed, the file will be read.

3.1.7 Variables Variable names are strings of letters and digits, the first character must be a lower case letter. There exists also the notion of local variables, for their names hold the same rules (see also 3.5.8).

The variable *a* gets the type vector after execution of the statement `a=[1,2]`, and the type matrix after `a=[[1,2]]`.

The variable **$** has the value of the expression that has been evaluated as a result of the last statement entered. (Note: here *statement* is used in the sense of the forthcoming section 3.5; this implies that

```
    10
    13; $
```

returns 10 rather than 13.)

3.1.8 Functions If a function is called, all of its arguments are computed; next the function will be executed. During the function execution, the arguments are considered as local variables. They can be changed, but the parameters at which the function is called remain unchanged (call by value). It is also possible to have variables local to a function. For example the assignment *loc a=10* in a function definition introduces by execution a new variable *a* local to that function. After execution of the function the local variables are vanished and global variables with the same name are again avaible with there old values. (Warning: if a variable name is used without *loc* then first is checked wether that variable name already exists. If it exists then the last variable with that name that is introduced (during computation) is used, otherwise it is considered as a local variabele. Thus it is possible to change global variables, but not to create new global variables.)

Functions can only be defined on top level. At the moment of definition of a function, the function is nothing more than a string. During the function call the interpreter gives the symbols their types and values. Subsequently, the function will be executed.

3.1.9 Abort To abort a computation that is running, type `<control> c`, that is, press the control key and the `c` simultaneously.

3.2. Types

The available types of the objects of computation are **int** for integer, **bin** for arbitrary length integer, **vec** for vector, **mat** for matrix, **grp** for group, **tex** for text, and **vid** for void.

3.2.1 Integer *Input notation*: a string of digits possibly headed by a minus sign. The passing from arbitrary length integers (**bin**) to integers (**int**) and back is done automatically. An exception is made for vectors and matrices: their entries cannot be **bin**. We have chosen for this option because of speed and memory storage reasons, bearing in mind that most of the computations we are concerned with involve matrices and vectors with relatively small integers. Since there is no overflow check in most of the algorithms involved, the user should always be aware of possible overflow.

3.2.2 Vector All vector entries, in this manual usually referred to as *components*, are integers. *Input notation*: `[` followed by a row of integers separated by commas, and ending with `]` . If v is a vector with n components (i.e., of length n), then, for each i ($1 \leq i \leq n$), the number $v[i]$ is the i-th component of v.

3.2.3 Matrix All matrix entries are integers. *Input notation*: `[` followed by a row of vectors separated by commas, and ending with `]`, all the vectors must have the same size. If a is a matrix with n rows, then, for each i ($1 \leq i \leq m$), the vector $a[i]$ is the i-th row of the matrix a. Example:

```
> [[1,2],[3,-4]]
| 1  2 |
| 3 -4 |
```

3.2.4 Group As mentioned in Section 2.0, the isomorphism class of a simply-connected reductive complex Lie group (i.e., an object of type **grp**) is described by a character string. A group name consists of a string of upper case letters followed by numbers. The letters are from A,B,C,D,E,F,G,T. Here A,...,G refers to the Cartan denotation of the Lie type of a complex simple simply connected Lie group and the number following it to the simple group in question. Thus, the classical groups are (cf. [Bourb 1968])

`An` $= SL(n+1,\mathbb{C})$, `Bn` $= Spin(2n+1,\mathbb{C})$, `Cn` $= Sp(2n,\mathbb{C})$, `Dn` $= Spin(2n,\mathbb{C})$.

The letter T stands for torus and the number following it for its dimension. Thus Tn $= (GL(1,\mathbb{C}))^n \cong (\mathbb{C}^*)^n$. The full string stands for the direct product of the simple simply connected Lie groups and tori of which it is built up [fine print: according to this convention, the mathematical function `centr` returns a simply connected group; the actual group that the result refers to is a reductive group (a centralizer of a semisimple element), which is a central quotient of the simply connected group]. Example:

A3T4B12A4T6A3E7 stands for the direct product

$$SL(4,\mathbb{C}) \times Spin(25,\mathbb{C}) \times SL(5,\mathbb{C}) \times SL(4,\mathbb{C}) \times E_7(\mathbb{C}) \times (GL(1,\mathbb{C}))^{10}.$$

Note: The actual group assigned to \$ will be A3B12A4A3E7T10. In other words, the order of the factors does not change but for those containing part of the central torus. The full central torus is collected and listed at the end. This is of importance when working with the corresponding Weyl group, and when selecting a factor.

If the group g is a direct product of a torus and n simple Lie groups, then $g[i]$ ($1 \leq i \leq n$) is the i-th component of g and $g[0]$ is the central torus (i.e., the intersection of a maximal torus with the center of the full group). Thus, for g as in the above example, we have $g[0] = T10$, and $g[2] = B12$.

3.2.5 Text A text looks like "any string, without carriage returns, beginning and ending with the character ".

3.2.6 Void Void is the type having no value. For instance the function *print* returns no value, thus returns a *void*.

There are various standard functions that facilitate the use of these types. For example `size`(v), where v is a vector, gives the number of components (entries, or dimension) of the vector. See Section 3.6 for more.

3.3. Operators

We describe the standard operators defined in LiE . No provisions have been made for the user to define new operators. This is no serious drawback since user defined functions are allowed (cf. section 3.4).

3.3.1 Arithmetic operators The following operators provide the standard arithmetic for integers, vectors and matrices. The usual priority rules are valid. In operators applied to vectors and/or matrices, the sizes (respectively row and column sizes) often must satisfy some evident additional conditions (like $size(y) == colsize(x)$ for $+$ defined on a pair x, y of a matrix x and a vector y.)

operator definition	interpretation
int (**int** x + **int** y)	$x+y$
mat (**mat** x + **mat** y)	$x+y$
vec (**vec** x + **vec** y)	$x+y$
vec (**int** x + **vec** y)	$[x,\ y[1],\ y[2], .., \ y[size(y)]]$
vec (**vec** x + **int** y)	$[x[1],\ x[2], .., \ x[size(x)],\ y]$
mat (**mat** x + **vec** y)	$[x[1],\ x[2],\ , .., \ x[rowsize(x)],\ y]$
int (**int** x - **int** y)	$x-y$
vec (**vec** x - **vec** y)	$x-y$
mat (**mat** x - **mat** y)	$x-y$
vec (**vec** x - **int** y)	$[x[1], ..., \ x[y-1],\ x[y+1], .., \ x[size(x)]]$
mat (**mat** x - **int** y)	$[x[1], ..., \ x[y-1],\ x[y+1], .., \ x[size(x)]]$
mat (* **mat** x)	x^{T}, the transposed of x
int (**int** x * **int** y)	$x \cdot y$
vec (**int** x * **vec** y)	$x \cdot y$ (scalar product)
vec (**vec** x * **mat** y)	$x \cdot y$
mat (**int** x * **mat** y)	$x \cdot y$ (scalar product)
int (**vec** x * **vec** y)	standard inner product of x and y
mat (**mat** x * **mat** y)	matrix product xy of x and y
vec (**mat** x * **vec** y)	$((xy^{\mathsf{T}})^{\mathsf{T}})$, where $^{\mathsf{T}}$ means 'transposed of'
int (**int** x ^ **int** y)	x^y
mat (**mat** x ^ **int** y)	x^y
vec (**vec** x ^ **vec** y)	$[x[1], .., x[size(x)], y[1], .., y[size(y)]]$
mat (**mat** x ^ **mat** y)	$[x[1], .., x[rowsize(x)], y[1], .., y[rowsize(y)]]$
tex (**tex** x ^ **tex** y)	concatenation of x and y
int (**int** x / **int** y)	x/y rounded of downwards
vec (**vec** x / **int** y)	$[x[1]/y, ..., x[size(x)]/y]$
mat (**mat** x / **int** y)	$[x[1]/y, ..., x[rowsize(x)]/y]$
int (**int** x % **int** y)	x mod y

int (**vec** x % **int** y)	$[x[1] \bmod y, ..., x[size(x)] \bmod y]$
int (**mat** x % **int** y)	$[x[1] \bmod y, ..., x[rowsize(x)] \bmod y]$

3.3.2 Logical operators As in C, no special type 'boolean' has been created. We also abide by the rule that the value of an integer, whenever interpreted as a boolean, stands for **true** if and only if it is nonzero. Nevertheless, the result of a logical operator will not be accepted as an **int** throughout (for example `(1 || 1) == 1` will not be accepted).

operator definition	interpretation
int x < **int** y	$x < y$
int x <= **int** y	$x \leq y$
int x > **int** y	$x > y$
int x >= **int** y	$x \geq y$
int x == **int** y	$x = y$
int x != **int** y	$x \neq y$
vec x == **vec** y	$\forall i \in \{1...n\}\ \ x[i] = y[i]$
mat x == **mat** y	$\forall i \in \{1...n\}\ \ x[i] = y[i]$
int x || **int** y	x or y
int x && **int** y	x and y
! **int** x	not x

3.3.3 Group operators The operator `*` applied to groups g and g' forms the direct product $g \times g'$. Example `A2T4A7*E6E7T4E8` returns `A2A7E6E7E8T8`.

3.4. Functions

Besides the definition and call of a function, we discuss two operators that use functions to construct objects, *viz.* **make** and **apply**.

3.4.1 Definition A *function definition* has shape

$$\begin{aligned} &functionname(\textbf{typeid1}\ arg11, arg12, ..., arg1n_1;\\ &\qquad \textbf{typeid2}\ arg21, arg22, ..., arg2n_2; ...;\\ &\qquad \textbf{typeidm}\ argm1, argm2, ..., argmn_m) = \ statement \end{aligned}$$

Where *functionname* and *argij* are identifiers and **typeid** is **int**, **vec**, **mat**, or **grp** . Example:

```
f(int x)=2*x
gcd(int x,y)=if y==0 then x else gcd(y,x%y) fi
```
.

3.4.2 Call A function call looks like `functionname` $(arexp1, arexp2, ..., arexpn)$. Continuing with the last example, we get 6 after the function call `f(3)` , while `gcd(4,6)` prompts 2. If a function is called, its arguments will be evaluated first.

3.4.3 Make In this section, n and n' are positive integers, v, v', and v'' vectors, and m is a matrix. The operator **make** is a tool to create new vectors and matrices using a function.

make(f, n), where f is a function from integers to integers, returns the vector v with $v[i] = f(i)$ for each $i \in \{1, ..., n\}$. For example, the command **make**$(f, 4)$ with f as in the above example prompts with `[2 4 6 8]` .

make(f, v'), where f is a function from integers to integers, returns the vector v with $v[i] = f(v'[i])$ for each $i \in \{1, ..., size(v')\}$. Example: `make(f,[1,2,3])` returns `[2 4 6]` .

make(f, v', v'') where f is a function from pairs of integers to integers, returns a vector v with $v[i] = f(v'[i], v''[i])$ for each $i \in \{1, ..., size(v')\}$.

make(f, n', n), where f is a function from pairs of integers to integers, returns a matrix m with $m[i,j] = f(i,j)$ for each $i \in \{1, ..., n'\}$ and $j \in \{1, ..., n\}$. For example `make(gcd,3,3)` returns

```
| 1  1  1 |
| 1  2  1 |
| 1  1  3 |  .
```

3.4.4 Apply Again n, n' will be integers throughout the section; v will be a vector and m a matrix. We shall use the n-th iterate of the function f:

$$f^n(n') = \begin{cases} f(n') & \text{if } n = 1; \\ f(f^{n-1}(m)\ ) & \text{if } n > 1. \end{cases}$$

iapply(f, n, n') where f is a function from integers to integers, returns the integer $f^n(n')$.

vapply(f, n, n') where f is a function from integers to integers, returns the vector $[n', f(n'), f^2(n'), ..., f^n(n')]$. Example: `vapply(f,4,1)` with f as above returns `[1 2 4 8 16]`.

vapply(f, n, v) where f is a function from vectors to vectors, returns the vector $f^n(v)$.

mapply(f, n, m) where f is a function from matrices to matrices, returns the matrix $f^n(m)$.

vapply(f, n, v) where f is a function from vectors to integers, returns the vector $[\ v[1], ..., v[size(v)], f(v), f^2(v), ..., f^n(v)]$, where

$$f^n(v) = \begin{cases} f(v) & \text{if } n = 1; \\ f(\ vapply(f, n-1, v)\ ) & \text{if } n > 1. \end{cases}$$

For example, the function `fib` builds the Fibonacci sequence

```
> f(vec x)=x[size(x)]+x[size(x)-1]
> fib(int s)=vapply(f,s,[1,1])
> fib(10)
[ 1 1 2 3 5 8 13 21 34 55 89 144 ]
```

3.5. Statements

Statements can be either *simple* or *compound*. There are 4 kinds of simple statements.

Assignment of shape: *identifier* = *arithmetic expression*.

Component assignment of shape: *identifier* [*arexpr1*] = *arexpr*.

Element assignment of shape: *identifier* [*arexpr1* , *arexpr2*] = *arexpr*.

Expression of shape: *arexpr*. A command consisting of an expression (be it logical or arithmetic) will be considered to be a statement. In fact LiE views the input `arexpr` as the assigment `$ = arexpr`.

Observe that a simple statement cannot be empty.

A compound statement is, like the name suggests, composed of simple statements. There are three ways to compose new statements from given ones, namely **sequencing**, the **if** and the **for** loop.

3.5.1 The 'sequence' operator The command

statement1; *statement2*

executes *statement1* followed by *statement2*.

3.5.2 The 'if then' operator The command

if *arexpr* **then** *statement* **fi**

executes *statement* if *arexpr* is not equal to 0.

3.5.3 The 'if then else' operator The command

if arexpr **then** *statement1* **else** *statement2* **fi**

where *arexpr* has type **integer**, executes *statement1* if *arexpr* is not equal to 0, and *statement2* otherwise.

3.5.4 The 'for' operator The command

for *ident* = *arexpr1* **to** *arexpr2* **do** *statement* **od**

where *arexpr1* and *arexpr2* have type integer, repeats *statement* a total of $arexpr2 - arexpr1 + 1$ times, and returns the last assumed value of *statement*. At the i-th execution the identifier *ident* has value $arexpr1 + i - 1$. After the **for** loop has been executed, the local variable *ident* no longer exists. It is allowed to change the value of *ident* during execution of the loop.

3.5.5 The 'for in' operator The command

for *ident* **in** *v* **do** *statement* **od**

where *v* has type **vector**, repeats *statement* a total of $size(v)$ times, and returns the last assumed value of *statement*. At the i-th execution of *statement*, the local variable *ident* will have the value of the i-th component of the vector v. Thus, *ident* starts with the value $v[1]$ and ends with the value $v[size(v)]$. For example

```
    sum(vec v)=sum_loc(0)
    sum_loc(int s)=for component in v do s=s+component od;s
```

By the way, the statement

```
    sum(vec v)=s_loc=0; \
    for component in v do s_loc=s_loc+component od;s_loc
```

would lead to a function returning the same integer but with the side effect that *s_loc* is a global variable.

Regarding *ident*, the same rules apply as before: it is not allowed to change *ident* during execution; after execution *ident* is no longer defined.

3.5.6 The 'for row' operator Finally, there is an analog for matrices as well:

for *ident* **row** a **do** *statement* **od**

where a has type **matrix**, repeats *statement* a total of $rowsize(a)$ times, and returns the last assumed value of *statement*. At the i-th execution of *statement* the local variable *ident* has the value of the i-th row of the matrix a. So *ident* starts with the value $a[1]$ and ends with the value $a[rowsize(a)]$. See the previous section for the rules regarding *ident*.

3.5.7 break It is possible to exit a **for** loop before the index reaches the end by using the function

break(int vec mat tex or **grp** a)

which returns a and exits the closest loop in which it is involved. (See also 3.6.2.) For example
```
    prime(int n)=v=[2];for i=3 to n do \
       if primetest(i) then v=v+i fi od;v
    primetest(int k)=for n in v do \
       if k%n==0 then break(0) else 1 fi od
    prime(10)
```
returns
```
     [2 3 5 7 ]
```

3.5.8 local variables and blocks Starting LiE , one enters the LiE shell the highest level. Variables assigned on this level are global variables and can in principle be used in all computations. Blocks are lower levels, where also variables can be introduced and these variables exist until the computations in LiE return to a higher level. In LiE the blocks are the following:

- The **then** and **else** part of the **if then else** operator.
- The **do od** part of all **for do** operators.
- The body of a function during a call of that function.
- A sequence of commands placed between curly braces { }.

Entering a block is going to a lower level, exiting a block is going to a higher level. During computations there is at each moment a complete hierarchy of levels. Inside a block an assignment *loc* a=1 creates a new variable with the name a and assigns the value 1 to it. When the block is left this variable will be removed. If a variable with the same name already did exist outside the block, it becomes again available with its old value. If in the assignment a variable name is used without *loc*, then first will be checked whether that variable name already exists. If it exists then the last variable with that name that is introduced during computation is used, otherwise it is considered as a local variable. It is not allowed to change the type of a variable inside a block, but it is possible to change the value of a global variable.

3.6. Standard functions

A few convenient functions are available as standard functions. It is up to the user to define more whenever convenient. The standard functions cannot be redefined.

3.6.1 Integers, Vectors, Matrices

name	description
int `abs`(**int** x)	the absolute value of x
vid `factor`(**int** x)	the factorization of x into prime factors
int `size`(**vec** v)	the number of components of v
vec `null`(**int** n)	the null vector of length n
int `rowsize`(**mat** a)	the number of rows of a
int `colsize`(**mat** a)	the number of columns of a
mat `id`(**int** n)=	the $n \times n$ identity matrix
mat `null`(**int** m,n)	the $m \times n$ null matrix
mat `blockmat`(**mat** a, **mat** b)	the block matrix of a and b
vec `diag`(**mat** a)	the vector $[a[1,1], a[2,2], ...]$
vec `vecmat`(**mat** a)	concatenation: $a[1]\wedge a[2]\wedge ...\wedge a[rowsize(a)]$
mat `matvec`(**vec** v; **int** n)	$[v[1..m], v[m+1..2m], ...]$

3.6.2 Void

name	description
vid void(**any type** a)	object of type void, for suppressing results
vid `print`(**any type** a)	prints a
vid `error`(**tex** a)	prints a and terminates execution of the statement
vid `break`()	terminates execution of the closest **for** loop in which it occurs

3.6.3 Groups

name	description
int `compsize`(**grp** g)	the number of simple components of g

Of course, the bulk of LiE are the mathematical standard functions documented in Chapter 4. They can also be found by executing the LiE command `?functions`. Here we give a few samples in which g stands for a variable of type **group**. The command **setdefault**(g) sets the default Lie group equal to g. Most mathematical functions depending on a Lie group g have g in their last arguments. Once the default Lie group has been set to g, it is possible to call these functions with the parameter g omitted. The missing parameter will then be assumed to have the value of the default Liegroup. For example, after execution of the compound statement

```
setdefault(A3); worbit([1,1,1])
```

the same matrix will appear as upon execution of the simple statement

```
worbit([1,1,1],A3)
```

(For details on **worbit**, cf. Chapter 4).

The function **int** `lierank`(g) returns the Lie rank of the group g.

A way to run over many of the available groups is furnished by the command **grp** `liegroup` (**int** m, n) , yielding

$$\begin{cases} \text{Tn} & \text{if} \quad m=0 \\ \text{An} & \text{if} \quad m=1 \quad \text{and} \quad n\geq 1 \\ \text{Bn} & \text{if} \quad m=2 \quad \text{and} \quad n\geq 2 \\ \text{Cn} & \text{if} \quad m=3 \quad \text{and} \quad n\geq 2 \\ \text{Dn} & \text{if} \quad m=4 \quad \text{and} \quad n\geq 3 \\ \text{En} & \text{if} \quad m=5 \quad \text{and} \quad 6\leq n\leq 8 \\ \text{Fn} & \text{if} \quad m=6 \quad \text{and} \quad n=4 \\ \text{Gn} & \text{if} \quad m=7 \quad \text{and} \quad n=2 \end{cases}$$

Conversely, the simple Lie group g can be encoded into a vector $[m, n]$ satisfying **liegroup** $(m, n) = g$ by means of **liecode**(g).

3.7. File management

The commands of this section use external files to the program LiE . They cannot be invoked inside compound statements.

It is possible to execute the commands listed in a file. If the name of that file is *filename*, it can be read during a LiE session by means of the command **read** *filename*. This will result in the required execution.

The file named **initfile** will be read upon entrance of the program LiE , before the first prompt appears. It has to be located in the same directory as where LiE has been started, and can be edited by the user. The command **edit** *filename* makes it possible to enter an edit session of the file named *filename* with the standard editor of your machine; if the UNIX environment variable $EDITOR has been set in the UNIX shell that executes LiE , it will select the editor named by that variable. If the editing session is terminated, the LiE session continues and the file is read. A default filename can be assigned with the command `edfil` *filename*. After this assignment the file *filename* can be edited via execution of the command `edit`.

To save the user defined functions of a particular session, execute the command **write** *filename*. As a result, these functions are written in the file *filename*. See also the command **on monitor** in section 3.10.

3.8. Information retrieval

3.8.1 Functions, etc. Information about a function, operator or a reserved word (like **for**) called *name* can be gotten by the command `?`*name*. A list of the documented reserved words can be gotten by the entering `?index`.

3.8.2 Learn Information about a mathematical term such as 'Lie group' can be gotten by entering `learn` *term*. For example `learn lie group`. (Upper and lower case characters will be distinguished.) A list of the documented terms can be gotten by entering `learn index`.

3.9. Garbage collection

To handle storage in the package LiE , a few key algorithms have been written, that take care of all memory allocations and garbage collections. These (and no other memory allocations) are used wherever needed in the collection of mathematical standard functions. They should be of no concern to the user. But if the user feels that there might be a bug, she can select the option to turn the garbage collection of, see the next section.

No `clear` *variable* or `delete` *variable* command has been installed, because the command *variable* `= 0` approximates the intended effect quite satisfactorily. It is possible to run the garbage collector by the explicit call *gcol*.

The command **used** provides the number of variables and functions used thus far.

3.10. System parameters

Using the command **on** *feature*, respectively **off** *feature* it is possible to turn on/off one of the following *features*: **bigints** (allowing to banish arbitrary length integers), **runtime** (printing the time used for each command), **gc** (garbage collection), **monitor** (writing to the file named **monfil** in the directory where LiE has been started), or **prompt** (to suppress the prompt character >).

Finally, we repeat that the abort character `<control>c` forces the running command to end.

3.11. Examples

Here we list some elementary examples to illustrate the interpreter language. More intricate examples, involving the use of mathematical standard functions, appear in Chapter 5.

3.11.1 Operators The first example illustrates the dependency of the operators on the types of the arguments.

```
#usual addition#
> 1+2+3
     6
#adding components to a vector#
> [1]+2+3
[ 1 2 3 ]
> v=[1,2]
> v+3
[ 1 2 3 ]
> v=1+2
> v+3
     6
```

Some matrix operations:

```
> v=[1,2]
> m=[v]
> m
| 1 2 |
> *m
| 1 |
| 2 |
> *m*m
```

```
| 1 2 |
| 2 4 |
> m**m
| 5 |
```

3.11.2 Functions The next session exhibits the difference between functions with 0 arguments and variables

```
> x=3
> f=x
> f
     3
> x=4
> f
     3
> x=3
> f()=x
> f
     3
> x=4
> f
     4
```

A function to compute the order of a nonsingular matrix

```
> ord(mat m)=id=id(rowsize(m));ord_loc(m,1)
> ord_loc(mat p;int n)=if (p==id) then n else ord_loc(m*p,n+1) fi
> ord([[0,1],[1,0]])
2
```

Functions involving groups:

```
> setdefault(A3)
> cartan
|  2 -1  0 |
| -1  2 -1 |
|  0 -1  2 |
> quit
end program
```

3.11.3 Sets The intersection **meet**(v, v') of two sets of integers represented by the respective vectors v and v' can be found in the following straightforward manner:

```
meet(vec v,vp)= ans=null(0); for i=1 to size(v) do \
  if belongsto(v[i],vp) then ans=ans + v[i] fi od; ans

belongsto(int j; vec v) = yes = 0; for i=1 to size(v) do  \
  if j == v[i] then yes = 1; break fi od; yes
```

The standard function **sort** (cf. Chapter 4) can be used to write a speedier version. The union of two set operations can be dealt with likewise. See **redsetmat** in Chapter 4 for facilitating similar operations for sets of vectors.

LiE MANUAL

Chapter 4. STANDARD FUNCTIONS

In this chapter, we list the mathematical standard functions of LiE . With each function listed, we give an interpretation of its arguments and the result of its call; furthermore, whenever worthy of mention, a brief indication is given of the algorithm involved in its implementation. For terminology see Chapters 2 and 3.

For brevity, the function arguments have been named $n, n', n'', ...$ for integers, $m, m', m'', ...$ for matrices, $v, v', v'', ...$ for vectors, $g, g', g'', ...$ for groups, and $s, s', s'', ...$ for strings. Although, often, after **setdefault**(g) has been set, the last argument g can be omitted, we shall not separately itemize the functions with fewer arguments.

4.1. adams(n, v, g)

Returns the virtual multiplicity matrix of the virtual module whose dominant weight decomposition matrix m' is obtained from the dominant weight multiplicity matrix of v by multiplication by n of the weights (i.e., all entries but those in the last column). The adams operator is the 'weight analog' of the operator that, given a character χ of a group g and a number n, computes the decomposition of the class function $\gamma \mapsto \chi(\gamma^n)$ as an integral linear combination of irreducible characters. The adams operator is used in **plethysm**, **symtensor**, and **alttensor**. *Algorithm:* First, **mul** is applied to v, then the resulting weights are multiplied by n, and finally **vdecomp** comes into action.

4.2. addmul(m, m', g)

Returns the multiplicity matrix of the sum of the multiplicities given by the matrices m and m'. *Algorithm:* the rows of m and m' are put in a single matrix, to which **redmulmat** is applied.

4.3. adjoint(g)

Returns the highest weight of the adjoint representation of adjoint representation of the group g. The group has to be simple. [For semisimple groups: just perform adjoint for each component; add to each highest weight obtained the appropriate zero components; the direct sum of the representations with the highest weights thus obtained is the adjoint of the semisimple group. The adjoint representation of a torus Tn is the trivial one (corresponding to highest weight $[0, ..., 0]$ with n components).]

4.4. alttensor(n, v, g)

Returns the decomposition matrix of the n-th exterior power of the module with highest weight v. See also **symtensor**.

4.5. branch(v, g', m, g)

Returns the decomposition matrix of the restriction to g' of the g-module of highest weight v with respect to the restriction matrix m. For fundamental Lie subgroups (among which the Levi) this matrix can be gotten by use of **resmat**. Branching to T amounts to **mul**. *Algorithm:* The weights of the module with highest v are generated, one at a time; to each weight the matrix m is applied; if the result is a dominant weight of g', it is appended as a row to a matrix a. Finally the matrix a is subjugated to **decomp** with respect to g'. We repeat that LiE generates the weights one at a time, using Freudenthal's Multiplicity Formula and a dynamic version of **worbit** to prevent storage problems.

4.6. cartan(g)

Returns the Cartan matrix of g. If g is simple, the labelling is Bourbaki's, see [Bourb 1968].

4.7. cartan(v, v')

Returns the 'Cartan product' $< v, v' >$ of the roots v and v', that is, the integral value $2(v, v')/(v', v')$. [Not really an inner product because the function is not linear in the second factor.] Related to **inprod** and **norm**.

4.8. carttype(m, g)

Returns the semisimple group (or rather its type) generated by the root groups whose roots are given in the matrix m of roots in fundamental shape (i.e., all roots are positive and mutually have obtuse angles). Useful in connection with **fundam** and **closure**. See also **centrtype**.

4.9. center(g)

Returns a matrix whose rows are semisimple elements generating the centre of g. [In most simple cases it could have been a vector, but not if the type is D_{2n}.]

4.10. centroots(v, g)

Returns the matrix whose rows form the full set of positive roots centralizing the semisimple elements represented by the vector v. Here a root $\alpha \in \Phi$ is said to *centralize* v if v commutes with all elements of the fundamental subgroup of type A1 and closed subsystem of roots $\{\alpha, -\alpha\}$. *Algorithm:* first **posroots**, then, a row v' belongs to the result of **centroots** if the standard inner product of v' and v is 0 modulo n.

4.11. centroots(m, g)

Returns the matrix whose rows form the full set of positive roots centralizing the semisimple elements of T represented by the rows of m. Used in conjunction with

carttype or **fundam**, gives the type, resp. a nice presentation of the centralizer. See also **centrtype**.

4.12. centrtype(v, g)

Returns the Lie group type of the centralizer of the semisimple element v of T. See also **centroots**. [Actually the centralizer (although connected) need not be simply connected, so the interpretation of the type **grp** of section 3.2.4 does not admit a precise description of the actual centralizer; the result refers to the unique simply connected group C covering the centralizer subgroup (in other words, there is a finite central subgroup Z of C such that the precise centralizer is isomorphic to the quotient C/Z of C by Z.]

4.13. centrtype(m, g)

Returns the Lie group type of the centralizer of the semisimple elements of T represented by the rows of m. *Algorithm:* The algorithm uses **centroots**; firstly, it isolates simple components; then in most cases it recognizes the type from the size of these components. It can be tested independently by using **carttype**(**fundam**(**centroots** (v, g), g), g). [The check makes sense as the routines with which the Lie groups are found differ entirely]. (A pre-LiE version of this function, only implemented for En, has been used for [CohGri 1987].)

4.14. closure(m, g)

Returns a (root) matrix whose rows are the fundamental roots of the smallest closed subsystem of the root system Φ of g containing all roots of the root matrix m (i.e. rows of m are roots). It uses **fundam**. If the diagram of g has double bonds, the latter functions is applied twice, the second time after roots have been added that are sums of roots of the system generated by the result of **fundam**(m).

4.15. collect(m, g', m', g)

Returns the decomposition matrix m'' of the g-module whose restriction (branching) to the reductive subgroup g' with respect to the restriction matrix $r = (m')^{-1}$ has decomposition matrix m. (This operation makes sense if m' is invertible; in particular we must have **lierank**(g') = **lierank**(g).) Thus, in a way it is the inverse of **branch**, and the result can be checked by performing **m == branchr**(m'', g', r, g), where **branchr** is as in Example 5.8.2.

4.16. contragr(v, g)

Yields the highest weight of the contragredient of the representation with highest weight v.

4.17. decomp(m, g)

Returns the decomposition matrix of the g-module whose multiplicity matrix coincides with m.

4.18. **deg**(v, g)

Returns the degree (= dimension) of the representation of g with highest weight v.

4.19. **detcartan**(g)

Returns the determinant of the Cartan matrix of g. See also **cartan** and **icartan**.

4.20. **diagram**(g)

Prints the Dynkin diagram of g with the labeling used by Bourbaki.

4.21. **dim**(g)

Returns the dimension of the Lie group g.

4.22. **dominant**(v, g)

Returns the dominant weight in the g-orbit of the weight v.

4.23. **domweights**(v, g)

Returns a matrix whose rows are the dominant weights which lie under v. Here v' is said to *lie under* v if $v - v'$ is a sum of positive roots with non-negative integer coefficients.

4.24. **exponents**(g)

Returns the exponents of the given Lie group. (The number of exponents equal to 0 is the dimension of the central torus.)

4.25. **factor**(n)

Prints the factorization of n into prime factors. (Returns a **vid**, i.e. nothing.) Only prime factors smaller than 2^{15} are found.

4.26. **fundam**(m, g)

Returns a fundamental set of roots for the subsystem generated by the set of roots that form the rows of m. *Algorithm:* first, all roots in m are made positive; then each pair of roots that is not fundamental in the plane they span is replaced by a fundamental pair of that plane, and **setredmat** is applied to the result; this step is repeated till no more changes occur.

4.27. highroot(g)

Returns the highest root of the root system of g. If g is simple, this root is the last row of **posroots**(g).

4.28. icartan(g)

Returns the inverse of the cartan matrix of the simple group g multiplied by the determinant of the Cartan matrix in order to keep all matrix entries integral.

4.29. length(v, g)

Returns the length of the Weyl group element with coxeter word v, that is, the minimal length of an expression of $v[1]\,v[2]$... as a product of fundamental reflections. If v is reduced (i.e. $v =$ **reduce**(v, g)), then this is just the size of the vector v. The algorithm depends on **reduce**.

4.30. liecode(g)

Returns a vector with 2 components n, n' such that **liegroup**(n, n') $= g$.

4.31. liegroup(n, n')

Returns a torus or simple group according to the following rule: **liegroup** $(0, n')$ $=$ Tn', **liegroup** $(1, n')$ $=$ An', **liegroup** $(2, n')$ $=$ Bn', **liegroup** $(3, n')$ $=$ Cn', **liegroup** $(4, n')$ $=$ Dn', **liegroup** $(5, n')$ $=$ En', **liegroup** $(6, 4)$ $=$ F4, **liegroup** $(7, 2)$ $=$ G2. This may help to run examples over many simple Lie groups using a `for` loop. The inverse of this function is **liecode**.

4.32. lierank(g)

Returns the Lie rank of g.

4.33. longword(g)

Returns the longest element of the Weyl group with respect to the length (as a product of fundamental reflections).

4.34. lreduce(v', v, g)

where v is a Weyl word and v' a set S of indices of fundamental reflections (elements are the components of v') returns a Weyl word representing the distinguished left coset representative of $W_S v$.

4.35. lrreduce(v', v, v'', g)

where v is a Weyl word and v' and w'' represent sets S and S', respectively, of fundamental reflections returns the a Weyl word representing the distinguished double coset representative of $W_S w W_{S'}$.

4.36. **mul**(v, g)

Returns the multiplicity matrix of the g-module with highest weight v. *Algorithm:* Freudenthal's multiplicity formula, see [Hum 1972] and [Kruse 1971].

4.37. **mul**(v, v', g)

Returns the multiplicity of v' in the g-module with highest weight v.

4.38. **nextpart**(v)

Returns the next partition of `sum` (v) in lexicographical order. If v is the last one, it will return v again.

4.39. **nextpermu**(v)

Returns the next permutation of v in lexicographical order. If v is the last one, it will return v again.

4.40. **norm**(v, g)

Returns the norm (v, v) of the root v.

4.41. **numproots**(g)

Returns the number of positive roots of the root system of g, that is, **rowsize** (**posroots**(g)). The number of roots is twice as much.

4.42. **orbit**(v, m)

Here v is a vector and m is a single $n1 \times n2$-matrix which has to be interpreted as a collection M of $n1/n2$ square matrices of dimension $n2$; the first consists of the first $n2$ rows of m, and so on. The algorithm tries 1000 images of v and stops if it does not succeed closing up. To change the default limit, use the next function.

4.43. **orbit**(n, v, m)

As **orbit**(v, m), but now with 1000 replaced by n.

4.44. **plethysm**(v, v', g)

Returns the decomposition matrix of the g-module obtained from the module with highest weight v' by taking the symmetrized tensor with respect to the partition v. For example **plethysm**$([2], v, g)$ has the same effect as **symtensor**$(2, v, g)$ and **plethysm**$([1, 1], v, g)$ should lead to the same result as **alttensor**$(2, v, g)$ [it makes sense to check: the algorithms differ]. *Algorithm:* the classical Frobenius Formula (cf. [And 1967] and [JamKer 1981])

$$\mathbf{plethysm}([v],\lambda)=\frac{1}{d!}\bigoplus_{p} nclass(p,d)\,\chi^{v}(p)\bigotimes_{i=1}^{size(p)}\mathbf{adams}(p[i],\lambda),$$

where p runs over all partitions of $d=v[1]+..+v[size(v)]$, the number $nclass(p,d)$ counts the elements of the symmetric group on d letters in the conjugacy class with partition (=cycle pattern) p, and χ^{v} is the irreducible character of the symmetric group on d letters corresponding to the partition v. Thus, the algorithm uses **adams**, **addmul**, **tensor**, and **symchar**.

4.45. posroots(g)

Returns a matrix whose rows are the positive roots of g with respect to a fixed torus and a fixed ordering. The first rows are the fundamental roots and the last row is **highroot**(g). **numproots**(g) gives the number of rows.

4.46. ptensor(n, v, g)

Returns the decomposition matrix of the n-th tensor of the irreducible g-module with highest weight v.

4.47. ptensor(n, m, g)

Returns the decomposition matrix of the n-th tensor of the irreducible g-module with decomposition matrix m.

4.48. redmulmat(m)

(*abbreviation of* reduce - multiplicity - matrix) Returns the matrix representing the same multiplicity matrix as m, but sorted according to the lexicographical ordering of rows on all but the last component, and having at most one row with specified entries in all but the last component. It thus provides a canonical form for multiplicity matrices.

4.49. redsetmat(m)

(*abbreviation of* reduce - set - matrix) Returns a canonical form for a set matrix: it returns a matrix with lexicographically ordered rows and representing the same set of vectors as m; it is obtained from m by sorting rows using **sortmat** and deleting doubly occurring rows.

4.50. reduce(v, g)

Returns a shortest vector w such that v and w are Weyl words representing the same element of W. See also **lreduce** en **rreduce** and **lrreduce**. *Algorithm:* uses the action of W on the weight lattice.

4.51. reflection(v, g)

Returns the reflection in the Weyl group W of g with root v with respect to the basis of fundamental roots.

4.52. resmat(m, g)

Returns the restriction matrix of the Lie subgroup whose root system has fundamental system as given in m.

4.53. rreduce(v, v', g)

where v is a Weyl word and v' is a vector representing a set S of fundamental reflections returns a Weyl word representing the distinguished coset representative of vW_S. See also **lrreduce** and **reduce** and **lreduce**.

4.54. setdefault(g)

Set default Lie group equal to g. This Lie group is used in other functions if the optional argument g is omitted. The command **setdefault** lets you know which Lie group is the default.

4.55. sort(m)

Returns the matrix obtained from m by sorting according to the lexicographical order of integer vectors, the biggest one first. *Algorithm:* quicksort.

4.56. sort(v)

Returns the vector obtained from v by sorting according to the usual order of integers, the biggest one first. *Algorithm:* quicksort.

4.57. spectrum(v', v, g)

Returns the vector whose j-th component contains the multiplicity of the eigenvalue $e^{2\pi i j/v[r+1]}$ of the semisimple element v on the module with highest weight v'. *Algorithm:* uses **mul** and **worbit**. (A pre-LiE version of this function, only implemented for En, has been used for [CohGri 1987].)

4.58. symchar(v)

(symmetric group character) Returns the character matrix of the character of the symmetric group on n letters, where $n = v_1 + v_2 + \ldots$, corresponding to the partition v. *Algorithm:* uses **partitions** and the next function.

4.59. **symchar**(v, v')

(symmetric group character) Returns the value (an integer) of the character of the symmetric group on n letters, where $n = v_1 + v_2 + \ldots$, corresponding to the partition v on the conjugacy class corresponding to partition v'. *Algorithm:* uses the formula that expresses the character as an alternating sum of characters of permutation representations on sets of flags, see [JamKer 1981].

4.60. **symorbit**(v)

(symmetric group orbit) Returns the matrix obtained from v by letting each row be a permutation of the components of v, each permutation occurring once and the rows being ordered lexicographically, the biggest one first.

4.61. **symtensor**(n, v, g)

(symmetric tensor) Returns the decomposition matrix of the n-th symmetric tensor of the irreducible g-module with highest weight v. *Algorithm:* uses the recursion

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

This formula turns into a recursion formula for **alttensor** upon changing the sign of the k-th summand to $(-1)^{k-1}$ for each $k \in \{1, \ldots n\}$.

4.62. **tensor**(v, v', g)

Returns the decomposition matrix of the tensor product of the g-representations with highest weights v and v'. *Algorithm:* Klimyk's formula has been implemented, see [Hum 1972, Exerc. 24.9]. To run over an orbit of the Weyl group, we use a dynamic version of **worbit** to prevent storage of the whole orbit.

4.63. **tensor**(v, v', v'', g)

Returns the multiplicity of the weight v'' in the tensor decomposition of the tensor of the g-modules with highest weights v and v'.

4.64. **tensor**(m, m', g)

Returns the decomposition matrix of the tensor of the two g-modules with respective decomposition matrices m and m'.

4.65. **vdecomp**(m, g)

(virtual decomposition) Returns the virtual decomposition matrix of the virtual module with multiplicity matrix m. The algorithm is very similar to the usual one for **decomp**. This function is used in **adams**.

4.66. **waction**(v, v', g)

(Weyl action) Returns the weight that is the image $v \cdot w$ of the weight v under the element $w \in W$ corresponding to the Weyl word v'.

4.67. **worbit**(v, g)

Returns the matrix whose rows form the orbit of the weight v under the Weyl group of g. *Algorithm:* for the symmetric group, the Weyl group of An, after suitable linear transformations, a loop is set up, starting with **sortvec**(v), and using **nextpermu** to create the orbit. For the Weyl groups of Bn, Cn and Dn, a similar procedure is followed. For the exceptional groups (En,F4, G2), a big subgroup U of the Weyl group W is chosen that is the Weyl group of a Lie subgroup of type A,B,C or D (or a product of simple ones of this form), as well as a specified set of coset representatives of this subgroup in the Weyl group W. The coset representatives are used to create representatives of the orbits into which the W-orbit splits after restriction to U; then, the U-orbit of each representative is constructed as above. This algorithm is much faster than the general function **orbit**. The latter can be used to check results of moderate size, using **reflection** to obtain a matrix presentation of the fundamental reflections.

4.68. **worbitsize**(v, g)

Returns the length of the orbit of the weight v under the Weyl group of g.

4.69. **worder**(g)

Returns the order of the Weyl group of g.

4.70. **worder**(v, g)

Returns the order of the subgroup of the Weyl group of g generated by the fundamental reflections whose indices occur in the set v.

4.71. **wrtaction**(v, v', g)

Returns the image vector $v \cdot w$ of the root v under the element of the Weyl group W of g corresponding to the Weyl word v'.

4.72. **wrtorbit**(v, g)

Returns the root matrix of the orbit of the root vector v under the Weyl group of g.

4.73. **wword**(m, g)

Returns a Weyl word such that the product of fundamental reflections involved is an element of the Weyl group representing the square matrix m of dimension the

Lie rank of g. The inverse of this function can be written in the interpreter by use of **reflection**.

4.74. wword(v, g)

Returns a Weyl word corresponding to an element of the Weyl group of g sending the weight v to a dominant weight. In fact, the returned Weyl word w is the unique left reduced one in the coset $W_S w$, where W_S is the stabilizer of **dominant**(v). (Thus, S consists of the (indices of the) fundamental reflections fixing the dominant vector.)

LIE MANUAL
Chapter 5. EXAMPLES

We provide some examples of how we use or intend LiE to be used for computations.

5.1. General

5.1.1 Ordering The standard function **sort** sorts the entries of a vector v from big to small. The inverse ordering is obtained by

```
-sort(-v)
```

5.1.2 Union of vectors Suppose $m1$ and $m2$ are matrices representing sets of vectors. Then

```
redsetmat(m1^m2)
```

returns the matrix whose rows are the vectors in the union of the two.

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

```
 sum(vec v)=loc ans=0; for i in v do ans=ans+i od; ans
prod(vec v)=loc ans=1; for i=1 to size(v) do ans =v[i]*ans od; ans
```

5.1.4 Comparing groups The operator == is not defined for groups, the following function is a LiE shell version.

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

5.2. Roots

Here are a few simple examples of how to get to know a root system.

5.2.1 All roots The function **roots**(g) defined as follows

```
roots(grp g) = m = posroots(g); m ^ ((-1)*m)
```

returns the matrix containing all roots of the root system of the group g.

5.2.2 numproots The number of positive roots is a standard function. It can be simulated by `rowsize(posroots())` for the default group.

5.2.3 half sum of positive roots The expression

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

occurs in Weyl's character formula, and several other expressions. We construct a function **rho** to obtain it in LiE for the root system Φ determined by g.

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

5.2.4 inprod(v, v') Is a standard function returning the inner product (v, v') of the roots v and v'. See section 2 for the definition of the inner product on the root space. It can be simulated by using the norm on shell level.

5.3. Highest root of a nonsimple system

The function **highroot** has only been defined for simple groups g. If g is nonsimple, application of the following function gives the highest root.

```
highr(grp g) = h=[]; \
  for i=1 to compsize(g) do h2=highroot(g[i]); h= h^h2 od; \
 h^(null(liecode(g[0])[2]))
```

5.4. Weyl words

5.4.1 Length The length of a weyl group element represented by the Weyl word w is a standard function. It can, however, be gotten as **lengthprime(w)** defined as follows:

```
length(vec w) = size(reduce(w))
```

provided a default group has been set.

5.4.2 From a weyl word to a weyl group element

The standard function **wword** transforms a weyl word into the corresponding weyl group element. For the inverse function, the standard functions **reflection** is useful:

```
#welt is short for weyl element#
welt(vec w)=ans=reflection(w[1]); \
for i= 2 to size(w) do ans = ans * reflection(w[i]) od; ans
```

5.4.3 The Coxeter matrix The Coxeter matrix of a Weyl group is the matrix $(m_{i,j})_{1\leq i,j\leq s}$ where $m_{i,j}$ is the order of the product $r_i r_j$ of the fundamental reflections r_i and r_j. Here is a (rather inefficient) way to compute it.

```
coxmat()= m=id(lierank()); \
for i=1 to rowsize(m)-1 do for j=i+1 to rowsize(m) do \
  m[i,j] = ord(fund_refl(i) * fund_refl(j)) ; \
  m[j,i] = m[i,j] od od;m
fund_refl(int n) = rt=null(lierank()); rt[n] = 1; reflection(rt)
```

Here `ord(mat m)` is as defined in 3.10.2.

5.4.4 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,...] of length m, where m is the order of the product $r_i r_j$ of the corresponding fundamental reflections, by [i,j,i,...] of the same length. The routine **nextrewrite** produces this replacement (if possible) in the weyl word v for the subword that begins at the k-th component of v.

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

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

The function coxmat is as in the previous section.

5.4.5 The Bruhat ordering The following function **bruhat**(v) returns a weyl word for each weyl group element that is covered by v in the so-called Bruhat order.

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

rmnull(vec v)= if size(v) == 0 then [] else \
     if v[size(v)] == 0 then rmnull(v-size(v)) else v fi fi
```

5.5. Cosets in The Weyl group

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

5.5.1 Right cosets Suppose S is a subset of $\{1, ..., r\}$ and W is a Weyl group of rank r. Then there is a natural system of representatives of the cosets of the Weyl subgroup in W generated by the fundamental reflection with index in S. This is the set of all distinguished right coset representatives, that is, all elements of W that have right reduced Weyl words with respect to S. Here is how to print this set:

```
charv(vec w)= y=diag(id(lierank()));\
for i=1 to size(w) do y[w[i]]=0 od; y
rcosets(vec w) = for x row worbit(charv(w)) \
do print(wword(x)) od
```

Again, before invoking the command `rcosets(w)`, a default group has to be set; for example, after the above definitions of `charv` and `lcosets` have been read, the left coset representatives for the subsystem $A1A1A1$ in $D4$ can be found by the session:

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

Note that [1,3,4] represents the nodes corresponding to the subsystem A1A1A1, as can be verified by performing `diagram(D4)`. Another - more elaborate - way to verify this is to ask for the Cartan type of the subsystem generated by the fundamental roots with indices $1, 2, 4$:

```
setdefault(D4); m= [[1,0,0,0],[0,0,1,0],[0,0,0,1]]
carttype(m)
```

5.5.2 Left cosets Using that a weyl word v is left reduced with respect to the subset w of $\{1, ..., s\}$ if and only if its inverse $[v[size(v)], v[size(v)-1], ..., v[1]]$ is right reduced with respect to w, we can write the following variation to the previous example to obtain a print of the list of right coset representatives:

```
inverse(vec v) = vinv=v; for i=1 to size(v) do \
vinv[size(v)-i+1] = v[i] od;vinv
lcosets(vec u) = for x row worbit(charv(inverse(u))) \
do print(inverse(wword(x))) od
```

where `charv` is as before.

5.5.3 Double cosets We now construct a function **dcosets**(v, w, g) printing the full set of distinguished double coset representatives, displayed as left, right reduced weyl group words, with respect to the subsets v and w of $\{1, ..., r\}$. Then modify `rcosets` such that it only prints those (right w-reduced) weyl words that are left reduced with respect to v.

```
dcosets(vec v,w )= \
for x row worbit(charv(w))  do wrd=wword(x);\
if length(wrd) == length(lreduce(v,wrd)) \
then print(wrd) 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. This can be achieved by appending the dummy entry 0 sufficiently often. A good upper bound for the number of columns needed is `lrreduce(v,longword(),w)`.

5.6. Semisimple elements

In LiE , a semisimple element of g is represented by a vector $[a_1, ..., a_r, a_{r+1}]$. This vector corresponds to the element $\prod_{i=1}^{r} h_i(\lambda^{a_i})$ of the maximal torus T. (Here $h_i(\lambda)$ is the unique element of T satisfying $h_i(\lambda)^{\omega_j} = \lambda^{\delta_{i,j}}$ for all j.)

5.6.1 $GL(n, \mathbb{C})$ For the linear group $SL(n, r+1)$ there is a much more familiar way to describe a semisimple element, namely as a diagonal element in the standard representation. If h is a diagonal element with entries $(\lambda^{b_1}, ..., \lambda^{b_{r+1}})$ in the standard representation then the corresponding torus representation is

$$h_1(\lambda^{b_1})h_2(\lambda^{b_1+b_2})...h_r(\lambda^{b_1+\cdots+b_r}).$$

Thus, the semisimple element with exponents $[b_1, ..., b_{r+1}]$ of λ along the main diagonal in the standard representation can be created by the following function **mksselt** (*abbreviation* of make semisimple element), valid only for g=An.

```
    mksselt(vec b; int ord)= ss=null(size(b)-1); ss[1]=b[1]; \
    for i=2 to size(ss) do ss[i] =ss[i]+b[i] od; ss^[ord]
```

Here λ is understood to be $e^{2\pi i/ord}$.

5.6.2 $SO(12, \mathbb{C})$ Here is yet another example, now with the orthogonal group D6. Consider the standard representation of the orthogonal group $SO(12, \mathbb{C})$ (note: this is not a faithful representation: a central element of the simply connected group maps to the identity). We fix a basis $e_1, ..., e_6, f_1, ..., f_6$ of the underlying 12-dimensional complex vector space with respect to which the bilinear form $\langle \cdot, \cdot \rangle$ fixed by $SO(12, \mathbb{C})$ is (a nonzero multiple of) satisfies $\langle e_i, e_j \rangle = \langle f_i, f_j \rangle = 0$ and $\langle e_i, f_j \rangle = \delta_{i,j}$ for all $i, j \in \{1, ..., 6\}$. Suppose now that $h \in SO(12, \mathbb{C})$ is given by the diagonal matrix with diagonal entries $[a_1, a_2, ..., a_6, a_1^{-1}, a_2^{-1}, ..., a_6^{-1}]$. Then the matrix m in the following command transforms $[a_1, a_2, ..., a_6]$ into the first part (all but the last component) of the canonical form for a semisimple element.

```
    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]]
```

5.6.3 spectrum The function **spectrum** provides a means to recognize the semisimple element specified in a more natural form. For instance, the session

```
    > setdefault(A4)
         A4
    > h=[1,0,0,0,2]
    > spectrum([1,0,0,0],h)
    [ 3 2 ]
```

shows that h is an element of order 2 with precisely 2 eigenvalues -1 in the standard representation. Thus it is conjugate to `(mksselt([0,0,0,1,1]))^[2]`, with `mksselt` as above. The centralizer of h can be found by the command `centrtype(h)` On the other hand, `spectrum(adjoint(),h)[1]` returns the dimension of the Lie subalgebra of the Lie algebra fixed by h (viewed as an automorphism of the Lie algebra of g), which is the Lie algebra of the centralizer of h. Thus we have the check

```
    spectrum(adjoint(),h)[1] == dim(centrtype(h))
```

5.6.4 centrtype Continuing with the semisimple element of the preceding paragraph,

```
    > centrtype(h)
         A2A1T1
```

```
> centroots(h)
| 1 0 0 0 |
| 0 0 1 0 |
| 0 0 0 1 |
| 0 0 1 1 |

> r=resmat(closure($))

> r
| 0 0 1 |
| 0 0 0 |
| 1 0 0 |
| 0 1 0 |

> carttype(closure(centroots(h)))
     A2A1

> branch([1,0,0,0],A2A1,r)
| 1 0 0 1 |
| 0 0 1 1 |
```

5.7. Checks

The standard functions are chosen so as to create possibilities to cross check results.

5.7.1 Dimension The standard function **dimension**(g) gives the dimension of the Lie group g and so coincides with the dimension of the Lie algebra, i.e. with the **degree** of the **adjoint** representation. Thus we must have response 1 to the command

```
    deg(adjoint) == dim
```

after setting the default group equal to g.

5.7.2 Multiplicities This is one of the main reasons why a function like **mul** has been supplied. It helps to test the outcome of **plethysm**, **tensor** and **branching**. But for **mul** itself, checks are conceivable as well, to wit, by the following formula:

$$\sum_{\mu\in\Lambda^+(T),\mu\leq\lambda} \mathbf{mul}(\lambda,\mu))\cdot \mid W/W_\mu \mid = \mathbf{deg}(\lambda).$$

5.7.3 Branching Similarly, for branching, we have

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

where h is a reductive subgroup of g with restriction matrix m. Since the multiplicity **mul** (**mat** m; **vec** v) of the weight v in the g-module with decomposition matrix m has not been implemented as a standard function, this has to be programmed in the interpreter language.

5.7.4 Tensors A function to check **alttensor** and **symtensor** against **ptensor**:

```
    check2(vec v)=al=alttensor(v);sy=symtensor(v); \
    ans1=addmul(al,sy); ans2 = redmulmat(ptensor(2,v)); \
    if ans1 == ans2 then print("OK") else print("failure")
```

and one to check against **plethysm**:

```
    checkal(int p; vec v)=al=alttensor(p,v);av=diag(id(p)); \
    if al == redmulmat(plethysm(av,v)) then \
          print("OK") else print("failure")
```

and similarly for symtensor, with `av=[p]`.

5.8. Branching

5.8.1 Branching semisimple groups Suppose h is a closed simple subgroup of $g = g' \times g''$. Branching the g-module with highest weight $v + w$, where $v \in \Lambda(T \cap g')$, $w \in \Lambda(T \cap g'')$ are highest weights of g' and g'' respectively, to h amounts to taking the tensor product of the h-modules obtained from the g-module with highest weigth v and the g'-module with highest weight w by branching each to the projection of h onto the respective components of g. Thus it is possible to extend the function **branch** to non-simple reductive groups.

5.8.2 Branching reducible modules Suppose we want to branch a module M from the default group to the group h with restriction matrix r. If M is irreducible with highest weight v, then `branch(v,h,r)` does the job. Otherwise, put the highest weights of the M in a matrix, each row represent a single irreducible constituent (so m is no a decomposition matrix for M), and run **branchr** defined as follows

```
    branchr(mat m;grp h;mat r; grp g) = ans=branch(m[1],h,r,g); \
      for i=2 to rowsize(m) do \
      ans = addmul(ans,branch(m[i],h,r,g)) od; ans
```

5.9. Overflow

Due to the choice of type **int** rather than **bin** for matrix and vector entries, vector and matrix operations leading to big integer entries are not to be trusted. In the example below, the VAX/780 returned an 'orbit' of length 64, due to its arithmetic modulo a power of 2.

```
    r=reflection([1,1,1,1],D4)
    addmul(r,r)
    orbit([1,0,0,0],$)
```

5.10. Maximal semisimple subgroups

Information on subgroups of a Lie group can be stored on a file and read whenever convenient. We have begun such a documentation by defining two functions, one for the subgroup information, the other for the restriction matrices (needed for **branch**).

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

```
    levimat(int i) = m=null(lierank()-1,lierank()); \
    for j= 1 to i-1 do m[j,j]=1 od; \
    for j=i to lierank()-1 do m[j,j+1]=1 od; closure(m)

    levitype(int i) = carttype(levimat(i))

    levidiag(int i) = diagram (levitype(i))

    levires(int i) = resmat(levimat(i))

    levibranch(vec v; int i) = m=levimat(i); gp = carttype(m); \
      r=resmat(m);m=0; branch(v,gp,r)
```

It will be clear that **levibranch** gives the decomposition matrix of the Levi subgroup of type **levitype**. The diagram prompted by **levidiag** give the ordering of the fundamental roots of the Levi subgroup with respect to which the restriction matrix (returned by **levires**) and the resulting decomposition matrix have been given.

5.10.2 maxsub(g) prints a list of isomorphism types of non-maximal rank maximal semisimple subgroups of g (We do not guarantee that the list is complete!). Also **resmat(g, g', n)** returns the restriction matrix of the n-th maximal subgroup g' (with the same type as g') of g, where the ordering is as in the result of **maxsub**. (Useful in conjunction with **maxsub** and **branch**.) We only give this example for the cases $g =$ E6, E7, E8, F4, G2.

```
    off gc
    # Since garbage collection can be time consuming and does not  #
    # make sence during reading this file in is better to turn it  #
    # off. But do not forget to put a on gc at the and of the file.#

    # Global variables: #
    stackgroup=T0
    resmatgroup=T0
    nsubgr=0
    sta=T0;stb=T0;stc=T0;std=T0;ste=T0;stf=T0;stg=T0
    rga=[[]];rgb=[[]];rgc=[[]];rgd=[[]];rge=[[]];rgf=[[]];rgg=[[]]

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

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

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

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

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

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

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

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

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

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

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

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

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

LiE MANUAL

Chapter 6. REFERENCES

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

[And 1977] C.M. Andersen, *Clebsch-Gordan series for symmetrized tensor products,* J. Math. Phys., **8**(1977), 988-997.

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

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

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

[Brem ea 1985] M.R. Bremner, R.V. Moody, J. Patera, *Tables of dominant weight multiplicities for representations of simple Lie algebras,* Monographs and Textbooks in Pure and Appl. Math. 90, Dekker, New York, 1985.

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

[CohGri 1987] A.M. Cohen & R.L. Griess, *On finite simple subgroups of the complex Lie group of type E_8*, pp. 367-405 in: Proc. of Symp. in Pure Math. 47[2] (eds.: P. Fong), Amer. Math. Soc., Providence, 1987.

[Hum 1974] Humphreys, J.E., *Introduction to Lie algebras and representation theory,* Springer, New York, 1974.

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

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

[McKay ea 1981] W.G. McKay & J. Patera, *Tables of dimensions, indices and branching rules for representations of simple Lie algebras,* Lecture Notes in Pure and Appl. Math. 69, Dekker, New York, 1981.

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

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

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

LiE MANUAL

Chapter 7. INDEX

```
name
        Pre - an input line editor.
synopsis
        Pre program
description
        Pre is an input line editing preprocessor with history,
        based on the command-line editor of the Korn shell.
        It is designed as an aid for programs which ask for
        keyboard input, but have poor editing facilities themselves.
        Interrupts are ignored.

Using Pre in a LiE session
        You can use LiE with and without Pre. By default it is available,
        however if you want to run LiE without Pre you can turn it of by
        editing the shell script LiE as indicated in that script.
        Type ESCape to enter edit mode, where the following edit commands
        are supported:

        Cursor Motion
                l, space  ; One position to the right.
                h, backsp ; One position to the left.
                w         ; To the beginning of the next word.
                W         ;  ... , including punctuation.
                b         ; To the beginning of the previous word.
                B         ;  ... , including punctuation.
                e         ; To the end of the word.
                E         ;  ... , including punctuation.
                $         ; To the end of the line.
                ^, 0      ; To the beginning of the line.
                fc        ; To the next character c.
                Fc        ; To the previous character c.
                ;         ; Repeat f or F.
                ,         ;  ... , but in the reverse direction.

        Inserting Text
                i         ; Insert text.
                I         ; Insert text at the start of the line.
                a         ; Append text.
                A         ; Append text at the end of the line.
                        Type ESCape to entermode again.

        Deleting Text
                x         ; Delete character.
                X         ; Delete character before the cursor.
                dd        ; Delete the whole line.
                dw        ; Delete the rest of the word.
                dW        ;  ... , including punctuation.
                D         ; Delete the rest of the line.

        Changing Text
                ~         ; Reverse case and advance.
                rc        ; Exchange character with c.
                cc        ; Change the whole line.
                cw        ; Change the rest of the word.
                cW        ;  ... , including  punctuation.
                C         ; Change the rest of the line.
                        Type ESCape to entermode again.



        History
                +, j      ; Get the next history line.
        -, k       ; Get the previous history line.
        /string    ; Searchs backwards in history for
                          the following string.
        ?string    ; Searchs forwards in history for
                          the following string.
        n          ; Repeat previous search command.
        N          ;  ... , but in the other direction.

Undo
        u          ; Undo the last change.
        U          ; Restore the whole line.

Yank
        yw         ; Yank a word.
        yW         ;  ... , including punctuation.

Paste
        p          ; Paste what is yanked by yw, yW, dw, dW,
                          D, x and X, after the cursorposition.
        P          ;  ... , on the cursorposition.

Miscellaneous
        .          ; Repeat the previous insert, append, change,
                          delete or paste command.
        !          ; Pass the following line to the shell.
        return     ; Return the current line.
        qQ         ; Quit.
        Cntrl_L    ; Redraw the line.
        \          ; Return the current line after
                     having returned the immediately
                     preceding lines from history
                     ending with an \.
                     For example, type the following:
                          Pre cat
                          abcd\  <CR>
                          efgh\  <CR>
                          ijl\   <CR>
                          ESC
                          -
                          e
                          i
                          k
                          ESC
                          \
                     and watch to be printed:
                          abcd\
                          efgh\
                          ijkl\
```