Max-Planck-Institut
für Mathematik
in den Naturwissenschaften
Leipzig

SuperLie. A package for Lie algebra and Super Algebra computations in *Mathematica* software system

by

Pavel Grozman, and Dimitry A. Leites

Technical Report no.: 7 2006

SuperLie

A package for Lie Algebra and Super Algebra computations in *Mathematica* software system

P. Grozman

■ Abstract

SuperLie is a Mathematica–based package designed for solutions of scientific and computational problems related to Lie algebras and Lie superalgebras, their q–deformations included. Using **SuperLie** one can construct objects habitual for the mathematician (vector spaces and superspaces, algebras and modules over these algebras) in a way (hopefully) accessible to the engineer. It can solve various applied problems and theoretical problems of considerable importance to the physicists. In particular, **SuperLie** allows one to perform calculations and symbolic transformations in order to determine generators and relations, vacuum vectors (highest and lowest), compute homology and cohomology; calculate the Shapovalov determinant, and so on. It is possible to output the result in TeX format.

■ Disclaimer

■ Download address

The documentation and the package can be downloaded from

www.equaonline.com/math/SuperLie/

For a concise background necessary to start programming and a list of problems both already solved and to be solved with the help of SuperLie as well as a brief comparison with the existing related packages known to us, see P.Grozman and D.Leites, **SuperLie** and problems (to be) solved with it. Preprint MPI–2003–39 (www.mpim–bonn.mpg.de) referred to in what follows as [GL]. An updated version of [GL] will be regularly put on www.equaonline.com/math/SuperLie/

■ Acknowledgements

I am thankful to all individuals who helped me and all organizations that supported this work since 1992: NFR (1991, 1992), The Swedish institute (1993–95), Equa Simulation AB and TBBS, Stockholm (2001–2002), the Department of Mathematics of Stockholm University (since 1991).

I am particularly thankful to Professors D.Zagier and B.Julia for their help and enthusiasm in displaying the package at Max–Planck–Instutute für Mathematik, Bonn and Ecole Normale Superieur, Paris, respectively.

I am thankful to I.Shchepochkina, A.Sergeev, P.Etingof, M.Kontsevich for interesting problems to solve. A.Shapovalov wrote the first draft of the Introduction to **SuperLie**. Thanks are due to A.Shapovalov and O.Denisenkova a year–long self–sacrificing usage of **SuperLie** during which several bugs were discovered.

The whole project was launched and *super*wised by D.Leites as unexhausible source of problems partly listed in [GL].

SuperLie Contents

SuperLie: a Fast Introduction

■ Purpose

The **SuperLie** Package is a tool for calculation of various data for Lie algebras and superalgebras. Using **SuperLie** Package you can

- make the numeric and symbolic calculations with vectors;
- build vector (super)spaces, algebras and modules over algebras;
- convert the results to TeX format.

■ Principles

■ Vectors and vector expressions

□ Vectors and scalars

Vectors in **SuperLie** are:

- Symbols declared as vectors; such symbols with indices, as v_i or `v[i, j]`;
- Linear combinations of vectors with scalar coefficients;
- Vector expressions, like $v_1 \otimes v_2$;
- The results of vector–valued functions and operators.

Scalars in **SuperLie** are:

- Symbols declared as scalars; such symbols with indices, as c_i or `c[i, j]`;
- Numbers and in–built *Mathematica* constants;
- Scalar expressions, like c_1 `+` c_2 `* Sin[`α`]`;
- The results of scalar–valued functions and operators.

Declaring vectors and scalars

```
Vector Scalar VectorQ ScalarQ UnVector UnScalar
```

□ Vector operations

The arithmetic operations +, −, ∗, / and ^ can be used with vectors as well as with scalars.

The multiplication $u * v * \ldots$ of vectors is usually interpreted as operation in an associative algebra. It may be, e.g., (super)symmetric product, composition of differential operators, and so on.

There are also defined operations of tensor multiplications $u \otimes v \otimes \ldots$ and $u ** v ** \ldots$, exterior multiplication $u \wedge v \wedge \ldots$, tensor power $v^{\otimes n}$.

Other operators available in *Mathematica* may also be defined as vector operations.

Vector operations

```
CircleTimes Tp tPower VPlus VPower VTimes Wedge wedge
```

□ Functions and evaluation rules for vector expressions

SuperLie defines a list of rules and functions used to evaluate or simplify vector expressions.

• A rule may be applied manually, using *Mathematica* replacement operator *expr* `/.` *rule* or *expr* `//.` *rule*

• A rule may be declared as an operation property; then it will be applied automatically whenever possible.

Functions

```
dNormal EnvNormal ExpandOp SymmetricNormal TCollect VCollect VExpand VNormal
VSort
```

Rules

```
AdditiveRule AntiSkewSymmetricRule AntiSymmetricRule dSortRule EnvSortRule Expand-
OpRule HomogenRule JacobiRule LeibnizRule LinearCollectRule LinearRule LogPower-
Rule SkewSymmetricRule SymmetricRule TestFirstRule ThreadGradedRule VExpandRule
ZeroArgRule
```

Property declarations

```
Additive AntiSkewSymmetric AntiSymmetric DegTimes Graded Homogen Jacobi Leibniz
Linear LogPower SkewSymmetric Symmetric TestFirst ThreadGraded ZeroArg
```

Testing properties

```
AntiSkewSymmetricQ AntiSymmetricQ GradedQ SkewSymmetricQ SymmetricQ
```

Cancelling properties

```
UnAdditive UnAntiSkewSymmetric UnAntiSymmetric UnDegTimes UnGraded UnHomogen
UnJacobi UnLeibniz UnLinear UnLogPower UnSkewSymmetric UnSymmetric UnTestFirst
UnThreadGraded UnZeroArg
```

Functions and evaluation rules for scalar coefficients

```
SimplifySign SimplifySignRule SVExpandRule SVFactorRule SVNormalRule SVSimplify-
Rule $SNormal
```

■ Algebras and modules

□ Space constructors

Space constructors are commands to build (or declare) vector spaces, algebras and modules. There are constructors of vector and tensor spaces; algebra constructors (of matrices, polynomials, vector fields; free algebras; algebras with a Cartan matrix; subalgebras); module constructors (tensor modules, free modules, submodules, comodules, and so on.). More constructors can be added.

General constructors

```
Algebra CommutativeLieAlgebra SpacePlus TensorSpace TrivialSpace VectorSpace
```

Matrix algebras

```
glAlgebra pslAlgebra psq2Algebra psqAlgebra q2Algebra qAlgebra slAlgebra
sqAlgebra
```

Algebra structures on the space of polynomials and related structures

`ButtinAlgebra ContactAlgebra DiffAlgebra ExteriorAlgebra MoebiusAlgebra OKAlgebra PoissonAlgebra RamondAlgebra VectorLieAlgebra`

Algebras and modules from generators

`CartanMatrixAlgebra FreeLieAlgebra HWModule`

Related modules

`CoLeft CoRight DLeft DRight MRight NewRelative PiLeft PiRight`

Sub– and quotientspaces

`AlgebraDecomposition DefSubAlgebra QuotientModule GradedKerSpace Ideal KerSpace RestrictModule SubAlgebra SubModule SubSpace`

Modifications of vector spaces

`ReGrade`

□ Constructor options

Space constructors allow you to give additional arguments that specify options with rules of the form *name* −> *value*. Here are listed some common options used by different constructors.

Action

`Algebra`

Dimension and grading options

`Dim GList Grade GRange PList`

Implementation details

`CTimes Split`

Naming options

`Bracket bracket Div Mapping`

Format options

`Output Standard TeX Traditional`

Relation options

`CoLeft CoRight DLeft DRight MRight PiLeft PiRight`

□ Space properties

A ***space property*** is a function whose argument is the name of the space. The properties reflect the results of the space constructor.

Properties that reflect the definition of the space

```
Basis BasisPattern Bracket bracket BracketMode CartanTriade CompList Components
DecompositionList GRange InSpace NGen Rank Regular Tabular TheAlgebra TheModule
TheSpace
```

Properties that represent the results of calculations

```
DecompositionRule Dim FDim GenBasis GenRel GList Image PDim PList
```

Modules–relatives

```
CoLeft CoRight DLeft DRight MLeft MRight PiLeft PiRight
```

□ Functions and operators on vector spaces

The following functions are defined by the space constructor. Their arguments should be either even or odd vectors. The result of application of these functions to non–vector arguments is unpredictable. A linear combination of vector–valued functions is also a vector–valued function of the same arguments. A given polynomial in scalar–valued functions is also treated as a scalar–valued function of the same arguments, if all the involved functions are defined as having scalar values and scalar or vector arguments (see `Vectors and Scalars`).

Bracket and action on modules

```
Act act Bb bb Kb kb Lb lb Mb mb Ob ob Pb pb Rb rb
```

Operators related to concrete algebras

```
ContactK Δ Der der Der0 Div EulerOp HamiltonianH Mapping RamondD RamondK ZRamondD
```

Scalar functions defined on vector spaces

```
Grade P Parity PolyGrade Weight
```

□ Defining new operation on existing spaces

Sometimes, algebras are defined on already existing vector spaces. In this case no constructor is required; it suffices just to define the algebra operation.

Enveloping algebra

```
EnvelopingOperation EnvelopingSymbol $EnvLess
```

Algebra of differential operators

```
CleardSymbol dSymbol
```

■ Tools

SuperLie package provides a number of functions for manipulation with vector expressions, including

- Evaluating and simplifying vector expressions;
- Solving vector equations (with either scalar or vector unknowns);
- Working with expressions with indefinite coefficients;
- Preparing output in TEX format.

□ Working with vector expressions

Normalization

The following functions and rules may be used to convert vector expressions to the ***normal form***, i.e., to linear combination of basis elements. These functions do not change the value of vector expressions.

```
dNormal dSortRule EnvNormal EnvSortRule ExpandOp ExpandOpRule LinearCollectRule
SymmetricNormal TCollect VCollect VExpand VExpandRule VNormal VSort
```

Maps

These functions are used to map vectors to a different vector space or to change the basis in vector expressions.

```
LinearChange Mapping MappingRule
```

Polynomials

These are functions for working with (super)symmetric polynomials.

```
Deg DegreeBasis FilterBasis GradeBasis LDer UpToDegreeBasis
```

Symbolic operators

The following functions return linear operators. These operators may be used in symbolic calculations, because any linear combination of vector–valued functions is also treated as a function: `(f + α g)[x] = f[x] + α g[x]`.

```
ZId ZLDer ZRDer
```

Conditions and iterations

A number of rules of symbolic evaluation works with unexpanded conditions and iterations.

```
VIf VSum
```

Linear expressions with indefinite coefficients

A linear expression with indefinite scalar coefficients represents a subspace in a vector space, e.g., a solution of a linear system.

```
GeneralPreImage GeneralReduce GeneralSolve GeneralSum GeneralZero
```

Other functions

```
Delta MatchList VBasis VOrder VOrderQ VSameQ
```

Scalar functions

```
Plus2 SimplifySign SimplifySignRule SVExpandRule SVFactorRule SVNormalRule SVSim-
plifyRule Times2 WeightMark $SNormal $Solve
```

□ Solving vector equations

SuperLie provides functions for solving vector equations with either vector or scalar unknowns.

```
GeneralSolve GeneralZero ScalarEquation SVSolve VSolve $Solve
```

□ Splitting vector lists and expressions

Splitting is used to divide large vector expressions into homogeneous parts (e.g., of the same weight or degree).

A ***splitted sum*** is the list $\{key_1 \to expr_1, key_2 \to expr_2, \ldots\}$, where the keys are all different and sorted in canonical order, and $expr_i$ are vector expressions.

A ***splitted list*** is an expression of the same form, where $expr_i$ are lists of vectors.

Splitting expressions

```
SkipVal SplitList SplitSum
```

Accessing members of splitted expressions

```
ForSplit PartSplit
```

Manipulations with splitted expressions

```
AddSplit ApplySplit JoinSplit MapSplit MergeSplit
```

□ Preparing output

Defining output format

```
Output Standard TeX Traditional UnOutput UnStandard UnTeX UnTraditional
```

Formatting functions

```
ArgForm SeqForm
```

■ Programming

SuperLie provides tools that facilitate writing new functions.

□ Domain definition

```
CondOp PlusOp PowerOp SumOp
```

□ Notations

```
CTimes NewBracket NewBrace NewOverscript NewPower NewSuperscript Operator OpSym-
bol RemoveOverscript RemovePower RemoveSuperscript StopUseAsSymbol UseAsSymbol
```

□ Iterations

```
UniqueCounters WithUnique
```

□ Debug

```
DateString DPrint TimeString $DPrint $DPrintLabel
```

□ Preprocessor

```
GPlus GPower GTimes Plus$ Power$ PreSL Times$ WithoutPreSL
```

■ Derived packages

The following subpackages are included in the delivery:

SuperLie‘Cohom is a package for calculating cohomology;

SuperLie‘Sing is a package for calculating singular vectors.

SuperLie: Reference Manual

■ Introduction

The **SuperLie** Package is a tool for calculation of various data for Lie algebras and superalgebras. Using **Super Lie** Package you can build vector (super)spaces, algebras and modules over algebras; make the numeric and symbolic calculations with vectors (e.g., find homology and cohomology, relations between generators in an algebra, vacuum vectors, and so on); convert the results to `TeX` format.

■ Packages used by SuperLie

SuperLie uses two other packages that are designed as part of **SuperLie** project, but can be used separately. These packages are: **`Domain.m`** introducing the object–oriented style in *Mathematica*, **`Enum.m`** defining the generalized `For` loop.

Each of these packages is described in a separate document.

■ Vectors and scalars. Domains.

□ Vectors in *Mathematica* and in mathematics

In *Mathematica,* a *vector* is a list of coordinates $\{x_1,\ x_2,\ \ldots,\ x_n\}$. This represents an element of the standard n–dimensional vector space over an arbitrary numerical field. The dimension n given as an non–negative integer. The vector space as a whole is *never* used and has no name.

In mathematics, the vectors are elements of *any* vector space, not necessarily represented as lists of coodrinates. There are also the spaces of polynomials, spaces of vector fields, spaces of differential operators and many other spaces. Every space is used with some personal name, e.g., $V = \mathbb{R}^n$, $\mathfrak{a} = \Omega^\bullet(\mathbb{R}^n)$, $\mathfrak{g} = \mathfrak{gl}(n)$. The vectors are denoted so that it is possible to determine the space containing each vector: $v_1 + 2\,v_2 \in V$, $g_i^j \in \mathfrak{gl}(n)$, $f(x)\,dx_1 \wedge dx_2 \in \Omega^\bullet(\mathbb{R}^n)$.

□ Vectors in SuperLie

The vectors in **`SuperLie`** are represented in a symbolic form using the basis of the space. All vectors are represented as linear combinations of the elements of the basis with scalar coefficients. Each element of the basis of the vector space has a symbolic name. These names can be symbols (x, θ, dt), indexed symbols (x_1, $g_{i,j}$), expressions involving vector operations ($d[x_2]$, $p^2 * q^2$, $dx_1 \wedge dx_2$) or more sophisticated expressions. The only requirement is that the vectors must be declared as vectors and there must exist a unique "normal" form of such expressions (two expressions represent the same vector if and only if the operation of "normalization" transforms them to the identical expressions).

□ Vectors and Scalars

All symbols representing vectors must be declared as members of `Vector` domain. All symbols representing scalar coefficients and indices must be declared as members of `Scalar` domain. All undeclared symbols are members of `Common` domain.

The declaration of domain of the symbol is not necessary if a value was assigned to the symbol, e.g., if $x = c * v_1$, the symbol v must be `Vector`, c must be `Scalar` and x can be undeclared.

□ Vectors and polynomials. The multiplication

The vectors in **SuperLie** look like the polynomials in *Mathematica.* There is only a difference in the meaning of the multiplication. For vectors, *multiplication* is a generic name for several different associative operations: the tensor product, the [skew−]symmetric product, the multiplication in the enveloping algebra, the composition of the differential operators. Most of the multiplications have the same traditional notation as ordinary multiplication (except tensor and skew−symmetric product). The multiplication of vectors is, in general, *noncommutative*.

SuperLie allows one to use the traditional multiplication in the front end input. In the internal format it uses three different operations: `Times` for multiplications of scalars, `VTimes` for vectors and `SVTimes` to multiply vectors by scalars.

The operation `VTimes` is not commutative. This means that the factors are not ordered automatically. Instead, special operations are provided to order the factors, for each type of multiplication.

The tensor product is denoted by $**$ or $\otimes$ (CircleTimes). The skew−symmetric product is denoted by $\wedge$ (Wedge).

■ Vector operations.

The arithmetic operations $+$, $-$, $*$, / and ^ can be used with vectors as well as with scalars. The multiplication $u * v * ...$ and power v^n of vectors are usually interpreted as an associative product and power, respectively. There are also defined operations of tensor multiplication $u \otimes v \otimes ...$, exterior multiplication $u \wedge v \wedge ...$ and the tensor power $v^{\otimes n}$.

The term "operation" in *Mathematica* means the same as "function". We will use the term "operation" for a certain function if

(a) the commonly used format of the function is symbolic, as $a + b + c$ instead of `Plus`$[a, b, c]$.

(b) when one cannot calculate the result of the function, the unevaluated expression is treated as the result. The expression can be simplified or modified according to the rules prescribed for this operation.

Every operation in *Mathematica* has a full name, as `Plus`, and (optionally) a symbolic name, as $+$.

■ Linearity and other properties.

The linear functions and operators must first be defined on a basis and then expanded via linearity. This expansion will be made automatically if you declare that the function is linear. In the same way some other properties can be declared, e.g., homogeneity, additivity, (super)symmetry, Leibniz rule. You can declare and cancel the properties of functions as you need.

Properties are defined using **Domain** package (see the corresponding document).

▪ Space constructors

Space constructors are commands to build (or declare) vector spaces, algebras and modules. There are constructors of vector and tensor spaces; algebra constructors (of matrices, polynomials, vector fields; free algebras; algebras with Cartan matrix; subalgebras); module constructors (tensor modules, free modules, submodules, comodules, and so on). More constructors can be added.

▪ Tools

SuperLie package provides a number of tools for manipulation with vector expressions, such as computing "normal" form of expressions, solving vector equations, etc.

■ Vectors and scalars

▪ Vector, Scalar and Common domains.

Symbolic names can represent the vectors in two ways: either the symbol stands for a value or it is the value itself. For example, if we assign $x = c * v_1$, the symbol x stands for the value $c * v_1$, while v_1 does not stand for any value, it is a value itself.

All symbols representing vector values must be declared as members of `Vector` domain. All symbols representing scalar values (coefficients and indices) must be declared as members of `Scalar` domain. All numbers and constants are scalars by definition. All undeclared symbols are members of `Common` domain.

The declaration of domain is not necessary (and useless) for the symbols standing for any value (as x in the example above).

If a symbol v is defined to be a vector or scalar, any expression v[...] with header v and $v_{...}$ with any indices will also be a vector (resp. scalar).

□ Vector, UnVector, VectorQ

`Vector`[*obj*, ...] defines objects (they must be symbols) as vectors. Another way to define a vector (together with other properties of the objects) is `Define`[*obj*, {`Vector`, *property* ...}] or `SetProperties`[*obj*, {`Vector`, *property* ...}], see **`Domain`** Package.

Vector–valued operations must also be declared as Vectors, e.g., `Vector[Times]`.

`SetProperties`[*op*, `Vector` → *i*] tells that the *i*–th argument of the operation *op* must be a vector. Here *i* can be an integer, `First`, `Last` or `All`.

`UnVector`[*obj*, ...] or `ClearProperties`[*obj*, {`Vector`, ...}] clear the vector definition.

`VectorQ`[*x*] returns `True` if *x* is a vector (an object of `Vector` domain) and `False` otherwise. This definition shadows the system function with the same name, `VectorQ`.

□ Scalar, UnScalar, ScalarQ

`Scalar`[*obj*, ...] defines objects (they must be symbols) as scalars. Another way to define a scalar (together with other properties of an object) is `Define`[*obj*, {`Scalar`, *property* ...}] or `SetProperties`[*obj*, {`Scalar`, *property* ...}], see `Domain` Package.

Scalar−valued operations must also be declared as Scalars, e.g., `Scalar[Times]`.

`SetProperties`[*op*, `Scalar` → *i*] tells that the *i*−th argument of the operation *op* must be a scalar. Here *i* can be an integer, `First`, `Last` or `All`.

`UnScalar`[*obj*, ...] or `ClearProperties`[*obj*, {`Scalar`, ...}] clear the scalar definition.

`ScalarQ`[*x*] returns `True` if *x* is a scalar (an object of `Scalar` domain) and `False` otherwise.

■ Generic and specific operations

The front end preprocessor examines every input line obtained from the front end or terminal and replaces the operations +, −, ∗, / and ^ in this line by the "generic" operations `GPlus`, `GTimes` and `GPower`. Further on, these operations will be replaced with the operations in specific domains, `Vector` and `Scalar`. The operations in `Scalar` domain are built−in `Plus`, `Times` and `Power`, the operations in `Vector` domain are named `VPlus`, `VTimes` and `VPower`, the product $s * v$ of a scalar *s* by a vector *v* is named `SVTimes`.

The generic operation will not be replaced by the specific operation, if
− the domains of the operands are not defined;
− the expression containing the operation is not evaluated (due to the `Hold` attribute).

The following functions are used internally by the Super Lie package (they can be used to define new domains).

□ STimesOp

`STimesOp`[*domain*] gives the name of multiplication operation in the expression "scalar `*` (an element of the domain)".

□ PlusOp

`PlusOp`[*domain*] gives the name of Plus operation (`+`) in the domain.

□ PowerOp

`PowerOp`[*op*] gives the name of "power" operation associated with the "times" operation *op* or `None` if the "power" operation is not defined.

`PowerOp`[*op* → *name*] defines this operation.

The pw = PowerOp[op] operation is defined with the properties similar to the properties of the scalar Power:

$pw[x, 0] = op[]$,
$pw[x, 1] = x$,
$pw[pw[x, p], q] := pw[x, p * q]$,
$op[\ldots, pw[x, p], pw[x, q], \ldots] = op[\ldots, pw[x, p + q], \ldots]$.

The last property also embraces the case of the implicit first power:

$op[\ldots, x, pw[x, q], \ldots] = op[\ldots, pw[x, 1 + q], \ldots]$.

If you want to define the power operation associated with the (skew)symmetric operation op, you must first cancel the symmetry, then define the power operation and restore the symmetry:

UnSymmetric[op];
PowerOp[$op \to pr$];
Symmetric[op].

The reason is that the definition of the (skew)symmetry is different for operations with power and for those without power. The same is valid for Jacobi and Leibniz properties.

□ SumOp

SumOp[op] gives the name of "Sum" function associated with the "plus" operation op or None if the "Sum" operation is not defined.
SumOp[$op \to name$] defines this function.

The sum function in the Scalar domain is the system Sum function, the Vector sum function is named VSum. For both Sum and VSum the alternative iterator $\{i, from \to to\}$ is defined. If the difference $diff = to - from$ is a number, this iterator is replaced with

(a) $\{i, from, to - 1\}$ if $diff > 0$;
(b) $\{i, to, from - 1\}$ and the whole sum is multiplied by -1 if $diff < 0$.
(c) the whole sum is replaced with 0 if $diff = 0$;

□ CondOp

CondOp[*domain*] gives the name of "If" operation with values in the *domain* or None if the "If" operation is not defined.
CondOp[*domain* → *name*] defines this operation.

■ Vector operations

The following vector operation are formally defined for any vector arguments, regardless of the mathematical meaning. For example, we can write $x + v$ even if x and v are elements of different vector spaces. The meaning of the vector expressions in the user's problem, not programer's.

The result of applying of vector operation to non−vector operands in unpredictable.

□ VPlus, VTimes, VPower

VPlus[u, v, ...] or $u + v + \ldots$ is the sum of vectors. VPlus replaces the generic "+" if the operands are vectors. The operation is assotiative, i.e., the nested VPlus will be flattened. Other rules for VPlus are: operands equal to zero are replaced; VPlus[x] = x; VPlus[] = 0. For the efficiency reasons, the sorting and collection of similar terms are not made automatically; use VNormal or VCollect to this purpose.

VTimes[u, v, ...] or $u * v * \ldots$ or $u\,v\,\ldots$ or $u \times v \times \ldots$ is the multiplication of vectors. VTimes replaces the generic "*" if the operands are vectors. The operation is assotiative, but not commutative. The corresponding "power" operation is named VPower. Other rules are defined as properties (and therefore can be cancelled): ZeroArg, IdArg, Homogen.

The operation VTimes may be used for multiplication in the associative algebras in the three cases: in the (super)commutative case, in the enveloping algebra, in the algebra of differential operators. (Though so far we never used this option, VTimes can also be used for multiplication in the free

`VTimes`[u, v, ...] or $u * v * ...$ or $u\, v$ ... or $u \times v \times ...$ is the multiplication of vectors. `VTimes` replaces the generic "`*`" if the operands are vectors. The operation is assotiative, but not commutative. The corresponding "power" operation is named `VPower`. Other rules are defined as properties (and therefore can be cancelled): `ZeroArg`, `IdArg`, `Homogen`.

The operation `VTimes` may be used for multiplication in the associative algebras in the three cases: in the (super)commutative case, in the enveloping algebra, in the algebra of differential operators. (Though so far we never used this option, `VTimes` can also be used for multiplication in the free tensor algebra.) The difference only manifests itself when we have to sort the factors; this sorting is performed by different functions `SymmetricNormal` in the (super)commutative case, `EnvNormal` in the enveloping algebra, and `dNormal` in the algebra of differential operators.

`VPower`[v, n] or v `^` n or v^n is the (super)symmetric power of a vector. `VPower` replaces the generic "`^`" if the first operand is a vector.

Operations `-` and `/` are also available for vectors. The expression $u - v$ is equivalent to `VPlus`[u, `SVTimes`[-1, v]] and u / v is equivalent to `VTimes`[u, `VPower`[v, -1]].

The operation u / v is always defined, though does not always have a mathematical sense.

□ SVTimes

`SVTimes`[s, v] or $s * v$ or $s\, v$ or $s \times v$ is the product of a scalar s by a vector v. `SVTimes` replaces the generic "`*`" if the first operand is a scalar and the second one is a vector. In case of several operands the expression `GTimes`[s_1, ..., v_1, ...] is replaced with `SVTimes`[`Times`[s_1, ...], `VTimes`[v_1, ...]]. The rules for evaluation of `SVTimes` are: $1 * v = v$, $0 * v = 0$, $c * 0 = 0$, $a * (b * v) = (a\, b) * v$.

□ CircleTimes (tp), tPower

`CircleTimes`[u, v, ...] or `tp`[u, v, ...] or $u \otimes v \otimes ...$ is the tensor multiplication (the operation in the tensor algebra). The operation is associative. The corresponding "power" operation is named `tPower`. The evaluation rules of `CircleTimes` are `Linear` and `IdArg`.

`tPower`[v, n] or v `^`$\otimes$ n of $v^{\otimes n}$ is the n−th tensor power of the vector v.

□ Wedge, wedge

`Wedge`[u, v, ...] or $u \wedge v \wedge ...$ is the exterior multiplication (the operation in the exterior algebra). The operation is assotiative. The "power" operation for `Wedge` is not defined. The evaluation rules of `Wedge` are `Linear`, `Symmetric` and `IdArg`. After sorting the operands, `Wedge` is replaced with `wedge`.

`wedge`[e_1, ..., e_n] is the internal representation of the basis of exterior algebras. The external representation is also $e_1 \wedge ... \wedge e_n$.

□ NonCommutativeMultiply (Tp)

`NonCommutativeMultiply`[u, v, ...] or `Tp`[u, v, ...] or $u ** v ** ...$ denote the tensor product not regarded as multiplication in the tensor algebra, e.g., the vector−valued differential form $f[x, y] ** (d[x] \wedge d[y])$ is such a product. The single evaluation rule of `NonCommutativeMultiply` is `ZeroArg`. We have not declared the automatic linear expansion (property `Linear`) because we prefer to see as an answer to a problem something like $(4\, x + y) ** (d[x] \wedge d[y])$ rather than the expanded formula $4\, x ** (d[x] \wedge d[y]) + y ** (d[x] \wedge d[y])$. You can declare the property Linear (or Linear→First, Linear→Last) if your task requires it.

■ Active and passive forms of operators

Some operators used in SuperLie have two names: the active one and the passive one.

The active name is used when the operator should be evaluated. The passive name is used to write expression without evaluating.

For example, Der[ω] evaluates and returns the exterior derivative of ω, while the passive form der[ω] only denotes the derivative without evaluating.

Though the passive forms of operators are not evaluated, some reduction is still made. For example, the expression der[$2\,\omega_1 - \omega_2$] is usually expanded via linearity to $2\,\text{der}[\omega_1] - \text{der}[\omega_2]$. The reduction rules are defined as operator *properties* and may be changed by the user.

□ OpSymbol

If *s* denotes the active form of some operator, OpSymbol[*s*] returns the name of the passive form of the same operator.

□ Operator

If *s* denotes the passive form of some operator, Operator[*s*] returns the name of the active form of the same operator.

■ Properties of vector functions

In this section we describe the properties that any vector object or any function of the vector arguments can possess. All these properties are defined using the **`Domain`** package. Every property *prop* can manifest itself as

(a) `Value`: the function *prop*[*obj*] gives the value of the property on the object;

(b) transformation `Rule`: to be applied to the expressions containing the object whenever possible;

(c) an output `Format`: the property controls the appearance of the object in the text and TeX output;

(d) `Domain`: `Vector` and `Scalar` are such properties;

(e) `Flag`: the function *prop*Q[*obj*] gives `True` if the object possesses the property and `False` otherwise.

The same property can manifest itself in several ways, but actually we have only properties of either of some of the types (a) – (e), or `Flag` + one of (a) − (d).

Certain properties can be parametrized to more precisely describe the action of the property. For example, `Leibniz`[$f \to op$] tells that the function f acts via Leibniz rule on the expression *op*[*x*, ...]. Here *op* is a parameter of the property.

A `Rule` property can occur in several copies for the same object (with different parameters). For example, the same function f may possess Leibniz property with respect to several operation: `VTimes`, `Tp`, `tp`, `wedge`.

With every property *prop* several functions are associated:

prop[*obj*, ...]

If *prop* is a `Value` type property, *prop*[*obj*] gives the value of the property. Otherwise *prop*[*obj*, ...] declares the property of one or more objects.

prop[*obj* → *parm*, ...]
declares the property of the objects (for properties depending on a parameter);

Un*prop*[*obj*, ...]
Un*prop*[*obj* → *parm*, ...]
clears the property of one or more objects. The `Rule` type properties must be cleared exactly with the same parameter as they were declared. For other types of properties, the parameter can be omitted;

*prop*Q[*obj*, ...]
for properties of type `Flag`, it returns `True` (or the value of parameter) when the object possesses the property.

prop`Rule`[*obj*]
prop`Rule`[*obj*, *parm*]
gives the replacement rule, equivalent to the `Rule` type property. The rule can be used with the object which does not have this property. For example, the result of the transformation f[*arg*, ...] /. `ZeroArgRule`[f] is zero if one of the arguments of f is equal to `0`. If f possesses the property `ZeroArg`, this rule is useless, because f[arg, ...] is already equal to zero. To save the time of computation, such useless rules are suppressed.

In the following list, only the name, type and meaning of the properties are indicated; the associated functions are omitted.

□ ZeroArg (Rule)

`ZeroArg`[f] defines that f[...] = 0 if the argument (or one of arguments) of f is equal to 0.

□ IdArg (Rule)

`IdArg`[f] tells that if an expression like f[...] have arguments equal to `VTimes[]` (the vector unit), they must be removed: f[..., x, `VTimes[]`, y, ...] = f[..., x, y, ...].

□ Homogen (Rule)

`Homogen`[f → *deg*] tells that the scalar coefficients must be extracted from the arguments of f: $f[c * v] \to c^{deg} * f[v]$.
`Homogen`[f] is equivalent to `Homogen`[f → 1].
`Homogen`[f → `First`] and `Homogen`[f → `Last`] sets that the function f is only homogeneous in the first (last) argument (with degree 1).
The function f must be declared as vector– or scalar–valued and having `All` (resp. `First`, `Last`) vector arguments.

□ Symmetric, SkewSymmetric, AntiSymmetric, AntiSkewSymmetric (Rule and Flag)

The (skew) symmetry implies two rules of evaluations. The first rule tells that the arguments of (skew)symmetric functions of vector arguments must be sorted in the standard order:

`Symmetric`[*op*]: $op[..., x, y, ...] = (-1)^{P[x]\,P[y]}\, op[..., y, x, ...]$

`AntiSymmetric`[*op*]: $op[..., x, y, ...] = -(-1)^{P[x]\,P[y]}\, op[..., y, x, ...]$

`SkewSymmetric`[*op*] $op[..., x, y, ...] = (-1)^{(P[x]+1)\,(P[y]+1)}\, op[..., y, x, ...]$

`AntiSkewSymmetric`[*op*] $op[..., x, y, ...] = -(-1)^{(P[x]+1)\,(P[y]+1)}\, op[..., y, x, ...]$

Symmetric[*op*] $op[\ldots, x, y, \ldots] = (-1)^{\mathrm{P}[x]\,\mathrm{P}[y]}\, op[\ldots, y, x, \ldots]$

AntiSymmetric[*op*] $op[\ldots, x, y, \ldots] = -(-1)^{\mathrm{P}[x]\,\mathrm{P}[y]}\, op[\ldots, y, x, \ldots]$

SkewSymmetric[*op*]: $op[\ldots, x, y, \ldots] = (-1)^{(\mathrm{P}[x]+1)\,(\mathrm{P}[y]+1)}\, op[\ldots, y, x, \ldots]$

AntiSkewSymmetric[*op*]: $op[\ldots, x, y, \ldots] = -(-1)^{(\mathrm{P}[x]+1)\,(\mathrm{P}[y]+1)}\, op[\ldots, y, x, \ldots]$

The second rule tells that the value of a (skew)symmetric function is equal to 0 if it contains two neighboring odd (even, in case of a skew−symmetric function) equal arguments.

If the "power" operation pw = PowerOp[*op*] is defined, both rules are modified. In the first rule the arguments are sorted in the same order as the bases of pw. The second rule tells that $pw[x, n] = 0$ if x is odd (even, in case of a skew−symmetric function) and $|n| > 1$.

The function *op* must be declared as vector− or scalar−valued.

These rules may be used only when all operands are homogeneous with respect to the parity.

□ Leibniz, Jacobi (Rule)

Leibniz[$f \to g$] tells that the function f acts on $g[\ldots]$ as a derivation with parity P[f]:

$$f[g[x_1, x_2, \ldots]] = g[f[x_1], x_2, \ldots] \pm g[x_1, f[x_2], \ldots] \pm \ldots$$

Leibniz[$f \to \{g_1, \ldots\}$] tells that f acts on all $g_1, \ldots$.

Jacobi[$f \to g$] tells that $f[x, g[\ldots]]$ acts as the bracket in the Lie superalgebra g:

$$f[x, g[y_1, y_2, \ldots]] = g[f[x, y_1], y_2, \ldots] \pm g[y_1, f[x, y_2], \ldots] \pm \ldots$$

Jacobi[$f \to \{g_1, \ldots\}$] tells that f acts on all $g_1, \ldots$.

The functions f and g must be vector−valued functions of vector arguments. All arguments must be homogeneous with respect to the parity.

□ Graded (Flag)

If the vector operation *op* is graded, Grade[$a \sim op \sim b$] = Grade[a] + Grade[b]

□ ThreadGraded (Rule)

The function with property ThreadGraded acts as Grade on the graded operation:

ThreadGraded[f]: $f[a \sim op \sim b] = f[a] + f[b]$ for any graded operation *op*;

ThreadGraded[$f \to sm$]: $f[a \sim op \sim b] = f[a] \sim sm \sim f[b]$.

□ TestFirst (Rule)

The property TestFirst[f] tells that the value of $f[x_1 + \ldots]$ is equal to the value of $f[x_1]$ (here "+" is a vector operation VPlus).

□ LogPower (Rule)

LogPower[f] tells that $f[x^p] = p * f[x]$ (here "$*$" denotes SVTimes)
LogPower[$f \to op$] tells that $f[x^p] = op[p, f[x]]$.

□ **Additive** (Rule)

`Additive`[f]: f[..., $x + y$, ...] = f[..., x, ...] + f[..., y, ...]

`Additive`[$f \to$ `First`] and `Additive`[$f \to$ `Last`] sets that the function f is only additive in the first (last) argument.

This property can be assigned to functions that are declared having `All` (resp. `First`, `Last`) scalar or vector arguments and scalar or vector values.

□ **Linear** (Flag and Rule)

`Linear`[f] is the (multi)linearity of f . This property is equivalent to the union of properties `Additive`[f], `ZeroArg`[f] and `Homogen`[$f \to 1$].

`Linear`[$f \to$ `First`] and `Linear`[$f \to$ `Last`] sets that the function f is only linear in the first (last) argument.

The function f must be declared as vector– or scalar–valued and having `All` (resp. `First`, `Last`) vector arguments.

□ **Output, TeX, Standard, Traditional** (Format)

`Output`[$op \to form$] defines the output format of the object as `Format`[op[$args$]] := $form$[op[$args$]].

`TeX`[$op \to form$] defines the output TeX format of the objects as `Format`[op[$args$], `TeX`] := $form$[op[$args$]].

`Standard`[$op \to form$] defines the output format of the object in the Standard form as `Format`[op[$args$], `Standard`] := $form$[op[$args$]].

`Traditional`[$op \to form$] defines the output format of the object in the Traditional form as `Format`[op[$args$], `Traditional`] := $form$[op[$args$]].

The value of $form$ must be a function of one argument. This function is applied to every expression with header op when it is formatted for the output. For example, the function `Subscripted` will print the arguments of the expression as subscripts. In **Domain** and **SuperLie** packages, three formatting functions are defined: `InfixFormat`, `ArgForm` and `SeqForm`.

■ Vector spaces, Algebras and Modules

▪ Introduction

To *declare* (or define, or build) a vector space, an algebra or a module means to declare the name of the space, to set the properties of the space and to define the functions (operators) on the space (or with values in the space).

The properties of the space are functions whose argument is the name of the space, e.g., `Dim`[*v*] is the dimension of the space *v*, `TheAlgebra`[*m*] is the name of the algebra that acts on the module *m*. Some properties can have the second argument, e.g., `Dim`[*g*, *n*] is the dimension of the *n*−th component of the algebra *g*.

The arguments of the functions defined on the space are vectors − the elements of the space. The examples of such functions are the parity, the grading, the weight, the bracket in an algebra, the action of an algebra on a module, the scalar product, the homomorphisms, and so on.

You can easily build many vector spaces using the commands named space constructors. They define the properties of the space and the functions on the space; your task is only to choose the name of the space and to specify some parameters (e.g., dimension). Notice, however, that there is no space constructor for every kind of space, in some cases you must make all definitions manually (or to write a new space constructor).

▪ The properties of vector spaces

The properties of the space are the functions whose argument is the name of the space. The complete list of properties of vector spaces defined in the package consists of:

□ Dimension

`Dim`[*V*] returns the dimension of the space *V*.

`PDim`[*V*] returns the list containing the dimensions of the even and odd components.

`FDim`[*V*] returns the (super)dimension of *v* formatted for output.

`Dim`[*V*, *d*], `PDim`[*V*, *d*] and `FDim`[*V*, *d*] return the (super)dimension of the component of degree *d*.

□ Basis

The description of the basis of the vector space may contain:

1. An algorithm that determines whether any given vector expression represents an element of the basis of the space;
2. A method of enumerating all the elements of the basis (or all elements of the given degree);
3. An explicit description of the basis as expressions *V*[*i*, ..., *k*] with given number and ranges of subscripts;
4. Reference to one or several subspaces, each having the described basis.

The following functions correspond to the above four methods to describe the basis. The first one is mandatory for every vector space. The remaining are optional.

`BasisPattern`[*V*] returns the pattern matching all elements of the basis of the space *v*. This property is used to recognize the basis vectors in the expressions. You can read about patterns in the section Patterns of *Mathematica* book.

`Enum`[V] , `Enum`[V, n] and `EnumRange`[V, n] defines the method of enumerating of the basis of the space V (see the description of the package **Enum**). The property Enum is used to obtain the basis of the polynomials on the space and of the enveloping algebra.

`Components`[V] gives the list of descriptions of the regular components of the space V. The regular component is a subspace in which all elements of the basis have the same format $h[i, ..., k]$ with the same header h and the same number of subscripts. The action of the algebras must be defined by a single formula on each pair of regular components (see subsection **Action** below).

The description of each component is a list of 4 entries:

(1) the header h of the element of the basis;

(2) the number n of subscripts in the elements of the basis;

(3) the function of n arguments which being applied to the set of counters $i, ..., k$ returns the list of iterators $\{\{i, i_0, i_1\}, ... \{k, k_0, k_1\}\}$ for the loop embracing the whole basis of the component, and

(4) the test function of n arguments that being applied to the set of subscripts $i, ..., k$ which belong to the above loop, returns `True` if $v[i, ..., k]$ is an element of the basis of the component, and `False` otherwise. For example, the set $g[i, j]$, $1 \le i < j \le n$, is described as $\{g, 2, \texttt{Function}[\{i, j\}, \{\{i, 1, n-1\}, \{j, i+1, n\}\}], \texttt{True}\ \&\}$.

The function `Components` is used in the constructors of spaces–relatives.

`DecompositionList`[g, *name*] returns the list of subspaces $\{h_1, h_2, ...\}$ in the named decomposition of g in the direct sum $g = \oplus h_i$. The decomposition $g = g_+ \oplus g_0 \oplus g_-$ named `CartanTriade` is defined by the constructor `CartanMatrixAlgebra`. An arbitrary decomposition may be defined using the function–constructor `AlgebraDecomposition`. The decomposition is used in the constructors of irreducible modules.

v /. `DecompositionRule`[g, *name*] decomposes the vector $v \in g$ in the sum of elements of h_i.

□ Parity, grading and weight

The following properties can be defined only if the basis of the space has the form $\{x[1], ..., x[dim]\}$. They are used in the definition of the functions `P` (parity), `Grade` and `Weight`.

`PList`[V] returns the list of parities of the elements of the fixed basis.

`GList`[V] returns the list of degrees of the elements of the fixed basis.

`WList`[V] returns the list of weight of the elements of the fixed basis.

□ Parent relations

`Image`[V] returns the list of the images of the elements of the basis of V when V is a subspace of any other space.

`InSpace`[V] returns the name of the space that hosts the subspace V.

`TheAlgebra`[V] returns the name of the algebra that acts on the module V.

`TheSpace`[V] returns the name of the space from which V is derived.

□ Tensor properties

`Rank`[V] returns the tensor rank of V;

`CompList`[V] returns the list of tensor components of V.

□ Generators and relations

The following properties can be defined if the algebra or the module is built from generators.

`GRange`[V] returns $deg <$ `Infinity` if the algebra or the module V is evaluated from the generators only up to elements of degree *deg* (the degrees of the generators are given explicitly or by default; if the relations are not homogeneous, the algebra or module obtained is actually filtered rather than graded and *deg* determines the filtration). In this case the functions `Dim`, `PDim`, `FDim` return the dimension of the evaluated part of the space V.

`GenBasis`[V] returns the basis of V in terms of generators.

`GRange`[V] returns $deg <$ `Infinity` if the algebra or the module V is evaluated from the generators only up to elements of degree deg (the degrees of the generators are given explicitly or by default; if the relations are not homogeneous, the algebra or module obtained is actually filtered rather than graded and deg determines the filtration). In this case the functions `Dim`, `PDim`, `FDim` return the dimension of the evaluated part of the space V.

`GenBasis`[V] returns the basis of V in terms of generators.

`GenRel`[V] returns the list of relations between the generators.

`NGen`[V] returns the number of generators.

□ Action

`Bracket`[g] returns the name of the bracket operation in the Lie (super)algebra g or the action of the algebra on the module g.

`bracket`[g] returns the name of the unevaluated form of the bracket or the action of the algebra on the module (it is used when no evaluation is required).

`BracketMode`[V] may return `Tabular` if the basis of V has the form $\{x[1], \ldots, x[dim]\}$ and the bracket is defined using the table of values. It may return `Regular` if the regular components are defined on V (and on the algebra acting on V) and the bracket is a regular expression on each pair of regular components.

□ Spaces–relatives

For any (super)space V, there are defined 8 spaces, the relatives of V. If V is a module over a (super)algebra, the relatives are also modules. The list of relatives is (here Π is the (0|1)–dimensional space or the trivial odd module):

`MLeft`[V] is the space V itself
`CoLeft`[V] is the space of left even linear forms on V
`MRight`[V] is the space $\Pi \otimes V \otimes \Pi$
`CoRight`[V] is the space of right even linear forms on V
`PiRight`[V] is the space $V \otimes \Pi$
`DRight`[V] is the space of right odd linear forms on V
`PiLeft`[V] is the space $\Pi \otimes V$
`DLeft`[V] is the space of left odd linear forms on V (differential forms on V).

The order of relatives in the above list is not accidental. The functors of relation form a group isomorphic to the group of motion of the square. This group is generated by the dualization functor (`CoLeft`) and that of the right multiplication by Π (`PiRight`). The relatives are listed in accordance with our choice of generators.

`Relatives`[V] returns the list of the names of 8 spaces (modules) − the relatives of V (or `None` when the relative space in not built).

See also constructors `PiLeft`, `PiRight`, `MRight`, `CoLeft`, `DLeft` in the next section.

□ Changing properties of vector spaces

Most properties may be changed directly by assigning new values, e.g., `Bracket`[$\mathfrak{g}$] ^= `MyBracket`. In this way one can construct algebraic structures similar to ones defined in **SuperLie**.

Note that any such change may invalidate the vector space (except, perhaps, the changes in the output format), so the result should be carefully tested.

For some properties, modification are implemented as separate functions.

□ Regrading

`ReGrade`[V, *grading*] changes the grading on the space V and all known relatives of V.

The parameter *grading* may be either a list of new degrees for the elements of the basis (or for generators only if an algebra or a module is built using generators), or the name (a symbol or a number) of a particular grading predefined by the space constructor of V.

■ The properties and optional arguments of the space constructors

When a vector space (algebra, module) is defined by the space constructors, some of the properties can be modified using optional arguments of a space constructor, e.g.,

```
VectorSpace[x, Dim → 10, Output → Subscripted].
```

The complete list of optional arguments defined in the package consists of:

□ Dimension and parity

`Dim` → d sets the dimension of the even vector space. In this case d can be a non−negative integer, `Infinity` or a symbolic expression.

`Dim` → $(d_0 \mid d_1)$ sets the dimension of the created superspace. Both d_0 and d_1 can be non−negative integers, `Infinity`, or symbolic expressions. The parity of the elements of the basis are not defined (except for the case when $d_0 = 0$ or $d_1 = 0$).

`Dim` → $\{d_0, d_1, d_2, ..., d_m\}$ sets that the first d_0 elements of the basis of the created space are even, the next d_1 elements odd, the further d_2 elements even, etc. All the d_i must be non−negative integers, except that the d_0 and d_m (and also d_1 if $d_0 = 0$) can be `Infinity`. The last index may be also a symbolic expression.

`PList` → $\{p_1, p_2, ...\}$, where the p_i's are `0` or `1`, sets explicitly the parity of all element of the basis (or all generators) of the created space.

If, for some space V, no dimension is specified, the expression `Dim`[V] is used everywhere as the dimension of V.

□ Grading

`GList` → $\{g_1, g_2, ...\}$ sets the values of degree for the elements of the basis (or for generators only if an algebra or a module is built using generators). The default degree of generators is 1 for an algebra and 0 for a module.

□ Weight

`WList` → $\{w_1, w_2, ...\}$ sets the values of weight for the elements of the basis (or for the generators only if an algebra or a module is built from generators). The default weights are defined in some space constructors.

□ Parent relations

`TheSpace` → V declares that the created space (algebra, module) is derived from the original space V. Most space constructors assign the default value of this property.

□ Generators

`GRange` → *deg* restricts the evaluation to vectors of degree (or filtration) up to *deg* (see **Generators and relations** in the previous section).

□ Action

`Bracket` → *op* declares the name of the bracket operation in the Lie (super)algebra g or the action of the algebra on the module V. The default name of `Bracket` is `Act`. Some constructors use other defaults names, such as `Lb` for vector fields, `Pb` for the Poisson algebra, `Kb` for the Contact algebra, `Mb` for the superalgebra with the odd contact bracket.

`bracket` → *op* declares the name of the unevaluated form of the bracket or the action of the algebra on the module (it is used when no evaluation is required). The default value is the value of `Bracket's` argument with the lower–case first letter.

□ Output format

`Output` → f assigns the function that must be applied to an element of the basis to obtain its output format (or `None` if the output format is not defined or is defined separately). The default function is `Subscripted`. In the constructors of relatives, the default value is `Auto`, which means "the same format as for the original space".

`TeX` → f assignes the function that must be applied to an element of the basis to obtain its TEX format (or `None` if TEX format is not defined or is defined separately). The default value is `None`. In the constructors of relatives, the default value is `Auto` which means "the same format as for the original space".

`Standard` → f and `Traditional` → f define in the same way the standard and the traditional output format.

□ Relative spaces

relation → *name* where *relation* is the name of the relation, see the list in the beginning of the section (except `MLeft`), tells that the corresponding relative space must be generated and sets its name.

□ Other options

`Enum` → `False` suppresses the enumeration of the basis.

`Clear` → `False` allows to save the old definitions of the space that is redefining by the constructor. Only the properties not assigned by the constructor can be saved. This option may be used to define the action of several algebras on the same module.

`Algebra` → g tells that the submodule or a module–relative of the module m must be defined as modules over the algebra g rather than the algebra `TheAlgebra`[m]. This option is useful when several algebras act on the module m.

■ List of space constructors

The space constructors are commands to build (or declare) vector spaces, algebras and modules. Every constructor fixes a basis of the space, defines the properties of the space and functions on it. The following list embraces all constructors defined in the **Super Lie** package. More constructors can be added.

□ VectorSpace

`VectorSpace`[V, *options* ...] defines a new vector space V with the basis $V[i]$, where i is an integer and $1 \le i \le$ `Dim`$[V]$. The exception is the case where the list in the optional argument `Dim` $\to \{d_0, d_1, ...\}$ starts with ∞ or with $0, \infty$. In this case, the lower limit of i is $-\infty$.

The optional arguments are the following ones:

`Dim` or `PList` specifies the dimension of and the parity function on the space V;

`Output`, `TeX`, `Standard`, `Traditional` specifies the output format of the elements of the basis;

`GList` sets the degrees of the elements of the basis;

`TheSpace` tells that the created space is not the original space and sets the name of the original space.

`CoLeft`, ..., `DLeft` (the relatives) define the names of the spaces–relatives.

`Enum` → `False` suppresses the enumeration of the basis of V.

`Clear` → `False` preserves old definitions of V.

□ SubSpace

`SubSpace`[U, V, *basis*, *options* ...] defines a subspace $U \subset V$ with the given basis. The optional arguments are the same as for `VectorSpace`, except that `Dim` or `PList` are not supported. The basis of the space U will be named $U[i]$.

□ TensorSpace

`TensorSpace`[T, V, {*comp*, ...}, *options* ...] defines the space T as the tensor product of the components listed. The repeated components may be written as $comp^n$. All components must be relatives of the space V. Only format options are supported. The basis of T is $T[i_1, i_2, ...]$.

Example:

```
VectorSpace[V, Dim → 5, CoLeft → LV, DRight → DV];
TensorSpace[Tn, V, {V, LV^2, DV, V}]
```

□ CommutativeLieAlgebra

`CommutativeLieAlgebra`[V, *options* ...] defines the Lie bracket $[x[...], x[...]] = 0$ on V. Options: `Bracket` and format options.

□ MatrixLieAlgebra

`MatrixLieAlgebra`[g, V, *options* ...] defines g as the matrix Lie (super)algebra on the space V and the action of g on the space V and its relatives. Options: `Bracket`, `bracket` and format options. The basis elements of the matrix algebra are $g[i, j]$.

`MatrixLieAlgebra`[g, *options* ...] defines g as the Lie (super)algebra of matrices (the dimension is an optional argument).

□ SubAlgebra

`SubAlgebra`[h, g, {gen ...}, $options$ ...] defines the subalgebra $h \subseteq g$ generated by the elements gen ... . The basis elements of the subalgebra are denoted $h[i]$, $1 \le i \le$ `Dim`[h].

□ AlgebraDecomposition

`AlgebraDecomposition`[D, g, {h_1, h_2, ...}, $options$ ...] defines a decomposition D of the algebra $g = h_1 \oplus h_2 \oplus ...$. Each component h_i is either a subalgebra in g (defined using `SubAlgebra`) or a list {$name_i$, {g_{i_1}, ...}, $options$ ...} of arguments that generate the subalgebra h_i.

□ HWModule

`HWModule`[m, g, wt] builds the irreducible module m over the algebra g with highest weight wt. A decomposition $g = g_+ \oplus g_0 \oplus g_-$ with name `CartanTriade` should be defined on g.

Optional arguments:

`P` → p is the parity of the highest vector (default is 0)

`Grade` → r: all calculations are restricted to the elements of degree $\ge -|r|$ (the degree of the highest weight vector is 0).

`Order` → {y_1, ..., y_m} is the order in which the elements of g_- should appear in the expressions of the basis of v in terms of generators of $U(g_-)$. The default order is given by `Basis`[g_-].

See also `VectorSpace` for general options.

□ LWModule

`LWModule`[m, g, wt] builds the irreducible module m over the algebra g with lowest weight wt. A decomposition $g = g_+ \oplus g_0 \oplus g_-$ with name `CartanTriade` should be defined on g. The options are the same as for `HWModule`.

□ SubModule

`SubModule`[n, m, {gen ...}, $options$ ...] defines the $g-$submodule n of the $g-$module m generated by the elements gen... , where g is either the algebra indicated by the optional argument `Algebra` → g or (as default) `TheAlgebra`[m]. For other options see `SubSpace`.

The basis elements of the submodule are denoted $n[i]$, $1 \le i \le$ `Dim`[n].

□ RestrictModule

`RestrictModule`[m, g], where m is a subspace of any $G-$module M and g is a subalgebra of G, tests if the space m is $g-$invariant and, if it is, defines a $g-$module structure on m.

□ Ideal

`Ideal`[h, g, {gen ...}, $options$ ...] defines the ideal h of the algebra g generated by the elements gen ... . For options, see `SubSpace`.

The basis elements of the ideal are denoted $h[i]$, $1 \le i \le$ `Dim`[h].

□ PiLeft, PiRight, MRight

`PiLeft`[$m \to \pi m$, *options* ...] builds the module πm as $\Pi \otimes m$, where Π is the (0|1)−dimensional trivial module.

`PiRight`[$m \to m\pi$, *options* ...] builds the module $m\pi$ as $m \otimes \Pi$.

`MRight`[$m \to m'$, *options* ...] builds the module m' as $\Pi \otimes m \otimes \Pi$.

□ CoLeft

`CoLeft`[$m \to m'$, *options* ...] builds the module m' on the space of left even linear forms on m. The function is implemented only for finite dimensional modules with one−index bases. The option `Algebra` → g tells that m' must be defined as a g−module. The option `Clear` → `False` allows us to define the action of several algebras on m'.

□ DLeft

`DLeft`[$m \to dm$, *options* ...] builds the module dm on the space of left odd (i.e., exterior or differential) linear forms on m. The function is implemented only for finite dimensional modules with one−index bases. If m is an algebra, then `DLeft` defines the coaction `CoAct` : $dm \longrightarrow dm \wedge dm$. This allows one to use the derivative `Der` on the exterior forms on m, both with trivial coefficients and with coefficients in any m−module.

The option `Algebra` → g tells that the dm must be defined as g−module.

The option `Clear` → `False` allows one to define the action of several algebras on dm.

□ VectorLieAlgebra

`VectorLieAlgebra`[g, x] defines the Lie (super)algebra g as the Lie (super)algebra of vector fields on the space x together with its action on the (super)space of polynomials in x. The basis of g is $\{p_i[x] ** v[j]\}$, where $\{p_i[x]\}$ is the basis of polynomials in x and $\{v[j]\}$ is the basis of `CoLeft`[x] (the space of left even linear forms on x). The name of the Lie bracket and the action is `Lb`, the unevaluated form is `lb`.

Unless the space x is already graded, the standard grading is defined on x by assuming $\mathrm{Deg}[x_i] = 1$. The grading is extended to g.

The algebra g may be regraded by calling `ReGrade`[g, *grading*]. The predefined gradings are numbered from $-k$ to k there k is odd dimension of x. In the i−th grading, the first i odd elements of the basis of x have degree 0, all the other x_j have degree 1. In the $(-i)$−th grading, the last i odd elements of the basis of x have degree 0, all the other x_j have degree 1.

□ PoissonAlgebra

`PoissonAlgebra`[g, x] defines the Lie (super)algebra g as the Poisson algebra of the polynomials in x_1, ..., x_{2n} (the vector space x should be already defined) with the Poisson bracket

$$\{f, g\}_{P.b.} = -(-1)^{P[f]} \sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i} \frac{\partial g}{\partial x_{2n+1-i}} - (-1)^{P[x_i]} \frac{\partial f}{\partial x_{2n+1-i}} \frac{\partial g}{\partial x_i} \right).$$

`PoissonAlgebra`[g, {p, q}] defines the Lie (super)algebra g as the Poisson algebra of the polynomials in p_1, ..., p_n, q_1, ..., q_n (the vector spaces p and q should be already defined and have the same superdimension) with the Poisson bracket

$$\{f, g\}_{P.b.} = \sum_{i=1}^{n} (-1)^{P[f]\,P[p_i]} \left(\frac{\partial f}{\partial p_i} \frac{\partial g}{\partial q_i} - (-1)^{P[p_i]} \frac{\partial f}{\partial q_i} \frac{\partial g}{\partial p_i} \right).$$

`PoissonAlgebra`[g, {θ}] defines the Lie (super)algebra g as the Poisson algebra of the polynomials in odd indeterminates θ_1, ..., θ_n (the vector space θ should be already defined) with the Poisson bracket

$$\{f, g\}_{P.b.} = (-1)^{P[f]} \sum_{i=1}^{n} \frac{\partial f}{\partial \theta_i} \frac{\partial g}{\partial \theta_i}.$$

`PoissonAlgebra`[g, {p_1, ... p_n, q_n, ... q_1}] and `PoissonAlgebra`[g, {p_1, ... p_n, θ, q_n, ... q_1}], where all p_i, q_i and θ are (super)spaces, define the Lie (super)algebra g as the Poisson algebra of the polynomials on the direct sum of vector spaces with Poisson bracket equal to the sum of the Poisson brackets on the pairs p_i, q_i and on θ.

`PoissonAlgebra`[g, x, {{c_1, i_1, j_1}, ..., {c_m, i_m, j_m}}] defines the Poisson algebra g of the polynomials in x_1, ..., x_m (the vector space x should be already defined) with the Poisson bracket

$$\{f, g\}_{P.b.} = -(-1)^{P[f]} \sum_{k=1}^{m} c_k \frac{\partial f}{\partial x_{i_k}} \frac{\partial g}{\partial x_{j_k}}.$$

□ ContactAlgebra

`ContactAlgebra`[g, x, t] and `ContactAlgebra`[g, x, t, {{c_1, i_1, j_1}, ..., {c_m, i_m, j_m}}] define the Lie (super)algebra g as the contact algebra of the polynomials in t and the space x (or the direct sum of spaces if x is a list). The Poisson bracket on x is also defined. Here x should be either a space name or a list of space names (as for `PoissonAlgebra`).

□ FreeLieAlgebra

`FreeLieAlgebra`[g, {*gen* ...}, {*rel* ...}, *range*, *options* ...] defines the (super)algebra g generated (as a free algebra) by elements *gen*, ... with relations *rel*, ... Options `Grade` → {d_1, ...} and `PList` → {p_1, ...} define the degrees and parities of generators. All computations are made for elements with degree ≤ *range*. The basis of the new algebra will be $g[i]$, where $1 \le i \le$ `Dim`[g].

□ CartanMatrixAlgebra

`CartanMatrixAlgebra`[g, {x, h, y}, *matr*, *range*, *options* ...] defines the Lie (super)algebra with a given Cartan matrix *matr*. Its elements are named $h[i]$ (Cartan subalgebra), $x[i]$ (positive weight vectors), $y[i]$ (negative weight vectors). Computations go up to terms of degree *range*. Options `Grade` → {d_1, ...} and `PList` → {p_1, ...} define the degrees and parities of generators.

□ Classical Lie superalgebras

`glAlgebra`, see `MatrixLieAlgebra`.

`slAlgebra` builds the subalgebra of traceless matrices. For arguments see `MatrixLieAlgebra`.

`pslAlgebra` builds the quotient of `slAlgebra` modulo the algebra of scalar matrices. The `pslAlgebra` is defined if the dimension of the even and odd components of the original space are the same. For arguments see `MatrixLieAlgebra`.

For more examples, see the chapter Classical Lie Superalgebras.

■ Functions on vector spaces

The following functions are defined separately on each vector space (by the space constructor or manually). There are, however, some predefined, common for all spaces, properties of these functions. Notice that the result of application of these functions to non–vector arguments is unpredictable.

The functions `P`, `Grade`, `PolyGrade`, `Weight` are defined only for homogeneous vectors. This means that they are defined on a basis and the linear combination of the basis elements of the same parity (resp. degree, weight). The application of these functions to non–homogeneous vectors may produce an error in the calculations.

□ P

`P`[x] is the parity of x. The function `P` is defined on the homogeneous (odd or even) vectors only. The properties of `P` are `Scalar` (i.e., the value of `P` is scalar), `Homogen->0`, `TestFirst`, `ThreadGraded->(PolynomialMod[Plus[##],2]&)` and `LogPower->(SVTimes[PolynomialMod[#1,2],#2]&)`.

□ Act

`Act`[g_1, g_2] for elements g_1, g_2 of a Lie (super)algebra is the bracket operation in this algebra.

`Act`[g, m] for an element g of the algebra and an element m of a module over the algebra is the action of g on m. The name of the operation can be different, `Act` is only the default name.

The properties of `Act` are `Vector`, `Linear`, `Graded`, `Jacobi-> {tp,VTimes}`, `Output->ArgForm["[`1`,`2`]"]`, `TeX->ArgForm["[`1`,\\,`2`]"]`. The properties `Jacobi`, `Output`, `TeX` of other brackets can be different.

□ act

`act`[g_1, g_2] and `act`[g, m] represent the unevaluated operation `Act`. The properties of `Act` are `Vector`, `Linear`, `Graded`, `Output->ArgForm["[`1`,`2`]"]`, `TeX->ArgForm["[`1`,\\,`2`]"]`.

□ Grade

`Grade`[v] is the degree (grading) of the vector v. The function is only defined for graded spaces and homogeneous vectors. The properties of `Grade` are `Scalar`, `Homogen->0`, `ThreadGraded`, `LogPower->Times`, `TestFirst`.

□ PolyGrade

PolyGrade[v] returns $\mathbb{Z}^n$ – or $\mathbb{R}^n$ –grading of the vector v (as a list of numbers). Some space constructor defines PolyGrade to express grading in terms of generators. The properties of PolyGrade are Scalar, Homogen→0, ThreadGraded, LogPower→Times, TestFirst.

□ Weight

Weight[v] it the weight of the vector v. There is no generic definition of weight in **SuperLie** package, the weights are defined in some space constructors or by the user (using WList or manually). The properties of Weight are Scalar, Homogen→0, ThreadGraded, LogPower→Times, TestFirst.

□ Der, Der0

Der[ω], Der[f ** ω] is the exterior derivative on the space of forms (with trivial (scalar) coefficients as well as with coefficients in a module). The argument ω may be an odd left form on an algebra (i.e., an element of DLeft space–relative of the algebra) or an exterior product of such forms; f must be an element of a module over the same algebra. The derivative of 0–forms is written as Der[f ** Wedge[]] or Der0[f].

Der0[f] is the exterior derivative of the 0–form f ** Wedge[]. It is introduced in order to distinguish between Der[dx] , the derivative of a 1–form with trivial coefficients and Der[dx], the derivative of a 0–form with coefficient dx.

Der0[f, g] is the exterior derivative of the 0–form f, where f is regarded as an element on a g–module. This form of Der0 should be used when the algebra cannot be determined from f.

The properties of Der and Der0 are Vector, Vector->First, Linear->First.

□ der

der[ω] represents the unevaluated exterior derivative Der.

■ Tools

▪ Manipulation with vector expressions

VExpand[e] expands out all VTimes and SVTimes products in expression e.

VCollect[e] collects together the terms with the same elements of the basis, e.g., VCollect[$a\, v_1 - 2\,(v_1 - v_2)$] = $(a-2)\, v_1 + 2\, v_2$.

VNormal[e] returns the normal form of the vector expression e. The normal form of a vector is $c_1\, g_1 + c_2\, g_2 + \ldots$, where g_1, g_2, ... are different elements of the basis and c_1, c_2, ... are scalar coefficients reduced or simplified to ensure that two equal scalars are always reduced to the identical form. This evaluation of scalars is processed by the user–defined function $SNormal. The default setting of $SNormal is Expand, but in some cases it must be redefined to ensure that equal expressions will always give the same normal form. For example, if the coefficients are rational functions you must set $SNormal = Cancel. By default, the function VNormal does not sort the factors of the vector multiplication VTimes.

The following three functions (SymmetricNormal, EnvNormal, dNormal) do this differently.

The function VNormal is used by some constructors. It is possible to redefine the function

VNormal = SymmetricNormal

SymmetricNormal EnvNormal, dNormal

VNormal (for example, set VNormal = SymmetricNormal) in order to ensure the correct normal form of vector expressions.

SymmetricNormal[*e*] returns the normal form of the vector expression with the factors of the vector multiplication VTimes sorted, under the assumption that VTimes is (super)symmetric.

EnvNormal[*e*] returns the normal form of the vector expression with the factors of the vector multiplication VTimes sorted, under the assumption that VTimes is the multiplication in the enveloping algebra.

dNormal[*e*] returns the normal form of the vector expression with the factors of the vector multiplication VTimes sorted, under the assumption that VTimes is the multiplication in the algebra of differential operators.

LeibnizExpand[*d*[*parms* ..., *f*[*arg*]], *p*] expands *d*[*f*[*arg*]] as a derivation of parity *p* (in the case where *parms* is a vector and *p* is its parity, this is the expansion as the bracket in a Lie superalgebra). This function realizes the Leibniz and Jacobi rules.

VBasis[*e*] returns the list of the basis vectors encountered in the expression *e*.

VSort[*e*] sorts the terms of the vector sum *e* in a canonical order of the vector components (without expanding it). Observe that unlike this sorting, the usual Sort[*e*] orders with respect to scalar coefficients, since they come first from the left.

VOrder[*u*, *v*] and VSameQ[*u*, *v*] compares the vector components of the monomials *u* and *v*. The results are 0, ±1 for VOrder and True or False for VSameQ (cf. functions Order and SameQ).

□ Replacement rules

e /. VExpandRule expands out all VTimes and SVTimes products in expression *e*.

e /. SVExpandRule expands out all scalar coefficients in SVTimes.

e /. SVFactorRule factorize all scalar coefficients in SVTimes.

e /. SVSymplifyRule simplifies all scalar coefficients in SVTimes.

e /. SVNormalRule converts all scalar coefficients in SVTimes to the normal form using the function $SNormal.

e //. LinearCollectRule[*op*] tries to break out the factors in the sum *op*[...] + ...

SimplifySignRule is the rule for simplifying the expressions $(-1)^{polinomial}$.

■ Solving vector equations

VSolve[*eqns*, *vars*] attempts to solve a linear equation or a set of linear equations for the vector variables *vars*. VSolve[*eqns*] treats all vector variables encountered as *vars* above. For other parameters and options see Solve. In the simplest case of equation with one variable the function Solve can also be used (with InverseFunctions→True option to avoid the warning message).

SVSolve[*eqns*, *vars*] attempts to solve a vector equation or a set of equations for the scalar variables *vars*. SVSolve[*eqns*] treats all variables encountered in the scalar coefficients of SVTimes as vars above. For other parameters and options see Solve.

`ScalarEquation`[*eqns*] converts the vector equation or the system (list) of equations with scalar unknowns to the system of scalar equations.

■ Lists of vectors

`MatchList`[*expr*, *pattern*] returns the sorted list of maximal subexpressions of *expr*, matching the *pattern* (that is, subexpressions which are not parts of larger subexpressions of expr matching the same pattern).

`MatchList`[*expr*, *pattern*, *func*] returns the list of values of *func*[*term*] rather then the list of terms.

■ Splitted expressions

A splitted sum is the list $\{key_1 \to expr_1, key_2 \to expr_2, ...\}$, where the keys are sorted and $expr_i$ are vector sums. Splitted list is the expression of the same form, where $expr_i$ are lists of vectors. Keys can be any expressions.

A splitted sums and lists are used to divide large vector expressions into homogeneous parts (e.g., of the same weight or degree).

□ SplitSum, SplitList

`SplitSum`[*expr*, *pattern*] transforms the vector sum $expr = c_1 * v_1 + c_2 * v_2 + ...$ with v_i matching the pattern gathering terms with equal v. The result is the splitted sum $\{v_{i_1} \to sc_1, v_{i_2} \to sc_2, ...\}$, where $\{v_{i_1}, v_{i_2}, ...\}$ is the sorted list of subexpressions of *expr* matching the pattern (see function `MatchList`) and $sc_1, sc_2, ...$ are the sums of coefficients of $v_{i_1}, v_{i_2}, ...$. The argument *expr* can be not only sum but also a list or a single term matching the *pattern* (or $c * pattern$).

`SplitSum`[*expr*, *pattern*, *func*] transforms the same expression into a splitted list $\{f_1 \to se_1, f_2 \to se_2, ...\}$, where $\{f_1, f_2, ...\}$ is a sorted list of different values of the *func*[v_i] and $se_1, se_2, ...$ are the sums of the members of *expr* giving values $f_1, f_2, ...$ of the *func* (excluding members with *func*[v_i] == `SkipVal`).

`SplitList`[*expr*, *pattern*] and `SplitList`[*expr*, *pattern*, *func*] work in the same way as `SplitSum`, only the result is the splitted list $\{v_{i_1} \to lc_1, v_{i_2} \to lc_2, ...\}$ or, respectively, $\{f_1 \to le_1, f_2 \to le_2, ...\}$, where $lc_1, lc_2, ...$ are the lists of coefficients and $le_1, le_2, ...$ are the lists of members of *expr*.

□ ForSplit

`ForSplit`[{*expr*, *sel* → *memb*, *cnt*}, *body*] and `ForSplit`[{*expr*, *memb*, *cnt*}, *body*] evaluates the *body* in the loop for each member of the splitted sum or list *expr*. The symbols *sel* and *memb* are assigned to the current values of the selector and member of the splitted expression. The optional *cnt* is the loop counter. The functions `Continue`[], `Break`[] and `Return`[*value*] can be used in the *body*.

□ AddSplit

`AddSplit`[$expr_1, expr_2, ...$] adds terms with the same keys of splitted sums $expr_1, expr_2, ...$. Example: `AddSplit`[$\{a \to x, b \to y\}, \{a \to u, c \to v\}$] = $\{a \to x + u, b \to y, c \to v\}$.

□ JoinSplit

`JoinSplit`[$expr_1$, $expr_2$, ...] joins terms with the same keys of splitted lists $expr_1$, $expr_2$, ... Example: `JoinSplit`[$\{a \to \{x\},\ b \to \{y, z\}\},\ \{a \to \{u\},\ c \to \{v\}\}$] = $\{a \to \{x, u\},\ b \to \{y, z\},\ c \to \{v\}\}$.

□ ApplySplit

`ApplySplit`[*func*, *expr*] applies the function *func* to terms of the splitted sum or list *expr*. Example: `ApplySplit`[$f,\ \{a \to x,\ b \to y\}$] = $\{a \to f[x],\ b \to f[y]\}$.

□ MapSplit

`MapSplit`[*func*, *expr*] applies the function *func* to the members of the lists, terms of the splitted list *expr*. Example: `MapSplit`[$f,\ \{a \to \{x\},\ b \to \{y, z\}\}$] = $\{a \to \{f[x]\},\ b \to \{f[y],\ f[z]\}\}$.

□ PartSplit

`PartSplit`[*expr*, *key*] returns the part of the splitted expression *expr* with the given key or zero if *expr* has no part with the given *key*.

`PartSplit`[*expr*, *key*, *val*] returns *val* if *expr* has no part with the given *key*.

Examples: `PartSplit`[$\{a \to x,\ b \to y\},\ b$] = y; `PartSplit`[$\{a \to \{x\},\ b \to \{y, z\}\},\ c,\ \{\}$] = $\{\}$.

■ Vector Sum

`VSum`[*elt*, {*iter*} ...] evaluates the sum of vectors. Arguments are the same as in `Sum` function.

■ Expressions with Indefinite Coefficients

`GeneralSum` is a vector sum $c_1 * v_1 + c_2 * v_2 + ...$ with undetermined scalar coefficients $c_1,\ c_2,\ ...$. It is a general form of a given vector in the space with basis $v_1,\ v_2,\ ...$.

`GeneralSum`[*c*, *list*] declares *c* as scalars and returns the vector sum $c[1] * list[\![1]\!] + c[2] * list[\![2]\!] + ...$.

`GeneralSolve`[*equ*, *v*, *c*] solves the vector equation *equ* for the scalar unknowns *c*[1], ... , substitutes the coefficients found in the vector *v* (it is a general sum with coefficients *c*[1], ...), renumbers the remaining coefficients and returns the resulting vector.

`GeneralZero`[*g*, *v*, *c*] solves the equation `Act`[*g*, *v*] == 0 or the system of equations {`Act`[$g[\![i]\!]$, *v*] == 0} when *g* is a list. The result is the same as for `GeneralSolve`. The parameter *v* can be also list of vectors (basis); in this case the general sum with coefficients *c*[1], ... is used.

`GeneralReduce`[*v*, *c*] eliminates the insignificant coefficients from the general sum *v*, renumbers the remaining coefficients and returns the result. This function must be used if the dimension of the space of vectors of form *v* is less than the number of coefficients *c*.

■ Polynomials

`Deg`[*prod*, *x*] returns the degree of *x* in the symmetric product (or power).

`LDer`[*expr*, *x*] returns the left derivative of the expression from a supercommutative superalgebra.

`Delta`[*x*, *y*] = $\begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases}$.
`Delta`[*x*] = `Delta`[*x*, 0].

■ Notations

`NewBracket`[*brk*, *options*...] defines *brk* as a bracket in a Lie superalgebra. This definition sets the properties `Vector`, `Linear`, `Graded`. The options are:

`Unevaluated`→*name* tells the name of operation which represent the unevaluated bracket; the default value is `Auto`, in this case the first letter of *brk* is changed to the upper–case for the bracket operation and to the lower–case for the unevaluated form.

`Output`→*func* sets the format for the output of the expression *brk*[*g*, *h*]. The default function is `ArgForm`[" [‘1‘, ‘2‘] "];

`TeX`→*func* sets the format for TEX output of the expression *brk*[*g*, *h*]. The default function is `ArgForm`[" \left[‘1‘ \, ‘2‘ \right] "];

`Jacobi`→*op* sets the operation or the list of operations to be expanded automatically in expressions *brk*[*g*, *op*[*u*, *v*, ...]].

Introduction to SuperLie

■ 0. Introduction. Peculiarities of Mathematica important for SuperLie

■ 0.1. Symbolic representation. Input format, output format, and complete format

Mathematica enables one to perform computations in a symbolic form. This is achieved due to the possibility to work with a description of the object. One should always bear in mind that there are at least three formats of description: the input format, the screen output format and the complete format; besides, there is usually the output format of the final result into a file, for example, the TeX format.

The most important format is the complete one, since the transformations are performed with the description in the complete format. Such a description is cumbersome and is usually concealed from our eyes, but the standard command `FullForm` will reveal it.

Two objects are considered identical if their descriptions in the complete format coincide. To make mathematically equal expressions to remain equal in *Mathematica,* they should first be reduced to a standard form. In **SuperLie,** the functions `VNormal` and `VExpand` see to that.

In **SuperLie**, almost all objects are represented in a symbolic form and almost all computations are performed in a symbolic form. At all times there is available a list of transformation rules and the description is transformed according to these rules. Transformations are performed in the order of decrease of preference, returned is the description to which none of the rules is applicable. Therefore, you should not be astonished if the program returns you your input; this means that no appropriate rule is available.

■ 0.2. Small and capital letters. Patterns

Mathematica distinguishes small and capital letters; e.g., it considers b and B as totally different objects.

There are occasions when one has to have the answer in undercomputed form, e.g., the expression $[x,\ y] = 2\,h$ demonstrates a relation, whereas if we compute the left hand side we get a useless identity $2\,h = 2\,h$. To distinguish such cases, the operations that have to be computed are named with a capital letter, while the names of their not–to–be–computed twins start with a small letter. For example, the bracket in the Lie algebra is called Act and act, respectively, and the relation above is expressed as act$[x,\ y]$ = Act$[x,\ y]$.

One more reason why it is essential to distinguish smalls and capitals: quite a few functions in *Mathematica* are only applicable to the expressions satisfying a pattern. In **SuperLie**, for example, the basis of the vector space is given by means of a pattern; e.g., all the expressions of the form v[...] are by default considered as the basis elements of the space v.

■ 0.3. Lists

The ordered lists are expressed in *Mathematica* in curly brackets, separated by a comma. In particular, vectors are represented as lists with coordinates, for example, a basis of the 3–dimensional space consists of {1, 0, 0}, {0, 1, 0} and {0, 0, 1}. **SuperLie** freely uses lists, e.g., the lists of basis elements. Coordinate vectors can be used as weights, as well, in which case — important! — **SuperLie** treats them as scalars!

■ 0.4. Solving equations

Mathematica possesses powerful functions — `Solve, Eliminate` and `Reduce` — which allow one to solve and simplify the systems of equations, in particular, with parametric coefficients expressed by letters. Many computations in Lie algebras and Lie superalgebras can be reduced to this type of problems. For example, "to determine the kernel of an operator" is to solve a system of linear equations, while "to determine the image of an operator" is to simplify a linear system.

SuperLie has functions `GeneralSum, GeneralSolve, GeneralZero`, as well as `GeneralReduce, VSolve` and `SVSolve`; these functions allow one to express and solve the equations in a vector (invariant) form.

■ 1. General notions of SuperLie

■ 1.1. Objects and properties

SuperLie is an object–oriented package. This means that, as a rule, the object with which you wish to work has to be declared first and endowed with properties. For example, a given algebra has to be endowed with an *operation*; every homogeneous element of a *super*algebra has a *parity*; every derivation is extended from the space which generates the algebra to the whole algebra via Leibniz rule, and so on.

There are powerful commands that endow an object with several properties at a time. For example, the constructor `CartanMatrixAlgebra` constructs the Lie algebra from its Cartan matrix by declaring the algebra itself, and its elements, as vector objects, finds its basis (as of a vector space) expressed in terms of the generators (as of an algebra), determines a bilinear skew–symmetric bracket that satisfies the Jacobi identity, computes the relations, and so on.

More often, however, one has to add properties one at a time.

An extra property pertaining to an object in mathematics may do a harm during computations by the package: first, to slow down the computations, second, bring about the answer in a user–unfriendly form. For example, the distributivity property will lead to simplification and getting rid of the parentheses, while the answer in the form $(a + b)^{10}$ is usually more preferable than the binomial expansion. For this reason, the majority of properties can be "switched on/off" when needed, or one can as well apply them manually to a concrete expression.

■ 1.2. Vectors and scalars

SuperLie divides all the objects into three Domains: Vector, Scalar and Common or, better say, undecided ones. This division essentially differs from the conventional one. The meaning of this division is that for vectors we introduce new transformation rules for expressions, while for scalars the usual rules of *Mathematica* are applicable. The rules for common objects are such that the principal questions are put aside until the attribution is determined.

Thus, to avoid confusion, it is necessary to declare in time what are the symbols used: vectors or scalars. In particular, all the spaces, algebras, modules and their elements should be declared as vectors. Sometimes, attribution of the result of a vector operation automatically makes an object a vector, but at the beginning it is better not to hurry.

Thus, to avoid confusion, it is necessary to declare in time what are the symbols used: vectors or scalars. In particular, all the spaces, algebras, modules and their elements should be declared as vectors. Sometimes, attribution of the result of a vector operation automatically makes an object a vector, but at the beginning it is better not to hurry.

Suppose we had declared vv a vector space by means of the command VectorSpace[vv] (or, say, a Lie superalgebra by invoking an appropriate constructor). Then all the objects with name vv and indices, as well as their linear combinations with scalar coefficients (e.g., vv[i], vv[2, j], vv[$k+2$] + 2 * vv[1, 1]) are considered as vectors and elements from the vector space (or the Lie superalgebra) vv. Such a declaration will not, however, affect expressions of the form Vv[i] (a capital V is used) or vv2[j] (the name vv2 is distinct from vv).

■ 1.3. Spaces and bases

To determine a space, we have to indicate what vectors belong to it. To this end, one has to determine a basis of the space. After that, all the linear combinations of the elements of the basis (with scalar coefficients) will be considered as belonging to the space.

The most often used way to construct a space is by means of one or several standard functions called *constructors*. These constructors can be divided into two groups: the ones that declare and the ones that construct. The constructors that declare determine definitions and getting along without calculations. The constructors that construct do calculate (and the volume of calculations usually grows depressingly fast with dimension) and generate the space in question by means of the generators given (or checking if the space given is invariant) and recalculating the multiplication (action) table in the new basis.

Each constructor has mandatory and optional parameters. The mandatory parameters determine the name and the basis of the space as well as a minimal mandatory collection of properties and operations that determines a structure (of a superspace, algebra, module, and so on.) on it. The properties and operations of the space can be added, altered or switched off later by appropriate separate commands.

The declaring constructors can determine a concrete object, for example, the Lie superalgebra $\mathfrak{vect}(m \mid n)$, or work as functors, e.g., declare the dual space. Certain constructors, however, can do both, depending on the way the parameters are given. For example, glAlgebra[g, Dim −> $(m \mid n)$] constructs g as a concrete Lie superalgebra, $\mathfrak{gl}(m \mid n)$, whereas glAlgebra[g, x] acts as a functor on the superspace x: it declares g the Lie superalgebra of linear operators on x and, for free, makes x into a g−module!

Observe in passing that the parameters m and n, or the (super)dimension of the (super)space, can be infinite or symbolic. If it is symbolic, certain additional structures on g can not be constructed, e.g., the coaction Der. The same remark applies to the other constructors of Lie superalgebras.

Let us list the declaring constructors.

`VectorSpace` declares the space. With the help of `VectorSpace` one usually stocks the spaces for the constructors of other type.

`glAlgebra, slAlgebra, pslAlgebra` construct the matrix algebras $\mathfrak{gl}(m \mid n)$ and $\mathfrak{sl}(m \mid n)$, and the projectivisation $\mathfrak{psl}(n \mid n)$, respectively. If, as a parameter, a superspace is given, these constructors endow it with the structure of a module over this Lie superalgebra.

`VectorAlgebra, ContactAlgebra, PoissonAlgebra` construct the Lie superalgebras of vector fields with polynomial coefficients: $\mathfrak{vect}(m \mid n)$, $\mathfrak{k}(2n+1 \mid m)$ and $\mathfrak{po}(2n \mid m)$, respectively. If, as a parameter, a superspace v is given, these constructors endow the superspace of supersymmetric polynomials on the given superspace v with the structure of a module over the Lie superalgebra constructed.

`VectorAlgebra, ContactAlgebra, PoissonAlgebra` construct the Lie superalgebras of vector fields with polynomial coefficients: $\mathfrak{vect}(m \mid n)$, $\mathfrak{k}(2n+1 \mid m)$ and $\mathfrak{po}(2n \mid m)$, respectively. If, as a parameter, a superspace v is given, these constructors endow the superspace of supersymmetric polynomials on the given superspace v with the structure of a module over the Lie superalgebra constructed.

Constructors–functors:

`CommutativeLieAlgebra, FreeLieAlgebra` ––– their names speak for themselves.

Constructors of spaces–relatives: the dual space, the superspace with shifted parity and their compositions: `MLeft, CoLeft, MRight, CoRight, PiRight, DRight, PiLeft, DLeft`. The order of relatives in this list is important, it is used in the function `Relatives`. The first on the list is `MLeft`, i.e., the space itself, the result of application of the identity functor.

TensorSpace builds the tensor product of several spaces–relatives.

Constructing constructors:
... <Not constructed yet>

The simplest way to determine a space is to declare it with the help of the constructor `VectorSpace`. Let, for example, you executed the command

```
VectorSpace[v]
```

In this way you have determined a space named v. Having given the name of the space (for example, v, as in this example) we will by default consider all the expressions of the form v[...] as its (basis) elements (e.g., $v[2]$, $v[1, j+2]$, $v[a+b[3]]$, etc.) even if v is declared to be 1–dimensional. If for some reason such a definition of a collection of its elements does not suit us, we can define the space v differently, e.g., list its elements explicitly or determine the algorithm of recognition of the vectors that belong to v. Actually, the name of the space is the simplest pattern for basis recognition; this algorithm is, certainly, very approximative, but, usually, it suffices.

Generally speaking, element recognition is needed for solutions of vector equations; that is what **SuperLie** is based upon. The mistakes in the element recognition are the main sources of the mistakes in the computations.

Observe, however, that the noticeable part of the properties and functions automatically appears only on finite dimensional spaces with a one–index basis of the form $\{x[1], x[2], ..., x[n]\}$, where x is the name and n the dimension of the space. This concerns the lists of parities, weights and gradings of the basis elements (PList, GList, WList). On the spaces with bases of a more involved form, these functions are to be determined manually.

A simplest basis will automatically appear if we give the dimension of the space. The best way to do it, is to do it simultaneously with the declaration, e.g.,

```
VectorSpace[sl2, Dim -> 3]
```

declares a 3–dimensional space sl2 with basis {sl2[1], sl2[2], sl2[3]}.

`Basis` is one of the properties of a space. One can endow a space with other properties as well, e.g., determine its (super) dimension; construct and declare the spaces–relatives.

Further on, one usually determines functions acting on the elements of the space. For example, no meaningful job in infinite dimensional space is usually possible unless we divide the space into finite dimensional pieces. This is usually achieved by means of imposing gradings and weights.

On top of that, one has to define a parity of the superspace, a bracket on the Lie (super)algebra and an action of the algebra on the module.

■ 1.4. Vector operations

While working with scalars we use the usual signs of mathematical operations: $+$, $-$, $*$, /, ^, and *Mathematica* itself knows how to transform them into complete format. In **SuperLie**, we can use the same symbols to express the operations between vectors, as well as between scalars and vectors. If n is a scalar and u, v are vectors, we can write: $n*u*v$, u/n, $u\text{\^{}}n/v$, n/v (this is not a misprint, we do divide by a vector).

The multiplication of vectors and raising to a power are understood as a (super)symmetric product and power, respectively; the expression $/v$ is treated as a shorthand for $*v\text{\^{}}(-1)$ (this is convenient, e.g., when we deal with Laurent polynomials).

All these habitual operation signs are used in the input and (some of them) output format, but in the complete format all of them are replaced with the complete names of the operations. These names can be different, they depend on the operands' domains. For example, multiplying scalars we replace $*$ with a `Times`, multiplying a scalar by a vector we replace $*$ with a `SVTimes`, multiplying a vector by a vectors we replace $*$ with a `VTimes`, and if a domain of some of the factors is not defined we replace $*$ with a `GTimes` subsequently substituting a correct name after the domain is determined.

Thus, the vectors can by multiplied symmetrically ($u*v*...$), tensorially ($u\otimes v\otimes ...$), exteriorly ($u\wedge v\wedge ...$), and formally tensorially ($u**v**...$). One can as well raise to the tensor power ($u\text{\^{}}\otimes n$ or $u^{\otimes n}$).

Be careful with the powers! First, the replacement of the product of equal terms with a power can be banned or allowed, second, if we prescribed the derivation to act on the product via Leibniz rule, this will not automatically ensue the subsequent transformation rule of the powers; this, however, can be determined separately.

■ 1.5. Syntax preprocessor

The input format makes it possible to write down the expressions with vectors in the conventional form, with the operation sign between the terms, e.g., $u+v*w$. Further on, however, all the expressions are transformed into the complete format, where all the operations are expressed as functions, i.e., in front of the operands. For example, $u+v*w$ turns into `VPlus`[u, `VTimes`[v, w]]. Usually, *Mathematica* makes all this itself, whereas dealing with vectors inside **SuperLie** one has to boot first a special preprocessor. Without preprocessor *Mathematica* will replace the same signs of operations with scalar operations (e.g., $u+v*w$ will turn into `Plus`[u, `Times`[v, w]]) and it will be impossible to apply the specific transformation rules for vectors.

Luckily, there is no problem to peruse the final result in the conventional form, since one does not need preprocessor to transform the complete format to the output one.

■ 1.6. Objects of SuperLie

SuperLie treats vector spaces, elements (vectors and scalars), operations (also known as functions) and properties. Elements–vectors belong to spaces, only elements can be arguments of operations and functions. Only operations and functions can have properties, formally the latter are realized as functions with spaces as arguments. Properties of operations are, mainly, computation rules.

Among spaces we encounter superspaces, algebras and modules. Examples of elements: numbers, elements of a basis, linear combinations of the basis elements. Examples of operations: addition, parity, grading, the bracket in a Lie algebra, the exterior differential.

Examples of properties of spaces. Each space has at least three properties: dimension, basis, the list of declared relatives (e.g., the dual space). A property of a module is the algebra that acts on it.

Examples of properties of operations: linearity, (skew–)symmetry, Jacobi identity, the input format.

The current properties of an object (say, G) can be found out by typing ?? G (this is *Mathematica*'s command that reveals too many details) or About[G], a command from **SuperLie**.

■ 2. First steps

To work with SuperLie, you have to either copy the directory **SuperLie** to one of the subcatalogs (wherefrom your *Mathematica* can take other packages) or additionally tune *Mathematica* to the catalog containing these files.

Load the packages with commands

```
Needs["SuperLie`"]
```

The loading of the package will take a while. Success should be announced by the massage

```
SuperLie package installed
```

The command does nothing if the package is already loaded.

■ 2.1. How to construct an algebra or a module over an algebra by hand. Lie algebras $\mathfrak{gl}$(n) and $\mathfrak{sl}$(n) and the standard modules over them

This section is tougher than the neighboring ones. So you can skip it at the first reading. However, having skimmed it, you will get a better understanding of certain remarks and subtle points from other sections, even if you will not master how to construct a module or an algebra on your own. Let us explain how to construct an algebra. In principle, this is easy. One has to declare a space, describe it basis, describe the multiplication operation as a function which to every pair of basis elements assigns an element of the space and endow this function with bilinearity property.

For several standard Lie algebras the constructor–functions are written. They automatically execute the required routine actions. For example, the command

```
glAlgebra[g, Dim -> 5]
```

```
g = gl(5)
```

determines the Lie algebra $g = \mathfrak{gl}(5)$ with basis {$g[1, 1]$, $g[1, 2]$, ..., $g[5, 5]$}.

Regrettably, the brackets [., .] are already occupied in *Mathematica*, so the bracket in the Lie algebra will look familiar only in the output format. In the input format one has to type its full name. Usually this is `Act` (or `act` — for the not–to–be–executed bracket). For example, if you type

```
Act[g[1, 1], g[1, 2] + 2 g[3, 4]]
```

```
g1,2
```

you will get the computed result, whereas if you type

```
act[g[1, 1], g[1, 2] + 2 g[3, 4]]
```

```
[g1,1 , g1,2 ] + 2 [g1,1 , g3,4 ]
```

the action will not be calculated.

Having differently determined parameters of the constructor, one may, in addition to the Lie algebra, get its standard module. For example,

```
VectorSpace[x, Dim -> 4] (* declares x as a 4-dimensional space *)
 glAlgebra[g, x]
(* similarly declares g=gl (4),endowing x with a gl (4)-module structure *)
```

```
x is a vector space
```

```
g = gl(4)
```

The bilinear function that determines the action of the algebra on the module is also called `Act`. For example, `Act`[g[1, 3], x[3] + x[4]] returns x[1].

The constructor `slAlgebra` works similarly: it constructs $\mathfrak{sl}(n)$. Its parameters are the same as those of `glAlgebra`. The only but important difference: a more complicated basis: off–diagonal elements are expressed with two indices, the diagonal ones with one index. Namely, if $g = \mathfrak{gl}(n)$, then for a basis we can take all $g[i, j]$, whereas if $g = \mathfrak{sl}(n)$, then for a basis we can take all $g[i, j]$ for $i \neq j$ and $g[i]$ such that after identification of $\mathfrak{sl}(n)$ with a subalgebra of $\mathfrak{gl}(n)$ we have $g[i] = g[i, i] - g[i + 1, i + 1]$.

■ 2.2. Generators and relations

The constructors glAlgebra or slAlgebra do not to compute anything; they only declare.

More involved constructors generate algebras and modules over them by means of generators and defining relations. These constructors explicitly construct the action table and in doing so it is impossible to avoid computations, sometimes of considerable volume. In cases when the object to be generated may be infinite dimensional, the ability to shear calculations is a must. The parameter *range* serves to this purpose. By default the generators given are considered to be of degree 1, their brackets of degree 2; the brackets of the elements of degree 1 with the elements of degree 2 are of degree 3, and so on. The computations are performed until the sum of degrees (exponents) of the elements in the product or bracket surpasses range.

Given explicitly generators *gen...* and relations *rel...* the constructor `FreeLieAlgebra`[g, {*gen* ...}, {*rel* ...}, *range*] constructs a Lie algebra g. In other constructors, the generators and/or relations may be given implicitly. For example, `CartanMatrixAlgebra`[g, {x, h, y}, *matr*, *range*] constructs a Lie algebra $g = g(matr)$ with

$h[i]$ $x[i]$

$y[i]$

`GenRel`[g] `GenBasis`[g]

g g

CartanMatrixAlgebra[g, {x, h, y}, matr, range] g = g(matr)
Cartan matrix *matr*. Its basis consists of vectors *h*[*i*] (they span a Cartan subalgebra), *x*[*i*] (they are positive weight vectors corresponding to simple roots) and *y*[*i*] (they are negative weight vectors corresponding to opposite simple roots). The relations obtained are memorized. The relations are returned by the command GenRel[*g*] while GenBasis[*g*] returns the expression of the basis elements of the *space g* in terms of the *generators* of the *algebra g*.

■ 2.3. Example: $\mathfrak{g}_2$

Let as constructing the Lie algebra $\mathfrak{g}_2$ from its Cartan matrix. To be on the safe side, we restrict ourselves to elements of degree up to 50; surely, the list of elements will terminate earlier, but suppose we do not know when.

```
CartanMatrixAlgebra[g2, {x, h, y}, ( 2  -1 ), 50]
                                   (-3   2 )

14
```

In the answer stands the dimension of the Lie algebra constructed.

We can ask the computer to tell us what is now known about the Lie algebra constructed:

```
About[g2]

Domain: Vector

Flags: {Vector}

Values: {BasisPattern → _x | _h | _y, Dim → 14, PDim → {14, 0}, Enum → 3, Bracket → Act, bracket → act,
  GenBasis → {x1, x2, [x1, x2], [x2, [x1, x2]], [x2, [x2, [x1, x2]]], [[x1, x2], [x2, [x1, x2]]]},
  GenRel → {[x1, [x1, x2]] → 0, [x2, [x2, [x2, [x1, x2]]]] → 0}, GRange → 50}
```

■ 2.4. Subalgebras and submodules

If there is no standard constructor of a Lie algebra or a module you need, the simplest way out is to try to realize the structure needed as a substructure of one of the standard structures or, which is more difficult, as a quotient. If the dimension is not too high, the basis (or, at least, generators) can be listed explicitly and then you can apply one of the functions SubAlgebra, SubModule, RestrictModule, Ideal.

These functions, as well as CartanMatrixAlgebra, may be called *computing constructors*. By generating the corresponding subspace by generators given (or testing its invariance) they determine a new basis and recalculate the action table with respect to it (in particular, SubAlgebra enables one to execute a practically important change of a complicated basis with a one–index one); at the same time, as a by–product, they compute and memorize the relations obtained.

□ Example 2.4.1. Construct $\mathfrak{sl}(2)\oplus\mathfrak{sl}(2)$.

Let us realize $\mathfrak{sl}(2)\oplus\mathfrak{sl}(2)$ as a subalgebra of $\mathfrak{gl}(4)$:

We are constructing g=$\mathfrak{gl}$(4). Its elements are g[i,j]:

```
VectorSpace[x, Dim → 4]
glAlgebra[g, x]

x is a vector space

g = gl(4)
```

This is a list of 4 generators:

```
generators = {s1 → g[1, 2], s2 → g[2, 1], s3 → g[3, 4], s4 → g[4, 3]}
```

```
{s1 → g1,2, s2 → g2,1, s3 → g3,4, s4 → g4,3}
```

Compute the subalgebra

```
SubAlgebra[s, g, generators]
```

```
s is a sublagebra in g
```

This is the basis of s:

```
Basis[s]
```

```
{s1, s2, s3, s4, s5, s6}
```

This is the image of the basis of s in g:

```
Image[s]
```

```
{g1,2, g2,1, g3,4, g4,3, g1,1 - g2,2, g3,3 - g4,4}
```

This is the basis of s in terms of generators:

```
GenBasis[s]
```

```
{s1, s2, s3, s4, [s1, s2], [s3, s4]}
```

These are the relations between the generators:

```
GenRel[s]
```

```
{[s1, s3] == 0, [s1, s4] == 0, [s2, s3] == 0, [s2, s4] == 0, [s1, [s1, s2]] == -2 s1,
 [s2, [s1, s2]] == 2 s2, [s3, [s3, s4]] == -2 s3, [s4, [s3, s4]] == 2 s4}
```

It is possible to bracket the elements of a subalgebra with the elements of the algebra (we get an element of the algebra g):

```
Act[s[3], g[1, 3]]
```

```
- g1,4
```

One may not, alas!, bracket the elements of the algebra with those of a subalgebra nor act with a subalgebra on a module over the algebra. This action, however, is often needed and we will learn how to do it in a roundabout way by setting:
Act[s[i_],m[j_]]^:=Act[Image[s][[i]], m[j]].

□ Example 2.4.2. Construct the standard $\mathfrak{o}(4)$–module.

Let us realize the standard $\mathfrak{o}(4)$–module as the restriction of the standard $\mathfrak{gl}(4)$–module on the subalgebra $\mathfrak{o}(4)$ (consisting, depending on the initial problem, of skew–symmetric matrices or of the matrices X such that X^t B+BX=0, where B is not necessarily the identity matrix, just an invertible symmetric matrix).

<not written yet>

(* now x is simultaneously $\mathfrak{gl}(4)$ and $\mathfrak{o}(4)$−module *)

■ 2.5. Vector equations

Consider $\mathfrak{gl}(4)$ as a module over itself (with respect to the adjoint action).

```
glAlgebra[g, Dim → 4]
```

```
g = gl(4)
```

Suppose we wish to decompose it into irreducible submodules. For this, it suffices to find the highest and the lowest weight vectors. In our case, it suffices to find either highest or lowest weight vectors. Let us find the highest weight ones.

Let v be the expression of an arbitrary element of $\mathfrak{gl}(4)$ with indefinite coefficients:

```
gsum = GeneralSum[c, Basis[g]]
```

```
c[1] g1,1 + c[2] g2,2 + c[3] g3,3 + c[4] g4,4 + c[5] g1,2 + c[6] g2,3 + c[7] g3,4 + c[8] g1,3 +
 c[9] g2,4 + c[10] g1,4 + c[11] g2,1 + c[12] g3,2 + c[13] g4,3 + c[14] g3,1 + c[15] g4,2 + c[16] g4,1
```

The highest weight vectors are the vectors annihilated by (commuting with) g[1,2], g[2,3], g[3,4]:

```
raise = {g[1, 2], g[2, 3], g[3, 4]}
GeneralZero[raise, gsum, c]
```

```
{g1,2, g2,3, g3,4}
```

```
c[1] g1,1 + c[1] g2,2 + c[1] g3,3 + c[1] g4,4 + c[2] g1,4
```

The latter expression means that there are two highest weight vectors: $g_{1,4}$ (it generates $\mathfrak{sl}(4)$) and $E = g_{1,1} + g_{2,2} + g_{3,3} + g_{4,4}$ (it generates the one−dimensional center).

Let us describe the instruments for compilation and solution of equations in more detail.

`GeneralSum`[c, {v_1, ...}] declares the $c[i]$ as scalars and returns the vector sum of v_1 ,... with indefinite coefficients c[1],...

Let now equ be a vector equation (vec1 == vec2) containing unknown scalars of the form $c[i]$; let s be an indefinite sum from the previous paragraph. Then `GeneralSolve`[equ, s, c] solves the vector equation equ for unknowns c[1],... , substitutes the solutions found into s, renumbers the remaining unknowns without gaps and returns the expression obtained.

`GeneralZero`[g, s, c] solves the equation `Act`[g, s] == 0 (or a system of equations if g is a list). The result is the same as that of `GeneralSolve`. The parameter s may be a list of vectors, in which case an indeterminate sum with coefficients $c[i]$ is taken.

We could have spared the computation of, say gsum, in the previous example but write immediately `GeneralZero`[raise, `Basis`[g], c]. The result would have been the same.

`GeneralReduce`[s, c] eliminates the surplus coefficients in the indeterminate sum s, renumbers the remaining coefficients and returns the result. It is convenient to use this function when the

ad(g[1, 2])

`GeneralZero`[raise, `Basis`[*g*], *c*]

`GeneralReduce`[*s*, *c*] *s*

dimension of the space with elements of the form *s* is smaller than the number of the coefficients.

For example, suppose we wish to find out the dimension of the image of the operator ad(*g*[1, 2]). Compose gsum as in the last example and execute the command

```
GeneralReduce[Act[g[1, 2], gsum], c]
```

$$-c[1]\,g_{1,2} + c[2]\,g_{1,3} + c[3]\,g_{1,4} + c[4]\,(g_{1,1} - g_{2,2}) - c[5]\,g_{3,2} - c[6]\,g_{4,2}$$

We get an expression with 6 indeterminate coefficients, so, clearly, the dimension of the image is equal to 6.

■ 3. Algebras and superalgebras: tougher problems

■ 3.1. Superspaces: superdimension and parity. Gradings and weights

A powerful tool for computation of Lie algebra cohomology are various gradings of both the Lie algebra under the study and the module of coefficients, e.g., parity, degree, weight. In **SuperLie** the gradings are realized as functions on the elements of the spaces. To determine these functions, it suffices to give their values on the basis elements or on the generators (for an algebra or a module); their value on the other elements is defined via linearity by means of the bracketing or via action.

It goes without saying that a grading thus defined may turn out to be self–contradictory. **SuperLie** does not trace the contradictions and will return a chance result, so watch out!

To be graded (homogeneous) is a property of the operation that can be appointed or cancelled. If the operation Op is graded, the following rule applies:

Deg[Op[a, b]] = Deg[a] + Deg[b]

for all types of gradings. To see if the operation `Op` is graded, apply the function `GradedQ`.

Observe that the superstructuring or producing constructors endow the space produced with a grading using the gradings of the initial spaces. Therefore to determine, for example, a grading on the Poisson algebra given on a space sp, amounts to the sequence of the following operations: to define the weights on this space sp, then construct its symmetric algebra $S^*(\mathrm{sp})$ with the grading induced, and then make $S^*(\mathrm{sp})$ into the Poisson algebra with the new grading induced from $S^*(\mathrm{sp})$.

Parity. This is the grading –– called P –– by residues modulo 2, i.e., its values are $\overline{0}$ or $\overline{1}$. It is given, as any grading is always given, on the basis elements of the space or on the generators of the algebra or the module. When given, the Sign Rule is automatically taken into account in all products (brackets). If parity is not given but should be taken into account, the answers will contain factors of the form $(-1)^{P[x]}$.

The simplest way to determine a parity is to declare simultaneously with the announcement of the space the list of dimensions of the even and odd parts of the space with the help of the optional argument `Dim`. For example, VectorSpace[v, Dim → {6, 14}] makes the first 6 elements of the basis — v[1] to v[6] — even the other 14 — v[7] to v[20] — odd. This was a description of the basis of the superspace in the *standard* format. For a *nonstandard* basis, `Dim` lists the dimensions, starting with even vectors, e.g., Dim → {0, 3, 14, 15} makes the first 3 and the last 15 elements of the basis odd, the rest of the basis vectors even.

Observe that declaring Dim → (6 | 14) we determine a superdimension without affecting the parity of the basis elements!

In either case the first and/or the last elements of the list of arguments of `Dim` may be Infinity, determining an infinite dimension of the component.

Another method to determine parity: explicitly determine the list of parities of $\overline{0}$'s and $\overline{1}$'s after the argument PList. For example, the arguments PList → {0, 1, 1, 0, 0, 0} and Dim → {1, 2, 3} are equivalent.

Determined on a basis, the parity becomes a property of the space v. To see the lists of dimensions and parities determined on v, type PDim[v] or PList[v]. The function `P` returns the parity of

GList –> {g_1, g_2, ..}

PList PList → {0, 1, 1, 0, 0, 0} Dim → {1, 2, 3}

v PDim[v] PList[v] v

the element.

Grading is a generic term to denote any grading by means of scalars, usually integers, but noninteger and other scalar variables and expressions may be used. Same as parity, the grading is always given on the basis elements of the space or on the generators of the algebra or the module. The parameter GList −> {g_1, g_2, ..} in the constructor determines the grading of the basis elements or generators while the function (property) `GList`[*sp*] returns the list of the gradings of the elements (generators) of the space *sp*. By default the grading of the generators of the algebra are equal to 1, those of the module over the algebra are equal to 0. The function `Grade` returns the grading of the element.

The grading is used as a powerful tool for subdivision of the infinite dimensional space into finite dimensional components. The parameter Grade → *d* restricts the computation of the generators and defining relations with grading *d*. This is convenient in the study of spaces (algebras, modules) whose dimension is infinite or just huge, e.g., enveloping algebras. The function `Dim`[*sp*] returns the dimension of the space, while `Dim`[*sp*, *d*] returns the dimension of the component of grading *d*. The function `PDim` behaves similarly.

Weight is the grading by means of a list of scalars. Weight is determined and called in the same way as parity and grading with the help of the parameter and function `WList`. The function `Weight` returns the weight of the element.

■ 3.2. Spaces-relatives. The tensor space

The space, not a superspace, has two relatives: itself and its dual. For the superspace over complex numbers there are 8 relatives: one can (1) change parity, (2) consider linear functions acting on the left or on the right, (3) consider even and odd linear functions. (Over reals, one can also consider different linearities (linear and antilinear) over complex numbers. So far, we did not consider this case.)

If the space is a module over a Lie superalgebra, then all its relatives are also modules over the same Lie superalgebra. There is a constructor for each of the spaces−relatives; all of them are listed above. The list of the declared relatives of the space in the order indicated is a property of the space, one can get it by means of the function Relatives. This list always contains 8 elements; the nondeclared spaces should be replaced with None.

Observe that if a basis of the initial space is a one−indexed one, the respective basis of the spaces−relatives will be indexed accordingly.

An important example: *w* = `DLeft`[*v*], the space of differential 1−forms on the superspace *v*. Accordingly, the expression *w*[*i*] ∧ *w*[*j*] is a 2−form, *w*[*i*] ∧ *w*[*j*] ∧ *w*[*k*] is a 3−form,and so on. If *v* is an algebra, then the exterior algebra of exterior forms is automatically endowed with a derivation, the exterior differential, `Der`. This `Der` is defined on the space of 1−forms and its values lie in the space of 2−forms and the coaction and further extension to higher forms are defined via Leibniz rule.

■ 3.3. Vector operations. Tensor products

The constructor `TensorSpace`[*t*, *v* {comp, ...}, options ...] constructs the space *t* of tensors over *v*, by tensoring the components indicated. All these components must be relatives of *v*. Observe, however, that all the elements of *t* are of the form *t*[*i*, *j* ...], in other words, their expression never uses the fact that they are tensor products and the sign of tensor product is never used to express them.

SuperLie has no ready−to−use constructor for the associative tensor algebra, however, the tensor product operation ⊗ (other name — tp — is used for compatibility with the version for *Mathematica* 2.2) allows for any number of factors, is associative multi−linear by default and possesses a unity Id (i.e., Id ⊗ *v* = *v* ⊗ Id = *v* for any vector *v*). So we can, actually, perform actions in the tensor algebra. Usually, one uses ⊗ to multiply (to tensor) elements of the same space. The tensor product of *n* copies of the same element *x* can be expressed as *x*^⊗*n* or $x^{\otimes n}$ (in full format: `tPower`[x,n]). Observe, however, that by default *x*^⊗0 is not replaced with Id.

When we wish to (point out that we) tensor elements from different spaces, we should use the formal tensor product. The operation ∗∗ (full name `NonCommutativeMultiply`) is used in this case. This operation is binary (hence, one should not worry about associativity), is not linear by default, and does not recognize vector unity Id (i.e., neither *v* ∗∗ Id not Id ∗∗ *v* are replaced with *v*). Its only by default property is `ZeroArg`, i.e., formal tensoring by 0 gives a 0.

Instructive examples. 1. The constructor `VectorLieAlgebra`[*a*, *v*] constructs the algebra a with a basis consisting of expressions *w*[*i*], *v*[*j*] ∗∗ *v*[*i*], (*v*[*j*] ∗ *v*[*k*]) ∗∗ *w*[*l*], where *w* is the space of left even linear forms on *v*.

2. The space of 2−cochains on *g* with values in the module *m* has for a basis elements of the form *a* ∗∗ (dg[*i*] ∧ dg[*j*]), where dg is the space of left odd forms on *g*.

It is usually convenient to endow ∗∗ with "linearity with respect to one argument" property (say, in the last example, with respect to the second argument). This enables one to get the answer in a compressed, simplified, form, e.g., (4 *a* + *b*) ∗∗ dg[*i*] ∧ dg[*j*].

Alphabetical list of symbols in SuperLie

■ A

□ Act

`Act` is the default name of the bracket operation in Lie (super)algebras and the action of algebras on modules.

`Act`$[g_1, g_2]$ for elements g_1, g_2 of a Lie (super)algebra is the bracket operation in this algebra.

`Act`$[g, v]$ for an element g of the algebra and an element v of the module over the algebra is the action of g on v. The name of the operation can be different, `Act` is only the default name.

The properties of `Act` are `Vector`, `Linear`, `Graded`, `Jacobi→{CircleTimes,VTimes}`. The output format of `Act`$[x, y]$ is $[x, y]$.

□ act

`act`$[g_1, g_2]$ and `act`$[g, v]$ represent the unevaluated operation `Act`.

□ Additive

`Additive`$[f]$ constitutes the automatic additive expansion of f:

$$f[\ldots, x + y, \ldots] = f[\ldots, x, \ldots] + f[\ldots, y, \ldots]$$

`Additive`$[f \to$ `First`$]$ and `Additive`$[f \to$ `Last`$]$ see to that the function f is only additive in the first (last) argument.

Here + denote either `Plus` or `VPlus` depending on the definition of the function f. This property can be assigned to functions that are declared having `All` (resp. `First`, `Last`) `Scalar` or `Vector` arguments and scalar or vector values.

See also `AdditiveRule`, `UnAdditive`.

□ AdditiveRule

`AdditiveRule`$[f]$ is the replacement rule for the additive expansion of f:

$$f[\ldots, x + y, \ldots] \to f[\ldots, x, \ldots] + f[\ldots, y, \ldots].$$

`AdditiveRule`$[f,$ `First`$]$ and `AdditiveRule`$[f,$ `Last`$]$ are rules for the additive expansion of the first (last) argument of f.

The rule is valid for functions that are declared having `All` (resp. `First`, `Last`) `Scalar` or `Vector` arguments and scalar or vector values.

See also `Additive`, `UnAdditive`.

□ AddSplit

`AddSplit`$[e_1, e_2, \ldots]$ adds the terms of splitted sums $e_1, e_2, \ldots$ with equal keys and returns the merged splitted sum.

□ Algebra

`Algebra`[$\mathfrak{g}$, *options*] defines the vector space and algebra $\mathfrak{g}$. The operation on $\mathfrak{g}$ should be defined explicitly.

`Algebra` → $\mathfrak{g}$ is an option for some module constructors. It specifies the algebra acting on the modules in consideration.

□ AlgebraDecomposition

`AlgebraDecomposition`[F, $\mathfrak{g}$, {$\mathfrak{h}_1$, ...}] – defines the decomposition $F : \mathfrak{g} \to \mathfrak{h}_1 + ...$ of the Lie (super)algebra $\mathfrak{g}$ in the sum of subalgebras $\mathfrak{h}_1$, ... (The sum should be the direct sum of vector spaces).

□ AntiSkewSymmetric

`AntiSkewSymmetric`[f] introduces the automatic sorting of arguments of f using super–anti–skew symmetry $f[.., y, x, ..] = -(-1)^{(1+P[x])(1+P[y])} f[.., x, y, ..]$.

□ AntiSkewSymmetricQ

`AntiSkewSymmetricQ`[f] returns `True` if f was declared anti–skew–symmetric.

□ AntiSkewSymmetricRule

`AntiSkewSymmetricRule`[f] is the replacement rule for sorting of arguments of f using super–anti–skew symmetry $f[.., y, x, ..] \to -(-1)^{(1+P[x])(1+P[y])} f[.., x, y, ..]$.

□ AntiSymmetric

`AntiSymmetric`[f] introduces the automatic sorting of arguments of f using super–antisymmetry $f[.., y, x, ..] = -(-1)^{P[x]P[y]} f[.., x, y, ..]$.

□ AntiSymmetricQ

`AntiSymmetricQ`[f] returns `True` if f was declared antisymmetric.

□ AntiSymmetricRule

`AntiSymmetricRule`[f] is the replacement rule for sorting of arguments of f using super–antisymmetry $f[.., y, x, ..] \to -(-1)^{P[x]P[y]} f[.., x, y, ..]$.

□ ApplySplit

`ApplySplit`[f, e] applies function f to terms of the splitted sum or list e.

□ ArgForm

`ArgForm`[*controlstring*] [*f*[*x*, ...]] is equivalent to `StringForm`[*controlstring*, *x*, ...]. It is used as value of format options, e.g.,`TeX→ArgForm["\\left[`1`,\\,`2`\\right]"]`.

□ Auto

`Auto` is the default value for some options.

■ B

□ Basis

`Basis`[*V*] returns the basis of the space *V*. `Basis`[*V*, *d*] returns the basis of the *d*−th grade component of the space *V*.

□ BasisPattern

`BasisPattern`[*V*] returns the pattern for basis of the space *V*. This pattern matches all elements of the basis of *V*. No other expressions matching this pattern should be used in calculations.

□ Bb

`Bb`[*x*, *y*] is the Buttin bracket (operator).

□ bb

`bb`[*x*, *y*] is the Buttin bracket (unevaluated form).

□ bracket

`bracket`[$\mathfrak{g}$] returns the name of the unevaluated form of the bracket in Lie algebra $\mathfrak{g}$. `bracket`[$\mathfrak{g}$, $\mathfrak{m}$] returns the name of the unevaluated form of the action of the Lie algebra $\mathfrak{g}$ on the module $\mathfrak{m}$ (not implemented yet).

□ Bracket

`Bracket`[$\mathfrak{g}$] returns the name of the bracket in Lie algebra $\mathfrak{g}$. `Bracket`[$\mathfrak{g}$, $\mathfrak{m}$] returns the name of the action of the Lie algebra $\mathfrak{g}$ on the module $\mathfrak{m}$ (not implemented yet).

□ BracketMode

`BracketMode`[$\mathfrak{m}$] returns the method of the definition of the bracket operation on the algebra or of the action of the algebra on the module. The valid modes are `Regular` or `Tabular`.

□ ButtinAlgebra

`ButtinAlgebra`[$\mathfrak{g}$, {x, y}] defines a Buttin algebra $\mathfrak{g}$, the algebra of polynomials on x_1, ..., x_n, y_1, ..., y_n with the standard Buttin bracket. The spaces x and y should be declared with the same dimension. For every i the parities of x_i and y_i should be different.

■ C

□ CartanMatrixAlgebra

`CartanMatrixAlgebra`[$\mathfrak{g}$, {x, h, y}, *matr*, *range*] defines a Lie (super)algebra $\mathfrak{g}$ with given Cartan matrix *matr*. Its elements are named h_i (Cartan subalgebra), x_i (of positive weight), y_i (of negative weight). Computations go up to terms of degree *range*. The parities P[x_i] and grading `Grade`[x_i] of positive generators may be defined by options `PList` → {p_1, p_2, ...} and `GList` → {g_1, g_2, ...}. The option `Grade` → g limits computations up to terms of grading (degree) g.

□ CartanTriade

`CartanTriade`[$\mathfrak{g}$] = {$\mathfrak{g}_+$, $\mathfrak{g}_0$, $\mathfrak{g}_-$} is the decomposition of algebras into 3 subalgebras $\mathfrak{g} \longrightarrow \mathfrak{g}_+ \oplus \mathfrak{g}_0 \oplus \mathfrak{g}_-$, the components of positive, zero and negative weight.

□ CircleTimes

$v_1 \otimes v_2 \otimes ...$ denotes the tensor product of e_1, e_2, ... as operation in tensor algebra.

□ CleardSymbol

`CleardSymbol`[] clears the symbol that was used as differential operator.

□ CoLeft

`CoLeft`[$\mathfrak{m}$] returns the name of the space of left even linear form on $\mathfrak{m}$, or `None` if this space was not defined.

`CoLeft`[$\mathfrak{m} \to \mathfrak{l}$] builds the module $\mathfrak{l}$ on the space of left even linear form on $\mathfrak{m}$. The function is implemented only for finite−dimensional modules with one−indexed basis.

`CoLeft` → $\mathfrak{l}$ is the options to the `VectorSpace`.

□ CommutativeLieAlgebra

`CommutativeLieAlgebra`[V] defines on space V a Lie bracket $[v_i, v_j] = 0$.

□ CompList

If $V = \{v_{i_1,\ldots i_n}\}$ is an n−indexed tensor space declared with constructor `TensorSpace`, `CompList`[V] returns the list of vector spaces−components of the tensor space V.

□ Components

For an algebra (module) $\mathfrak{m}$ with `Regular` bracket (action of the algebra) `Components`[$\mathfrak{m}$] returns the list of the regular components of $\mathfrak{m}$.

□ CondOp

`CondOp`[*domain*] returns the name of a conditional operation with values in the domain. This operation is defined as

$$op[cond,\ value] := \begin{cases} value, & \text{if } cond == \texttt{True}, \\ 0, & \text{if } cond == \texttt{False}, \\ \text{unevaluated}, & \text{otherwise.} \end{cases}$$

The conditional operation is involved in symbolic processing.
`CondOp`[*domain* −> *name*] defines this operation.

□ ContactAlgebra

`ContactAlgebra`[$\mathfrak{g}$, x, t] and `ContactAlgebra`[$\mathfrak{g}$, x, t, {{c_1, i_1, j_1}, ..., {c_m, i_m, j_m}}] define the Lie (super)algebra $\mathfrak{g}$ as the contact algebra of the polynomials in x_1 ,... and t.

`ContactAlgebra`[$\mathfrak{g}$, {p, ..., q}, t] defines the contact algebra of polynomials in p_1 , ... q_n , t. The Poisson bracket on x or {p, ... q} is also defined. The spaces x or p, ..., q should be already defined (as for `PoissonAlgebra`).

□ ContactK

`ContactK`[$\mathfrak{g}$] is the operator from functions to vector fields (implemented as differential operators) associated with the bracket in the Contact algebra $\mathfrak{g}$.

□ CoRight

`CoRight`[$\mathfrak{m}$] returns the name of the space of right even linear form on $\mathfrak{m}$, or `None` if this space was not defined.

`CoRight`[$\mathfrak{m} \to \mathfrak{r}$] builds the module $\mathfrak{r}$ on the space of right even linear form on $\mathfrak{m}$. The function is implemented only for finite−dimensional modules with one−indexed basis.

`CoRight` → $\mathfrak{r}$ is the options to the `VectorSpace`.

□ CTimes

`CTimes`[*op*] declares new "Coefficient Times" operation that may be used instead of ** between coefficient and vector parts of forms, vector fields, etc. `CTimes` −> *op* is an options that indicates such operation.

■ D

□ Δ

△[$\mathfrak{g}$] is the Laplacian in the algebra $\mathfrak{g}$.

□ DateString

`DateString[]` returns a string representing the current date and time.

□ DecompositionList

`DecompositionList`[g, f] returns the list $\{g_1, \ldots\}$ of subalgebras in the decomposition f of subalgebra $g_1 \oplus \ldots \subset g$. The bracket operation is defined for any pair of elements in $g_1 \oplus \ldots$.

□ DecompositionRule

expr /. `DecompositionRule`[g, f] replaces the elements of the basis of g in the expression expr with their images under decomposition $f : g \longrightarrow g_1 \oplus \ldots$.

□ DefSubAlgebra

`DefSubAlgebra`[$\mathfrak{h}$, $\mathfrak{g}$, $\{g_1, \ldots\}$] – defines the subalgebra $\mathfrak{h}$ of the Lie (super)algebra $\mathfrak{g}$ on the vector space Span($g_1, \ldots$) $\subset \mathfrak{g}$. The space should be closed with respect to the bracket in $\mathfrak{g}$.

The bracket in the subalgebra is defined in `Tabular` form. It is fastest for calculations and supported by all functions in **SuperLie** package.

Option `Split`→f instructs to split the space in order to accelerate calculations.

□ Deg

`Deg`[*expr*, x] returns the degree of x (or all elements matching pattern x) in vector expression *expr*. If the expression is not homogeneous, the result is `Mixed`.

□ DegreeBasis

`DegreeBasis`[*deg*, *vars*] returns the list of elements of degree deg in the symmetric algebra of variables vars. `DegreeBasis`[*deg*, *vars*, *op*] uses *op* instead of `VTimes`. The operation *op* must be `Flat` and `Listable`.

□ DegTimes

`DegTimes`[*op*] introduces the property `Deg`[*op*[$v_1, \ldots v_n$], x] = `Deg`[v_1, x] + ... + `Deg`[v_n, x] .

□ Delta

$$\texttt{Delta}[x, y] = \begin{cases} 1, & \text{if } x == y \\ 0, & \text{if } x \neq y \end{cases}.$$

`Delta`[x] = `Delta`[x, 0].

□ Der

`Der`[ω] is the exterior derivative of the form (multilinear function) ω with trivial (scalar) coefficients.

`Der`[m ∗∗ ω] is the exterior derivative of the form m ∗∗ ω with coefficients in a module.

The form ω may be an odd left form on a Lie algebra or an exterior product of such forms; m must be an element of a module over the same algebra. The derivative of 0–form with coefficients in a $\mathfrak{g}$–module is written as `Der0`[m, $\mathfrak{g}$].

□ der

`der` is the passive form of the exterior derivative `Der`.

Properties: `Linear→First`

□ Der0

`Der0`[m, $\mathfrak{g}$] is the exterior derivative of the 0–form m ∗∗ `wedge`[] with coefficients in a $\mathfrak{g}$–module.

□ DiffAlgebra

`DiffAlgebra`[$\mathfrak{g}$, V] defines $\mathfrak{g}$ as the algebra of differential operators on the space V. The option `D` → d defines the notation for partial derivative: $\frac{\partial}{\partial x}$ is denoted by $d[x]$. The option `Bracket` → b declares the name of the supercommutator with respect to the standard parity induced by that of V. The default options are `D` → `d` and `Bracket` → `Dc`.

□ Dim

`Dim`[v] returns the dimension of the vector space v.

`Dim`[v, n] returns the dimension of the components of degree n of the space v.

`Dim` → d is the option of space constructors that defines the dimension of the constructed space. The value of d may be

- a non–negative integer, a scalar expression or `Infinity`. The result will be an even vector space of a given dimension.
- (d_0 | d_1), where both d_0 and d_1 are non–negative integers, `Infinity`, or symbolic expressions. The result will be a superspace of given dimension. The parity of the elements of the basis are not defined (except for the case when $d_0 == 0$ or $d_1 == 0$).
- {d_0, d_1, d_2, ..., d_m}, where all the d_i are non–negative integers with two exceptions:
  (1) d_0 (or d_1 if $d_0 == 0$) may be `Infinity` and
  (2) d_m may be a symbolic expression or `Infinity`. The first d_0 elements of the basis of the created space are even, the next d_1 elements are odd, the further d_2 elements are even, and so on.

□ Div

The options `Div`→*operation* of function `VectorLieAlgebra` gives the name the divergence of (poly)vector fields. The default name is also `Div`.

`Div`[v] calculates the divergence of the vector (or polyvector) field v.

□ DLeft

`DLeft`[$\mathfrak{m}$] returns the name of the space of left odd linear form on $\mathfrak{m}$ (*differential forms*, for short), or `None` if this space was not defined.

`DLeft`[$\mathfrak{m} \to \mathfrak{d}$] builds the module $\mathfrak{d}$ on the space of left odd linear forms on $\mathfrak{m}$ (differential forms). If $\mathfrak{m}$ is a Lie algebra, `DLeft` also defines the exterior derivative `Der` : $\mathfrak{d} \longrightarrow \mathfrak{d} \wedge \mathfrak{d}$. The function is implemented only for finite−dimensional modules with one−indexed basis.

`DLeft` → $\mathfrak{d}$ is the option of the `VectorSpace`.

□ dNormal

`dNormal`[*e*] gives the standard form of differential expression. The multiplication of vectors is interpreted as a composition of differential operators. See also `dSortRule`, `dSymbol`, `DiffAlgebra`, and `VNormal`.

□ DPrint

`DPrint`[*level*, *data* ...] prints *data* (using `Print`) if `$DPrint` ≥ *level*. See also `$DPrintLabel`.

□ DRight

`DRight`[$\mathfrak{m}$] returns the name of the space of right odd linear form on $\mathfrak{m}$, or `None` if this space was not defined.

`DRight`[$m \to md$] builds the module of right odd forms on m.

`DRight` → $\mathfrak{d}$ is an option of the `VectorSpace`.

□ dSortRule

e //. `dSortRule` sorts the terms in the differential expression *e*: $d[x]\, x \to x\, d[x] + 1$, where the symbol d is defined by the function `dSymbol` or option `D` of `DiffAlgebra`.

□ dSymbol

`dSymbol`[*d*] assigns the symbol used as differential operator: $d[x]^n$ represents the differential operator $\frac{\partial^n}{\partial x^n}$.

■ E

□ EnvelopingOperation

`EnvelopingOperation`[*times*, *power*, *brk*, *order*] defines *times* to be an operation in enveloping algebra with Lie bracket *brk*. The argument *power* is the power operation corresponding to *times*. The parameter *order* is an user−supplied function that defines the what will be called in what follows the *canonical* order: *order*[*f*, *g*] returns `True` if f and g are in the canonical order. (In other words, the user defines any order in the enveloping algebra and it will be called canonical as a point for reference.) The parameters *power*, *brk* and *order* are optional; the default values are `None`, `Act` and the value of `$EnvLess`, respectively.

□ EnvelopingSymbol

`EnvelopingSymbol`[*symb*, *mult*, *brk*, *order*] introduces notations: *symb*[...] for the elements of basis of an enveloping algebra, *mult*[...] for the multiplication in enveloping algebras with bracket *brk*. The user–supplied function *order*[*f*, *g*] should return `True` if *symb*[*f*, *g*] is in the canonical order in the enveloping algebra.

□ EnvNormal

`EnvNormal`[*e*] gives the standard form of an element *e* of the enveloping algebra with multiplication `VTimes` and bracket `Act`. The factors in the vector products are sorted (see `Sort`) using ordering function `$EnvLess`.

`EnvNormal`[*e*, *p*] uses *p* as an ordering function. The ordering function *p* is defined so at the value of *p*[*x*, *y*] is `True` if the factors *x* and *y* should appear in the products in that order.

See also `VNormal`.

□ EnvSortRule

e //. `EnvSortRule` sorts the terms of the multiplication in the enveloping algebra (with multiplication `VTimes`, bracket `Act` and order function `$EnvLess`) into the canonical order.

□ EulerOp

`EulerOp`[$\mathfrak{g}$] is the Euler operator in the algebra $\mathfrak{g}$.

□ ExpandOp

`ExpandOp`[*e*, *op*] expands the operation *op* in the expression *e* as a (noncommutative) multiplication.

□ ExpandOpRule

e //. `ExpandOpRule`[*op*] expands the operation *op* in the expression *e* as a (noncommutative) multiplication.

□ ExteriorAlgebra

`ExteriorAlgebra`[Ω, *x*] defines the exterior algebra on the space *x*. Options change the operation names: `Wedge` →exterior product, `Der` →derivative, `CTimes` →the multiplication in $x \cdot dx$. The `PiRight` relative of *x* should be defined.

■ F

□ FDim

`FDim`[*V*] returns dimension of the vector (super)space *V* formatted for output.

□ FilterBasis

`FilterBasis`[d, $\{x_1, ..., x_n\}$] returns the list of elements of grade $\leq d$ in the symmetric algebra in indeterminates $\{x_1, ..., x_n\}$. The grades of the $x_1, ..., x_n$ must be predefined. `FilterBasis`[d, $\{x_1, ..., x_n\}$, op] uses op instead of `VTimes`. The operation op must be `Flat` and `Listable`.

□ ForSplit

`ForSplit`[{$expr$, $memb$, cnt}, $body$] or `ForSplit`[{$expr$, $sel \to memb$, cnt}, $body$] evaluates the $body$ for each term of the splitted sum or list $expr$. The variables sel and $memb$ are assigned to the current values of the selector and the member of the $expr$, respectively. The optional cnt is the loop counter. The functions `Break`[], `Continue`[] and `Return`[$value$] can be used in the $body$.

□ FreeLieAlgebra

`FreeLieAlgebra`[$\mathfrak{g}$, $\{g_1, ... g_n\}$, $\{r_1, ... r_k\}$, $range$] defines the (super)algebra Lie $\mathfrak{g}$ generated (as free algebra) by the elements $g_1, ..., g_n$ modulo relations $r_1, ... r_k$. Options `Grade` $\to \{d_1, ... d_n\}$ and `PList` $\to \{p_1, ..., p_n\}$ define, respectively, the degrees and parities of the generators. All computation are made for elements with degree $\leq range$.

■ G

□ GenBasis

`GenBasis`[$\mathfrak{g}$] returns the representation of the basis of $\mathfrak{g}$ in terms of (free) generators. This list is prepared by some space constructors.

□ GeneralBasis

`GeneralBasis`[v, c] converts a general sum $v = c_1 v_1 + ... + c_n v_n$ with arbitrary vector expressions $v_1, ..., v_n$ into the list $\{v_1, ..., v_n\}$.

□ GeneralPreImage

`GeneralPreImage`[f, x, c, y, d] calculates the projection of $x = c_1 x_1 + c_2 y_2 ...$ to the preimage of $y = d_1 y_1 + d_2 y_2 ...$ under f.

□ GeneralReduce

`GeneralReduce`[v, c] eliminates insignificant coefficients in the general sum $v = c_1 e_1 + ...$, renumbers the remaining coefficients and returns the result.

□ GeneralSolve

`GeneralSolve`[equ, v, c] solves the vector equation equ for $c_1, ...$, substitutes the found coefficients $c_1, ...$ in the general sum $v = c_1 e_1 + ...$, renumbers the remaining coefficients c_i and returns the result.

`GeneralSolve`[equ, v, c, a] solves the equation eliminating the scalar coefficients $a_1, ...$.

□ GeneralSum

`GeneralSum`[c, {v_1, ...}] returns the general linear combination $c_1\, v_1 + ...$ and declares c_i scalars.

`GeneralSum`[c, {v_1, ...}, f] returns the general linear combination $v = c_1\, v_1 + ...$ such that $f[v] == 0$.

□ GeneralZero

`GeneralZero`[g, v, c, brk] returns the general solution of $brk[g, v]$ with coefficients c_1, ... The g is an element or a list of elements of an algebra $\mathfrak{g}$; v is either a list of vectors in the $\mathfrak{g}$–module or a general sum with coefficients c_i . The default value for brk is `Act`.

□ GenRel

`GenRel`[$\mathfrak{g}$] returns the list of relations between the generators of $\mathfrak{g}$. This list is prepared by some space constructors.

□ glAlgebra

`glAlgebra`[$\mathfrak{g}$, V, *options*] defines $\mathfrak{g}$ as a matrix Lie (super)algebra on the space V.

`glAlgebra`[$\mathfrak{g}$, *options*] defines $\mathfrak{g}$ as Lie (super)algebra of square matrices; the size of matrices and the parities of rows are given by options `Dim` or `PList`.

□ GList

`GList` → {d_1, ...} is the optional parameter of some space constructor functions. It is the list of degrees of generators of the new space.

□ GPlus

`GPlus` is the name of "+" in the `Common` domain. The operation + is treated as `GPlus` when (a) not all operands are defined as `Vectors` or `Scalars`, or (b) a scalar is added with a vector. The operation `GPlus` is automatically replaced with `VPlus` if all operands become vectors and with `Plus` if all operands become scalars.

□ GPower

`GPower` is the name of the power operation in the `Common` domain. The operations $a\text{^}b$ and a^b are treated as `GPower` when a is not defined as either `Vectors` or `Scalars`.

The operation `GPower` is automatically replaced with `VPower` when a is a vector and with `Power` when a is a scalar.

□ Grade

For a graded object m, the value `Grade`[m] is the grading (degree) of m.

`Grade` → {d_1, ...} is an option for space constructors.

□ GradeBasis

`GradeBasis`[d, {x_1, ... x_n}] returns the list of elements of grade d in the symmetric algebra in x_1, ... x_n. The grades of the indeterminates must be predefined. `GradeBasis`[d, {x_1, ... x_n}, op] uses op instead of `VTimes`. The operation op must be `Flat` and `Listable`.

□ Graded

`Graded`[op] introduces the property `Grade`[$a \sim op \sim b$] = `Grade`[a] + `Grade`[b].

□ GradedKerSpace

`GradedKerSpace`[U, V, f] calculates the subspace $U = \{v \in V \mid f(v) == 0\}$. Here f is a linear function or a list of linear functions on V. The oprions `From` → *degree* and `To` → *degree* restricts the calculations to the specified degrees (the defaults are 0 and `GRange`[V], respectively).

□ GradedQ

`GradedQ`[op] returns `True` if the operation op is graded.

□ GRange

`GRange` → n is an optional parameter for space constructors. It restricts the calculations to the range n (the range of generators must be given). `GRange`[V] returns this limit. The results of operations in V are evaluated only if their grades are ≤ `GRange`[V].

□ GTimes

`GTimes` is the name of multiplication in the `Common` domain. The operation * is treated as `GTimes` when not all operands are defined as `Vectors` or `Scalars`. The operation `GTimes` is automatically replaced with `VTimes`, `SVTimes`, and `Times`, when all operands become vectors or scalars.

■ H

□ HamiltonianH

`HamiltonianH`[$\mathfrak{g}$] is the operator from functions to vector fields (implemented as differential operators) associated with the bracket in the Poisson algebra $\mathfrak{g}$.

□ Homogen

`Homogen`[$f \to r$] introduces the property f[..., $c * v$, ...] = $c^r * f$[..., v, ...] for a scalar c and a vector v. Except for the case $r = 0$, the function f should be declared as `Vector` or `Scalar`.

`Homogen`[f] is equivalent to `Homogen`[$f \to 1$].

`Homogen`[$f \to$ `First`] and `Homogen`[$f \to$ `Last`] introduce homogeneity (with degree 1) in only the first or only the last argument of f.

□ HomogenRule

`HomogenRule`[f, r] is the replacement rule $f[\ldots, c*v, \ldots] \to c^r * f[\ldots, v, \ldots]$, where c is any scalar and v any vector. The parameter r is optional, the default value is 1.

`HomogenRule`[f, `First`, r] and `HomogenRule`[f, `Last`, r] act only on the first/last argument of f.

□ HWModule

`HWModule`[$\mathfrak{m}$, $\mathfrak{g}$, w] builds the irreducible $\mathfrak{g}$–module $\mathfrak{m}$ with highest weight w. The algebra $\mathfrak{g}$ should be expressed as the direct sum of spaces $\mathfrak{g} = \mathfrak{g}_+ \oplus \mathfrak{g}_0 \oplus \mathfrak{g}_-$ using functions `Decomposition-List` and `DecompositionRule` with `CartanTriade` as the name of the decomposition.These functions are defined by some constructors of Lie algebras. The function `Grade` should be defined on $\mathfrak{g}$ and should agree with decomposition (the sign of the grade should discriminate the parts). The function `Weight` should be defined on $\mathfrak{g}$ and should agree with Cartan subalgebra $\mathfrak{g}_0$: $[h_i, g]$ = `Weight`[g][[i]] g.

`HWModule`[$\mathfrak{m}$, $\mathfrak{g}$, w, `Grade` → r] makes all calculations down to degree $-r$. The algebra $\mathfrak{g}$ should be graded. The degree of the highest vector is assumed to be 0.

`HWModule`[$\mathfrak{m}$, $\mathfrak{g}$, w, `Grade` → r, `Factor` → `False`] builds the Verma module with given highest weight (does not make the quotient modulo the maximal submodule).

Option `Order` → $\{y_1, \ldots, y_m\}$ gives the order in which the elements of $\mathfrak{g}_-$ should appear in the expressions of the basis of $\mathfrak{m}$ in terms of generators of $U(\mathfrak{g}_-)$. Default order is given by `Basis`[$\mathfrak{g}_-$].

Option P → p gives the parity of the highest vector (default is 0).

■ I

□ Ideal

`Ideal`[i, $\mathfrak{g}$, {g_1, ...}] defines the ideal $\mathfrak{i}$ in the Lie (super)algebra $\mathfrak{g}$, generated by elements g_1, ...

□ Image

`Image`[V] returns the list of images of basis elements of V if V is defined as a subspace (subalgebra, submodule).

□ InSpace

If a space V is defined as subspace (subalgebra, submodule), `InSpace`[V] returns the name of the ambient space.

■ J

□ Jacobi

`Jacobi`[$f \to g$] introduces the property (f acts as a bracket in Lie superalgebra):

$$f[x,\ g[y_1, y_2,\ ..]] = g[f[x, y_1], y_2,\ ..] \pm g[y_1, f[x, y_2],\ ...] \pm ...$$

`Jacobi`[$f \to \{g_1,\ ...\}$] introduces this property for every g_i.

□ JacobiRule

`JacobiRule`[f, g] returns the rule which expands $f[x,\ g[x_1,\ ...],\ ...]$ into the sum of $\pm\ g[\ ...,\ f[x_i,\ ...]\ ,\ ...]$ (like the bracket in the Lie superalgebra).
`JacobiRule`[f, $\{g_1,\ ...,\ g_k\}$] returns the list of rules for function g_1 , ..., g_k .

□ JoinSplit

`JoinSplit`[$e_1, e_2,\ ...$] joins terms of splitted lists $e_1, e_2,\ ...$.

■ K

□ Kb

`Kb`[x, y] is the Contact bracket (the operator in the contact algebra).

□ kb

`kb`[x, y] represents the Contact bracket (the operator in the contact algebra) but is not unevaluated.

□ KerSpace

`KerSpace`[U, V, f] calculates the kernel subspace $U = \{v \in V \mid f(v) == 0\}$. Here f is a linear function or a list of linear functions on V.

■ L

□ Lb

`Lb`[x, y] is the Lie bracket (the bracket in the Lie (super)algebra of vector fields).

□ lb

`lb`[x, y] represents the Lie bracket (the operator in the algebra of vector fields) in the unevaluated form.

□ LDer

`LDer`[e, x, *ptrn*] calculates the left partial derivative of expression e with respect to x. The pattern *ptrn* should match all independent and none of the dependent variables.

□ Leibniz

`Leibniz`[$f \to g$] introduces the property (f acts on g as a derivation of parity `P`[f]):

$$f @ g[x_1, x_2, ...] = g[f @ x_1, x_2, ...] \pm g[x_1, f @ x_2, ...] \pm ... .$$

`Leibniz`[$f \to \{g_1, ...\}$] introduces this property for every g_i.

□ LeibnizRule

`LeibnizRule`[f, g] returns the rule to expand $f[g[x_1, ...]]$ into the sum of $\pm\, g[..., f[x_i], ...]$ (so f acts like the derivative of parity `P`[f]).

`LeibnizRule`[f, $\{g_1, ..., g_k\}$] returns the list of rules for function g_1, ..., g_k.

□ LieAlgebra

`Options[LieAlgebra]` holds default options for constructors of Lie algebras.

□ Linear

`Linear`[f] introduces the (multi)linearity of f. The linearity is equivalent to the union of 3 properties: `Additive`, `Homogen`→1 and `ZeroArg`.

`Linear`[$f \to$ `First`] and `Linear`[$f \to$ `Last`] introduces linearity only in the first/last argument.

The function f should be declared vector– or scalar–valued and having all (resp. first, last) vector argument.

□ LinearChange

`LinearChange`[*expr*, *rule*] applies the *rule* (that may be also a list of rules) to the linear components of *expr* in an attempt to transform the expression.

□ LinearCollectRule

`LinearCollectRule`[f] returns the list of rules that may be used to collect together the term in the sum $f[x_1, y_1, ..] + f[x_2, y_2, ..] + ...$ wich differ in one argument only.

`LinearCollectRule`[f, `First`] and `LinearCollectRule`[f, `Last`] return the rules for linear collecting in the first or last argument of f.

The function f should be declared vector– or scalar–valued and having all (resp. first, last) vector argument. Though the function f is (mathematically) linear, it should not be declared linear, homogeneous or additive.

□ LinearRule

`LinearRule`[f] returns the list of rules for (multi)linear expansion of expressions containing f[...].

`LinearRule`[f, `First`] and `LinearRule`[f, `Last`] return the rules for linear expansion in the first/last argument of f only.

The function f should be declared vector– or scalar–valued and having all (resp. first, last) vector argument. It should not be declared linear.

□ LogPower

`LogPower`[f] introduces the property $f[x^r] = r * f[x]$. The function f should be declared vector– or scalar–valued.

`LogPower`[$f \to op$] introduces the property $f[x^r] = op[r, f[x]]$.

□ LogPowerPule

`LogPowerRule`[f] returns the rule $f[x^r] \to r * f[x]$. The function f should be declared vector– or scalar–valued.

`LogPowerRule`[f, op] returns the rule $f[x^r] \to op[r, f[x]]$.

■ M

□ Mapping

The option `Mapping` $\to f$ instructs the sub– and quotiontspace constructors to define *fn* as a mapping function (immersion or projection).

`Mapping`[U, V] returns the last defined map $U \longrightarrow V$.

□ MappingRule

`MappingRule`[U, V] is a replacement rule that implements the last defined `Mapping` $U \longrightarrow V$.

□ MapSplit

`MapSplit`[f, e] applies function f to the member of lists — the terms of splitted list e.

□ MatchList

`MatchList`[e, p] returns the list of different terms in expression e, matching the pattern p. `MatchList`[e, p, f] returns the list of values of different values of f[*term*].

□ Mb

`Mb`[x, y] is the Moebius bracket (the operator in the Moebius–Poisson superalgebra).

□ mb

`mb`[x, y] represents the unevaluated Moebius bracket (the operator in the Moebius–Poisson superalgebra).

□ MergeSplit

`MergeSplit`[fn, e_1, e_2, ...] merges splitted expressions (lists or sums). The terms t_1, ... t_n with the same selector are merged using $fn[t_1, \ldots t_n]$.

□ Mixed

The functions `Parity` and `Deg` return `Mixed` if the argument is not homogeneous.

□ MLeft

`MLeft` is the first item in the list of relative spaces that denotes "the space itself".
`MLeft`[$\mathfrak{m}$] returns $\mathfrak{m}$ for any vector space $\mathfrak{m}$.

□ MoebiusAlgebra

`MoebiusAlgebra`[g, {x, θ, t}] defines the Moebius–Poisson superalgebra g as the algebra on the space of polynomials in x_1, ... x_n, θ, and t. The x may be also a list of components, as in `PoissonAlgebra`.

□ MRight

`MRight`[$\mathfrak{m}$] returns the name of the module $\Pi \otimes \mathfrak{m} \otimes \Pi$, where Π is the 1–dimensional trivial odd module, or `None` if this module was not defined.
`MRight`[$\mathfrak{m} \to \mathfrak{r}$] builds the module $\mathfrak{r} = \Pi \otimes \mathfrak{m} \otimes \Pi$, where Π is the 1–dimensional trivial odd module.
`MRight` $\to \mathfrak{r}$ is an optional parameter to the constructor `VectorSpace`.

■ N

□ NewBracket

`NewBracket`[op] prepares op to be used as the bracket in a Lie superalgebra.

□ NewBrace

`NewBrace`[op] prepares op to be used as the bracket in a Lie superalgebra.

□ NewOverscript

`NewOverscript`[op, " s "] introduces an overscript notation for $op[x] \longleftrightarrow \overset{s}{x}$ in the standard and traditional formats.
`NewOverscript`[op, " s ", $sTeX$] introduces also TeX notation for s.

□ NewPower

`NewPower`[*op*, " *s* "] introduces superscript notation for $op[x, r] \longleftrightarrow x^{s\,r}$ in the standard and traditional formats and *x*^*s r* for the output format and for the input in any format. The *s* should be a single character (not a letter or digit).

`NewPower`[*op*, " *s* ", *sTeX*] introduces also TeX notation for *s*.

□ NewRelative

`NewRelative`[relation, *V* −> *W*] creates and attaches a new relative to the family of spaces−relatives. The first argument is the relation of the new space *W* to the old space *V*. The valid relations are `CoLeft`, `MRight`, `CoRight`, `PiRight`, `DLeft`, `PiLeft`, `DRight`.

□ NewSuperscript

`NewSuperscript`[*op*, " *s* "] introduces superscript notation for $op[x] \longleftrightarrow x^s$ in the standard and traditional formats.

`NewSuperscript`[*op*, " *s* ", *sTeX*] introduces also TeX notation for *s*.

□ NGen

`NGen`[$\mathfrak{m}$] is the number of generators of the module or algebra $\mathfrak{m}$.

■ O

□ Ob

`Ob`[*x*, *y*] is the odd Contact bracket (the operator in the "odd" contact algebra).

□ ob

ob[*x*, *y*] represents the unevaluated odd Contact bracket (the operator in the "odd" contact algebra).

□ OKAlgebra

`OKAlgebra`[$\mathfrak{g}$, {*x*, *y* , *t*}] defins the "odd" Contect algebra $\mathfrak{g}$ as the contact algebra of the polynomials in $x_1, \ldots, x_n, y_1, \ldots y_n$ and *t* with the odd contact bracket $\{., .\}_{m.b.}$. The Buttin bracket $\{., .\}_{B.b.}$ is also defined. The spaces *x* and *y* should be already defined (as for `ButtinAlgebra`).

□ Operator

If *s* denotes the passive form of some operator, `Operator`[*s*] returns the name of the active form of the same operator.

□ OpSymbol

If *s* denotes the active form of some operator, `OpSymbol`[*s*] returns the name of the passive form of the same operator.

□ Output

`Output`[$v \to f$] defines the fomat of exprerssion v[...] in the Output form as f[v[...]] .

`Output` $\to f$ is an optional parameter to space constructors that defines the output format of the space elements.

■ P

□ P

`P`[v] returns the parity of the even or odd vector v. For mixed (inhomogeneous) vector the result is indefinite.

□ Parity

`Parity`[v] checks if v is a homogenious vector and returns its parity. For Inhomogeneous vectors, and when checking fails, it returns `Mixed`.

□ PartSplit

`PartSplit`[*expr*, *sel*] and `PartSplit`[*expr*, *sel, default*] return the part of splitted expression *expr* with given selector *sel* or the *default* if the part not found. The default value for *default* is 0.

□ Pb

`Pb`[x, y] is the Poisson bracket (the operation in the Poisson algebra).

□ pb

`pb`[x, y] represents the unevaluated Poisson bracket (the operation in the Poisson algebra).

□ PDim

`PDim`[V] returns the list {*evenDim*, *oddDim*} of dimensions of the even and odd components of the vector superspace V.

`PDim`[V, r] returns the superdimension of the component of degree r in the graded space V.

□ PiLeft

`PiLeft`[$\mathfrak{m}$] returns the name of the module $\Pi \otimes \mathfrak{m}$, where Π is the 1−dimensional trivial odd module, or `None` if this module was not defined.

`PiLeft`[$\mathfrak{m} \to \mathfrak{l}$] builds the module $\mathfrak{l} = \Pi \otimes \mathfrak{m}$, where Π is the 1−dimensional trivial odd module.

`PiLeft` $\to \mathfrak{l}$ is an optional parameter to the constructor `VectorSpace`.

□ PiRight

`PiRight`[$\mathfrak{m}$] returns the name of the module $\mathfrak{m} \otimes \Pi$, where Π is the 1−dimensional trivial odd module, or `None` if this module was not defined.

`PiRight`[$\mathfrak{m} \to \mathfrak{r}$] builds the module $\mathfrak{r} = \mathfrak{m} \otimes \Pi$, where Π is the 1−dimensional trivial odd module.

`PiRight` $\to \mathfrak{r}$ is an optional parameter to the constructor `VectorSpace`.

□ PList

`PList` $\to \{p_1, \ ...\}$ is the optional parameter for some space constructors. It is the list of parities of either (a) the vectors forming a basis of the new space, (b) the generators of an algebra or a module, or (c) rows and columns of matrices.

`PList[V]` returns the parity list used in the definition of the space.

□ Plus2

`Plus2`[a, b, ...] is the sum $a + b + ...$ modulo 2.

□ PlusOp

`PlusOp`[*domain*] is the name of + in the domain.

□ Plus$

When parsing the user's input, the operation `Plus` is first replaced with `GPlus$` and then, depending on operands, it may be replaced with `Plus$` or `VPlus$`. In the last step the operations `Plus$`, `VPlus$` and `GPlus$` are replaced with, respectively, `Plus`, `VPlus` and `GPlus`

To ensure that the final operation will be `Plus` independently of operands, one can enter `Plus$` instead of `Plus`.

□ PoissonAlgebra

`PoissonAlgebra`[$\mathfrak{g}$, x] defines the Lie (super)algebra $\mathfrak{g}$ as the Poisson algebra on the space of polynomials in x_i, ..., $x_{2\,n}$ (the vector space x should be already defined) with the Poisson bracket

$$\{f, g\}_{P.b.} = -(-1)^{P[f]} \sum_{i=1}^{n} \Big(\frac{\partial f}{\partial x_i} \frac{\partial g}{\partial x_{2\,n+1-i}} - (-1)^{P[x_i]} \frac{\partial f}{\partial x_{2\,n+1-i}} \frac{\partial g}{\partial x_i} \Big).$$

`PoissonAlgebra`[$\mathfrak{g}$, $\{p, q\}$] defines the Lie (super)algebra $\mathfrak{g}$ as the Poisson algebra of the polynomials in p_1, ..., p_n, q_1, ..., q_n (the vector spaces p and q should be already defined and have the same superdimension) with the Poisson bracket

$$\{f, g\}_{P.b.} = \sum_{i=1}^{n} (-1)^{P[f]\,P[p_i]} \Big(\frac{\partial f}{\partial p_i} \frac{\partial g}{\partial q_i} - (-1)^{P[p_i]} \frac{\partial f}{\partial q_i} \frac{\partial g}{\partial p_i} \Big).$$

`PoissonAlgebra`[$\mathfrak{g}$, $\{\theta\}$] defines the Lie (super)algebra $\mathfrak{g}$ as the Poisson algebra of the polynomials in the odd variables θ_1, ..., θ_n (the vector space θ should be already defined) with the Poisson bracket

$$\{f, g\}_{P.b.} = (-1)^{P[f]} \sum_{i=1}^{n} \frac{\partial f}{\partial \theta_i} \frac{\partial g}{\partial \theta_i}$$

$$\{f, g\}_{P.b.} = (-1)^{P[f]} \sum_{i=1}^{n} \frac{\partial f}{\partial \theta_i} \frac{\partial g}{\partial \theta_i}.$$

PoissonAlgebra[$\mathfrak{g}$, {p_1, ... p_n, q_n, ... q_1}] and PoissonAlgebra[$\mathfrak{g}$, {p_1, ... p_n, θ, q_n, ... q_1}], where all p_i, q_i and θ are (super)spaces, define the Lie (super)algebra *alg* as the Poisson algebra of the polynomials on the direct sum of vector spaces spaces with the Poisson bracket equal to the sum of the Poisson brackets on pairs p_i, q_i and on θ.

PoissonAlgebra[$\mathfrak{g}$, x, {{c_1, i_1, j_1}, ..., {c_m, i_m, j_m}}] defines the Poisson algebra with the bracket

$$\{f, g\}_{P.b.} = -(-1)^{P[f]} \sum_{k=1}^{m} c_k \frac{\partial f}{\partial x_{i_k}} \frac{\partial g}{\partial x_{j_k}}.$$

□ PolyGrade

PolyGrade[v] is a polygrading function. For $\mathbb{Z}^n$–graded or filtered modules, PolyGrade[v] returns a list {g_1, ... g_n} that represents the $\mathbb{Z}^n$–grading or filtration of the vector v.

For filtered modules, the function is defined on the elements of the basis. For graded modules, the function is defined on homogeneous vectors.

The function PolyGrade is defined by some space constructors. When an algebra is built from generators, the PolyGrade[v] is defined as the degree of the expression of v in term of generators.

□ PowerOp

PowerOp[*op*] is the name of "power" operation, associated with "times" operation *op*.

PowerOp[*op* → *name*] defines this operation.

□ Power$

When parsing the user's input, the operation Power is first replaced with GPower$ and then, depending on the first operand, it may be replaced with Power$ or VPower$. In the last step the operations Power$, VPower$ and GPower$ are replaced with, respectively, Power, VPower and GPpower.

To ensure that the final operation will be Power independently of the first operands, one can enter Power$ instead of Power.

□ PreSL

PreSL is preprocessor that converts operations +, −, *, /, ^ to vector or scalar operations.

$Pre = PreSL turns the preprocessor on. $Pre =. turns the preprocessor off.

□ pslAlgebra

pslAlgebra[$\mathfrak{g}$, V] defines the algebra $\mathfrak{g} = \mathfrak{psl}(V) = \mathfrak{sl}(V) / \langle \text{center} \rangle$. The even and odd components of V should have the same dimension.

pslAlgebra[$\mathfrak{g}$, Dim → n] and pslAlgebra[$\mathfrak{g}$, PList → {p_1, ... p_n}] defines the algebra $\mathfrak{g} = \mathfrak{psl}(n) = \mathfrak{sl}(n) / \langle \text{center} \rangle$. The dimension may be a number, an expression, or a list {n_1, n_2, ...}, as for VectorSpace.

□ psq2Algebra

`psq2Algebra`[g, n] defines the superalgebra $g = \mathfrak{psq}(n)^{(2)}$ with basis $g_i[k] = g_i\,\tau^k$, $g_{i,j}[k] = g_{i,j}\,\tau^k$.

□ psqAlgebra

`psqAlgebra`[g, {x, y}, n] defines the superalgebra $g = \mathfrak{psq}(n)$ with basis x_i, $x_{i,j}$ (the even elements) and y_i, $y_{i,j}$ (the odd elements).

■ Q

□ q2Algebra

`q2Algebra`[g, n] defines the superalgebra $g = \mathfrak{q}(n)^{(2)}$ with basis $g_{i,j}[k] = g_{i,j}\,\tau^k$.

□ qAlgebra

`qAlgebra`[g, {x, y}, n] defines the superalgebra $g = \mathfrak{q}(n)$ with basis $x_{i,j}$ (the even elements) and $y_{i,j}$ (the odd elements).

□ QuotientModule

`QuotientModule`[$\mathfrak{f}$, $\mathfrak{G}$, $\mathfrak{g}$, $\mathfrak{h}$] builds the quotient module $\mathfrak{f} = \mathfrak{g}/\mathfrak{h}$, where $\mathfrak{g}$ and $\mathfrak{h}$ are defined as submodules in $\mathfrak{G}$ and $\mathfrak{h} \subset \mathfrak{g} \subset \mathfrak{G}$.

Option `Split`→f instructs to split the space in order to accelerate calculations.

`QuotientModule`[$\mathfrak{f}$, $\mathfrak{G}$, $\mathfrak{g}$, $\mathfrak{h}$, *splitFn*] builds a graded submodule using splitting function *splitFn*.

■ R

□ RamondAlgebra

`RamondAlgebra`[g, {x, θ, t}] defines the Ramond superalgebra g as the algebra of the polynomials in x_1, ... x_n, θ, and t. The x may be also a list of components, as in `PoissonAlgebra`.

□ RamondD

`RamondD`[f, t, θ, *ptrn*] is the $\mathcal{D}$ operator used in the definition of the Ramond superalgebra. The pattern *ptrn* should match all independent variables.

□ RamondK

`RamondK`[*alg*] is the operator from functions to vector fields (implemented as differential operators) associated with the bracket in the Ramond superalgebra.

□ Rank

`Rank`[T] returnt the rank of the tensor space T.

□ Rb

`Rb`[*x*, *y*] is the Ramond bracket (the operator in the Ramond superalgebra).

□ rb

`rb`[*x*, *y*] represents the unevaluated Ramond bracket (the operator in the Ramond superalgebra).

□ ReGrade

`ReGrade`[*V*, *grading*] changes the grading on the space *V* and all known relatives of *V*.

The parameter *grading* may be either a list of new degrees for the elements of the basis (or for generators only if an algebra or a module is built using generators), or the name (a symbol or a number) of a particular grading predefined by the space constructor of *V*.

□ Regular

One of possible modes of the definition of the bracket and action.

□ RemoveOverscript

`RemoveOverscript`[*op*] removes the overscript notation (such as $\overline{x}$) for the operation *op*[*x*].

□ RemovePower

`RemovePower`[*op*] removes the power–like notation (such as $x^{\otimes n}$) for the operation *op*[*x*, *n*].

□ RemoveSuperscript

`RemoveSuperscript`[*op*] removes the superscript notation (such as $x^{\dagger}$) for the operation *op*[*x*].

□ RestrictModule

`RestrictModule`[*V*, $\mathfrak{g}$] defines an action of $\mathfrak{g}$ on *V*, if *V* is a defined as a subspace of some $\mathfrak{g}$–module.

■ S

□ Scalar

`Scalar`[*c*, ...] declares that *c*, ... are scalars. Any expression *c*[...] whose header is declared as scalar, is also a scalar. So `Scalar`[*f*] may be used to declare function *f* scalar–valued.

`Scalar`[$f \to n$] declares that the *n*–th operand of *f* is scalar. Here *n* may be a number, `First`, `Last`, `All` or _.

□ ScalarEquation

`ScalarEquation`[*equ*] converts the equation or a system of vector equations *equ* with scalar variables into a system of scalar equations.

□ ScalarQ

`ScalarQ`[*c*] and `ScalarQ`[*c*[...]] returns `True` if *c* was declared `Scalar`.

□ SeqForm

`SeqForm`[e_1, ...] [*h*[a_1, ..]] prints the sequence e_1, ..., substituting *h*, a_1, ... instead of placeholders `#0`, `#1`, ...

□ SimplifySign

`SimplifySign`[*expr*] simplifies the expressions $(-1)^{polynom}$ in *expr*.

□ SimplifySignRule

`SimplifySignRule` is the replacement rule for simplifying $(-1)^{polynom}$.

□ SkewSymmetric

`SkewSymmetric`[*f*] introduces an automatical sorting of arguments of *f* using super−skew−symmetry

$$f[\,..,\,y,\,x,\,..\,] = (-1)^{(1+P[x])\,(1+P[y])}\,f[\,..,\,x,\,y,\,..\,].$$

□ SkewSymmetricQ

`SkewSymmetricQ`[*f*] returns `True` if *f* was declared skew−symmetric.

□ SkewSymmetricRule

`SkewSymmetricRule`[*f*] is the replacement rule for sorting of arguments of *f* using super−skew−symmetry

$$f[\,..,\,y,\,x,\,..\,] \to (-1)^{(1+P[x])\,(1+P[y])}\,f[\,..,\,x,\,y,\,..\,].$$

□ SkipVal

`SkipVal` is a special value of selection function *f* used in `SplitSum` and `SplitList`. If, for a member *m* of the splitting expression, *f*[*m*] == `SkipVal`, then *m* is omitted.

□ slAlgebra

`slAlgebra`[$\mathfrak{g}$, V] defines the Lie (super)algebra $\mathfrak{g} = \mathfrak{sl}(V)$ of traceless matrices on the space V.

`slAlgebra`[$\mathfrak{g}$, `Dim` → n] defines the algebra $\mathfrak{g} = \mathfrak{sl}(n)$ of traceless $n * n$ matrices. The dimension may be a number, expression, or a list $\{n_1, n_2, ...\}$, as for `VectorSpace`.

□ SpacePlus

`SpacePlus`[W, $\{V_1, ... V_n\}$] defines the space $W = V_1 \oplus ... \oplus V_n$ as the direct sum of the terms.

□ Split

The optional parameter `Split` → `f` instructs the space constructor to split the space into homogeneous components. The value of f is a constant on each component. Splitting increases the speed of calculations.

□ SplitList

`SplitList`[*list*, *ptrn*] transforms the list of vectors $\{c_1 * v_1, c_2 * v_2, ...\}$ with $v_1, v_2, ...$ matching pattern *ptrn*, gathering terms with equal v's. The result is the list $\{v_{\sigma_1} \to \{c_{\tau_{11}}, c_{\tau_{12}}, ..\}, v_{\sigma_2} \to \{...\}, ...\}$, where $\{v_{\sigma_1}, v_{\sigma_2}, ...\}$ is a sorted list of different v_i's in the *list* and $\{c_{\tau_{i,1}}, ...\}$ is the list of coefficients of all vectors v_k equal to v_{σ_i}.

`SplitList`[*list*, *ptrn, f*] transforms the list of vectors $\{c_1 * v_1, c_2 * v_2, ...\}$ with $v_1, v_2, ...$ matching pattern *ptrn* into a list $\{f_1 \to \{e_{11}, ...\}, f_2 \to \{e_{21}, ...\}, ...\}$, where $\{f_1, f_2, ...\}$ is a sorted list of different values of $f[v_i]$ and $\{e_{i1}, ...\}$ is the list of the members of *list* giving value f_i of the function f (not including members with $f[v_i]$ == `SkipVal`).

□ SplitSum

`Split Sum`[*sum*, *ptrn*] transforms the vector sum $c_1 * v_1 + c_2 * v_2 + ...$ with $v_1, v_2, ...$ matching pattern *ptrn*, gathering terms with equal v's. The result is the list $\{v_{\sigma_1} \to s_1, v_{\sigma_2} \to s_2, ...\}$, where $\{v_{\sigma_1}, v_{\sigma_2}, ...\}$ is a sorted list of different v_i's in the *sum* and s_i is the sum of coefficients of all vectors v_k equal to v_{σ_i}.

`Split Sum`[*sum*, *ptrn, f*] transforms the vector sum $c_1 * v_1 + c_2 * v_2 + ...$ with $v_1, v_2, ...$ matching pattern *ptrn* into a list $\{f_1 \to s_1, f_2 \to s_2, ...\}$, where $\{f_1, f_2, ...\}$ is a sorted list of different values of $f[v_i]$ and $\{e_{i1}, ...\}$ is the sum of the members of *sum* giving value f_i of function f (not including members with $f[v_i]$ == `SkipVal`).

□ sqAlgebra

`sqAlgebra`[g, {x, y}, n] defines the Lie superalgebra $g = \mathfrak{sq}(n)$ with basis $x_{i,j}$ (the even elements), y_i (the odd diagonal elements), and $y_{i,j}$ (the odd off–diagonal elements, $i \neq j$).

□ Standard

`Standard`[$v \to f$] defines the output format of the expression v[...] in Standard form as f[v[...]] .

`Standard` $\to f$ is an optional parameter to space constructors that defines the Standard output format of the space elements.

The output in the standard form produced by the function `Standard` may not be reused for the input. Use `MakeBoxes` and `InterpretationBox` to define reusable output forms.

`Standard`[$v \to$ `Subscripted`] defines also the interpretation of the subscripted v in Standard input format: $\mathtt{v}_{\mathtt{i},\ldots} \longrightarrow \mathtt{v[i,\ ...]}$.

□ STimesOp

`STimesOp`[*domain*] is the name of $*$ in *Scalar* $*$ *domain*.

□ StopUseAsSymbol

`StopUseAsSymbol`[*expr*] cancels usage of the expression *expr* (e.g., $\tilde{v}$) as a single symbol.

□ SubAlgebra

`SubAlgebra`[$\mathfrak{h}$, $\mathfrak{g}$, $\{g_1,\ ...\,(,\ g)_k\}$] or `SubAlgebra`[$\mathfrak{h}$, $\mathfrak{g}$, $\{h_1 \to g_1,\ ...\,(,\ h)_k \to g_k\}$] builds the subalgebra $\mathfrak{h}$ of the Lie (super)algebra $\mathfrak{g}$ generated (the subalgebra) by the elements $g_1,\ ...\, g_k \in \mathfrak{g}$. The second forms gives also names for the generators of $\mathfrak{g}$.

Specific options:

`GRange` $\to r$ is the maximal degree of relations (default is `Infinity`);

`Grade` $\to fn$ defines a grading on $\mathfrak{h}$: `Grade`[h] = fn[image[h]] (default is `Grade`).

`Grade` $\to r$ computes up to grade r (r should be a number; the `Grade` function must be defined in $\mathfrak{g}$).

`Grade` $\to \{fn, r\}$ computes up to grade r and defines a grading on $\mathfrak{h}$.

`Weight` $\to fn$ defines a weight on $\mathfrak{h}$: `Weight`[h] = fn[image[$\mathfrak{h}$]] (default is `Weight`).

The function `SubAlgebra` calculates the basis of $\mathfrak{h}$ in terms of generators and relations between the generators. They are stored as respective properties `GenBasis` and `GenRel` of algebra $\mathfrak{h}$.

□ SubModule

`SubModule`[$\mathfrak{n}$, $\mathfrak{m}$, $\{v_1,\ ...,\ v_k\}$] buillds the submodule $\mathfrak{n}$ of the module $\mathfrak{m}$ generated (the submodule) by the elements $v_1,\ ...,\ v_k \in \mathfrak{m}$.

`SubModule`[$\mathfrak{n}$, $\mathfrak{m}$, $\{v_1,\ ...,\ v_k\}$, `Split` $\to wt$] buillds a graded submodule $\mathfrak{n} \subset \mathfrak{m}$ generated by the elements $v_1,\ ...,\ v_k \in \mathfrak{m}$. The argument wt is the grading function. Graded submodules are calculated faster than non−graded ones.

`SubModule`[$\mathfrak{n}$, $\mathfrak{m}$, $\{v_1,\ ...,\ v_k\}$, `Algebra` $\to \mathfrak{g}$] should be used if more than one algebra acts on the module $\mathfrak{m}$. The function is implemented for finite dimensional algebras and modules with 1−index bases.

□ SubSpace

`SubSpace`[$U,\ V,\ \{v_1,\ ...\}$] defines a subspace $U \in V$ with a given basis $\{v_1,\ ...\}$.

□ SumOp

`SumOp`[*op*] returns the name of "Sum" function, associated with the "plus" operation *op*.

`SumOp`[*op* → *name*] defines this operation.

□ SVExpandRule

expr /. `SVExpandRule` expands out all scalar coefficients in $c * v$.

□ SVFactorRule

expr /. `SVFactorRule` factorizes all scalar coefficients in $c * v$.

□ SVNormalRule

expr /. `SVNormalRule` converts all scalar coefficients in $c * v$ to the normal form using the function `$SNormal`.

□ SVSimplifyRule

expr /. `SVSimplifyRule` simplifies all scalar coefficients in $c * v$.

□ SVSolve

`SVSolve`[*eqns*, *vars,* ...] attempts to solve a vector equation or set of equations *eqns* for the scalar variables *vars*. For other parameters and options, see `Solve`.

`SVSolve`[*eqns*] treats all non−vector variables encountered as *vars* above.

□ Symmetric

`Symmetric`[f] introduces an automatic sorting of arguments of f using supersymmetry

$$f[\,..,\,y,\,x,\,..] = (-1)^{P[x]\,P[y]}\,f[\,..,\,x,\,y,\,..].$$

□ SymmetricNormal

`SymmetricNormal`[*e*] returns the normal form of the vector expression, assuming the supersymmetry of the vector product `VTimes`. See also `VNormal`.

□ SymmetricQ

`SymmetricQ`[f] returns `True` if f was declared `Symmetric`.

□ SymmetricRule

`SymmetricRule`[f] is the replacement rule for sorting of arguments of f using supersymmetry

$$f[\,..,\,y,\,x,\,..] \to (-1)^{P[x]\,P[y]}\,f[\,..,\,x,\,y,\,..].$$

■ T

□ Tabular

One of possible modes of the definition of the bracket and action. These operations are defined via a "multiplication table".

□ TCollect

`TCollect`[*expr*, *case*] collects together terms $a ** b$ with the same a (if *case*=`First`) or with same b (if *case*=`Last`).

`TCollect`[*expr*, *case*, *op*] collect terms of operation *op* rather than $**$. The operation should be declared neither `Additive` nor `Linear`.

□ TensorSpace

`TensorSpace`[T, V, {C_1, C_2, ..}] defines the space $T = C_1 \otimes C_2 \otimes ...$. Repeated components may be written as C^n. All components must be relatives of the space V. The basis of T is $T[i_1, i_2, ...]$.

□ TestFirst

`TestFirst`[f] introduces the property $f[x_1 + ...] = f[x_1]$. This property reduces the time of the calculation of $f[x_1 + ...]$, but restricts f to homogeneous arguments only.

□ TestFirstRule

`TestFirstRule`[f] is the replacement rule $f[x_1 + ...] \to f[x_1]$. It may be applied to the homogeneous (with respect to f) vector sum.

□ TeX

`TeX`[$v \to f$] defines the output format of expression v[...] in TEX form as $f[v[...]]$.

`TeX` $\to f$ is an optional parameter to space constructors that defines the TEX output format of the space elements.

□ TheAlgebra

`TheAlgebra`[m] is the name of the algebra acting on the module m.

□ TheModule

`TheModule`[v] is the name of the module hosting the vector v.

□ TheSpace

For any space name or other vector symbol or expression x, `TheSpace`[x] returns the name of original space from which x is derived.

□ ThreadGraded

`ThreadGraded`[f] introduces the property $f[a \circ b] = f[a] + f[b]$ for any graded operation $\circ$. The function f should be declared to be scalar or vector.

`ThreadGraded`[$f \to plus$] introduces the property $f[a \circ b] = plus[f[a], f[b]]$.

□ ThreadGradedRule

`ThreadGradedRule`[f] is the replacement rule $f[a \circ b] \to f[a] + f[b]$ for any graded operation $\circ$. The function f should be declared to be scalar or vector.

`ThreadGradedRule`[f, $plus$] is the replacement rule $f[a \circ b] \to plus[f[a], f[b]]$

□ Times2

`Times2`[a, b, ...] is the product $a * b * ...$ modulo 2.

□ TimeString

`TimeString`[] returns a string representing the current time.

□ Times$

When parsing the user's input, the operation `Times` is first replaced by `GTimes$` and then, depending on operands, may be replaced with `Times$`, `VTimes$`, or `SVTimes$`. In the last step the operations `Times$`, `VTimes$`, `SVTimes$`, and `GTimes$` are replaced respectively with `Times`, `VTimes`, `SVTimes`, and `GTimes`.

To ensure that the final operation will be `Times` independently of operands, one can enter `Times$` instead of `Times`.

□ Tp

`Tp`[e_1, e_2, ...] or e_1 ** e_2 ** ... denotes the tensor product of e_1, e_2, ... in the cases where it is not regarded as an operation in the tensor algebra (for example, $f[x]$ ** dx). In the internal *Mathematica* format the operation `NonCommutativeMultiply` is used instead of `Tp`.

□ tPower

`tPower`[v, n], v^⊗n, or $v^{\otimes n}$ is the n–th tensor power of v.

□ Traditional

`Traditional`[$v \to f$] defines the output format of the expression v[...] in the `Traditional` form as $f[v[\ ...]]$.

`Traditional` → f is an optional parameter to space constructors that defines the `Traditional` output format of the space elements.

□ TrivialSpace

`TrivialSpace`[z] defines the space z of dimension 1 and basis {z}.
`TrivialSpace`[ζ, `Odd`] defines the space ζ of dimension 0|1 and basis {ζ}.

■ U

□ UnAdditive

`UnAdditive`[f] cancels the additive expansion of f[.., $x + y$, ..].
`UnAdditive`[$f \to$ `First`] and `UnAdditive`[$f \to$ `Last`] cancel the additive expansion of the first (last) argument of f.

□ UnAntiSkewSymmetric

`UnAntiSkewSymmetric`[f] cancels the automatic sorting of the operands of f.

□ UnAntiSymmetric

`UnAntiSymmetric`[f] cancels the automatic sorting of the operands of f.

□ UnDegTimes

`UnDegTimes`[op] cancels the property `Deg`[op[v_1, ... v_n], x] = `Deg`[v_1, x] + ... + `Deg`[v_1, x] .

□ UnGraded

`UnGraded`[op] cancels the property `Grade`[$a \sim op \sim b$] = `Grade`[a] + `Grade`[b].

□ UnHomogen

`UnHomogen`[f] cancels the property f[..., $c * v$, ...] = $c^r * f$[..., v, ...] for a scalar c and vector v.
`UnHomogen`[$f \to$ `First`] and `UnHomogen`[$f \to$ `Last`] cancels homogeneity in the first or last argument of f only.

□ UniqueCounters

`UniqueCounters`[*expr*] returns the *expr* with all counters in all sums and tables replaced with a unique symbol.

□ UnJacobi

`UnJacobi`[$f \to g$] cancels the property of action of f on g[...] as a bracket in Lie superalgebra.
`UnJacobi`[$f \to \{g_1, ...\}$] cancels this property for every g_i.

□ UnLeibniz

UnLeibniz[$f \to g$] cancels the property of action of f on g[...] as a derivation.
UnLeibniz[$f \to \{g_1, ...\}$] cancels this property for every g_i.

□ UnLinear

UnLinear[f] cancels the automatic (multi)linear expansion of f[...].
UnLinear[$f \to$ First] and UnLinear[$f \to$ Last] cancels the linearity in the first/last argument only.

□ UnLogPower

UnLogPower[f] cancels the property $f[x^r] = r * f[x]$.
UnLogPower[$f \to op$] cancels the property $f[x^r] = op[r, f[x]]$.

□ UnOutput

UnOutput[v] cancels the output format of the expression v[...] defined by Output .

□ UnScalar

UnScalar[c, ...] cancels the declaration of c as a scalar. See also Scalar and ScalarQ.

□ UnSkewSymmetric

UnSkewSymmetric[f] cancels the automatic sorting of the operands of f.

□ UnStandard

UnStandard[op] cancels the format of expression op[...] in the Standard form defined by Standard.

□ UnSymmetric

UnSymmetric[f] cancels the automatic sorting of the operands of f. See also Symmetric, SymmetricRule.

□ UnTestFirst

UnTestFirst[f] cancels the property $f[x_1 + ...] = f[x_1]$, so the function f may be applied to non–homogeneous vector sum. See also TestFirst.

□ UnTeX

UnTeX[v] cancels the format of the expression v[...] in TEX form defined by TeX.

□ UnThreadGraded

`UnThreadGraded`[f] cancels the property $f[a \circ b] = f[a] + f[b]$ for any graded operation $\circ$.
`UnThreadGraded`[$f \to plus$] cancels the property $f[a \circ b] = plus[f[a], f[b]]$.

□ UnTraditional

`UnTraditional`[op] cancels the format of the expression op[...] in the `Traditional` form defined by `Traditional`.

□ UnVector

`UnVector`[v, ...] cancels the declaration of v as a vector. See also `Vector` and `VectorQ`.

□ UnZeroArg

`UnZeroArg`[f] cancels the property f[..., 0, ...] = 0 introduced by `ZeroArg`[f].
`UnZeroArg`[$f \to$ `First`] and `UnZeroArg`[$f \to$ `Last`] cancel the properties introduced by respectively `ZeroArg`[$f \to$ `First`] and `ZeroArg`[$f \to$ `Last`].

□ UpToDegreeBasis

`UpToDegreeBasis`[d, {x_1, ..., x_n}] returns the list of elements of degree $\leq d$ in the symmetric algebra in x_1, ..., x_n.

`UpToDegreeBasis`[d, {x_1, ..., x_n}, op] uses op instead of `VTimes`. The operation op must be `Flat` and `Listable`.

□ UseAsSymbol

`UseAsSymbol`[e] allows to use the expression e as a single symbol. So e may be used, e.g., as a module's name. Typically, expressions in a symbol–like standard form (such as $\hat{x}$) are used as symbols.

■ V

□ VBasis

`VBasis`[$expr$] returns the list of different linear vector terms encountered in the expression.

□ VCollect

`VCollect`[$expr$] tries to simplify the expression $expr$ by collecting together the terms $c_i * v$ with the same v.

□ Vector

`Vector`[*v*, *w* ...] declares that *v*, *w*, ... are vectors. Any expression *v*[...] whose header is declared as vector, is also vector. So `Vector`[*f*] may be used to declare function *f* vector−valued.

`Vector`[*f* → *n*] declares that the *n*−th operand of *f* is a vector. Here *n* may be a number, `First`, `Last`, `All` or _.

□ VectorLieAlgebra

`VectorLieAlgebra`[$\mathfrak{g}$, *x*] defines a Lie (super)algebra $\mathfrak{g} = \mathfrak{vect}(x)$ of vector fields on the space *x* and its action on the (super)space of polynomials and polyvectors on *x*. The space *x* should be defined as well as the space of left even forms on *x* (`CoLeft` space).

The basis of $\mathfrak{g}$ is {p_i[*x*] ∗∗ *v*[*j*]}, where {p_i[*x*]} is the basis of polynomials on *x* and {*v*[*j*]} is the basis of `CoLeft`[*x*] (the space of left even linear forms on *x*). The name of the Lie bracket and the action is (by default) `Lb`, the unevaluated form is `lb`.

Unless the space *x* was already graded, the standard grading is defined on *x* assuming `Deg`[x_i] = 1. The grading of *x* induces a gradinng of $\mathfrak{g}$.

The algebra $\mathfrak{g}$ may be regraded by calling `ReGrade`[$\mathfrak{g}$, *grading*]. The predefined gradings are numbered from 0 to the odd dimension of *x*. In the *i*−th grading, the first *i* odd element of the basis of *x* have degree 0, all other x_j ' *s* have degree 1.

The following options alter the default operation names:

`Lb` is the name of Lie bracket (the default is `Lb`)

`CTimes` is the name of the tensor multiplication `**` (the default is `NonCommutativeMultiply`, a synonym of `Tp`)

`Wedge` is the multiplication of polyvectors (the default is `wedge`).

□ VectorQ

`VectorQ`[*x*] returns `True` if *x* is an object of `Vector` domain.

□ VectorSpace

`VectorSpace`[*V*] defines the vector space *V*. The dimension and parities may be given by optional parameters `Dim` → *d* or `PList` −> {p_1, ... p_n}.

□ VExpand

`VExpand`[*expr*] expands out all `VTimes` and `SVTimes` products in *expr*.

□ VExpandRule

expr //. `VExpandRule` expands out all `VTimes` and `SVTimes` products in *expr*.

□ VIf

`VIf`[*cond*, *v*] is a vector−valued version of the `If` function. It is evaluated to *v* if *cond* == `True` and to 0 if *cond* == `False`. A number of rules of symbolic evaluation works with unresolved `VIf`[...].

□ VNormal

`VNormal`[*expr*] tries to convert a vector expression to the standard (normal) form.

It expands out all `VTimes` and `SVTimes` products in *expr,* collects together terms $c_i * v$ with the same v, converts the scalar coefficients to the standard form.

□ VOrder

`VOrder`[v_1, v_2] returns $0, +1, -1$ depending on the order of vectors v_1 and v_2 .The scalar coefficients are ignored. See also the *Mathematica* function `Order`.

□ VOrderQ

`VOrderQ`[v_1, v_2] returns `True` or `False` depending on the order of vectors v_1 and v_2 .The scalar coefficients are ignored. See also the *Mathematica* function `OrderedQ`.

□ VPlus

`VPlus`[v_1, v_2, ...] is the vector sum of v_1, v_2, ... The expression $v_1 + v_2 + ...$ typed in the notebook is converted to `VPlus`[v_1, v_2, ...] if all terms are vectors.

□ VPower

`VPower`[v, n] is the n–th symmetric power of vector v. The expressions $v\hat{}n$ and v^n typed in the notebook are converted to `VPower`[v, n] if v is vector.

□ VSameQ

`VSameQ`[v_1, v_2] returns `True` if $v_1 = c_1 * v$, $v_2 = c_2 * v$ with same v and scalars c_1 , c_2 and `False` otherwise.

□ VSolve

`VSolve`[*eqns*, *vars,* ...] attempts to solve an equation or set of equations for the vector variables *vars*. For other parameters and options, see `Solve`.

`VSolve`[*eqns*] treats all vector variables encountered as *vars* above.

□ VSort

`VSort`[$v_1 + ...$] sorts the terms in the vector sum in the alphabetical order. The scalar coefficients are irrelevant for sorting.

□ VSum

VSum[*expr*, *iter*, ...] is the vector−valued version of the Sum function. A number of rules of symbolic evaluation works with unresolved VSum[...].

For both Sum and VSum, an alternative iterator $\{i, to \to from\}$ is defined. If the difference $diff = from - to$ is a number, this iterator is replaced with

(a) $\{i, from, to - 1\}$ if $diff > 0$;
(b) $\{i, to, from - 1\}$ and the whole sum is multiplied by -1 if $diff < 0$.
(c) the whole sum is replaced with 0 if $diff = 0$;

□ VTimes

VTimes[v_1, v_2, ...] is the product of vectors of v_1, v_2, ... The vector product may have various meanings. It may be used for any associative operation. The expression $v_1\ v_{2.}$ .., $v_1 * v_2 * ...$, or $v_1 \times v_2 \times ...$ typed in the notebook is converted to VTimes[v_1, v_2, ...] if all factors are vectors.

The vector product is non−symmetric by default, i.e., the terms are not sorted automatically. Use Symmetric and UnSymmetric to set and cancel the (super)symmetric sorting of terms.

■ W

□ Wedge

Wedge[v_1, v_2, ...] or $v_1 \wedge v_2 \wedge ...$ is the exterior multiplication (the operation in the exterior algebra). The operation is assotiative. The "power" operation for Wedge is not defined. The evaluation rules of Wedge are Linear, Symmetric and IdArg. After sorting the operands, Wedge is replaced with wedge.

□ wedge

wedge[e_1, ..., e_n] is the internal representation of the basis of exterior algebras. The external representation is $e_1 \wedge ... \wedge e_n$.

See also Wedge.

□ Weight

Weight[v] is the weight of the vector v. The weight is defined for homogeneous elements only.

□ WeightMark

WeightMark[*length*, m_1, ...] returns a list of given *length*. All elements of the result are initially set to 0. For every mark m_i, if $m_i > 0$, then the m_i−th element of the result is increased by 1. If $m_i < 0$, then the $(-m_i)$−th element is diminished by 1.

□ WithoutPreSL

Use WithoutPreSL[*expr*; ...] to prevent preprocessing of expressions when the preprocessing is turned on.

□ WithUnique

`WithUnique`[{s_1, ...}, *expr*] evaluates *expr* replacing the symbols s_1, ... with the new symbols with unique names.

■ Z

□ ZeroArg

`ZeroArg`[f] introduces the property f[..., 0, ...] = 0.
`ZeroArg`[$f \to$ `First`] and `ZeroArg`[$f \to$ `Last`] acts on the first/last argument of f only.

□ ZeroArgRule

`ZeroArgRule`[f] returns the replacement rule f[..., 0, ...] $\to$ 0.
`ZeroArg`[$f \to$ `First`] and `ZeroArg`[$f \to$ `Last`] return rules f[0, ...] $\to$ 0 and f[..., 0] $\to$ 0.

□ ZId

`ZId` is the identity operator. It is used in symbolic calculations.

□ ZLDer

`ZLDer`[x, *ptrn*] is the operator of left partial derivative with respect to x. `ZLDer`[x, *ptrn*][*expr*] is the left partial derivative of expression *expr* with respect to x. The pattern *ptrn* should match all the independent and none of the dependent variables. `ZLDer`[x, *ptrn*] may be used in symbolic calculations.

□ ZRamondD

`ZRamondD`[t, θ, *ptrn*] returns the Ramond operator $\mathcal{D}$ as a differential operator acting on the space of Laurent polynomials in t, θ, and the indeterminates matching the pattern *ptrn*.

■ $

□ $DPrint

`DPrint`[*level*, *data* ...] prints *data* (using `Print`) if `$DPrint` $\geq$ *level*.

□ $DPrintLabel

The value of `$DPrintLabel`[] is printed as a label for debug printing. Use `$DPrintLabel` = `DateString` or `TimeString` to use [date and] time as label. Use `$DPrintLabel` = `None` for debug printing without labels.

□ $EnvLess

The value of `$EnvLess` is the default sorting function for the product in enveloping algebras. The default value of `$EnvLess` is `OrderedQ[{#1,#2}]&`

□ $SNormal

The value of `$SNormal` is the user−defined function which is called by `VNormal` to convert the scalar coefficients to the normal form. It should always convert to zero the scalars that are really equal to zero. The default value of `$SNormal` is `Expand`.

□ $Solve

The value of `$Solve` is the user−defined function for solving the scalar equations. The default setting is `$Solve = Solve`.

Classical Lie Superalgebras in SuperLie

Here we describe how to define classical Lie superalgebras when working in **SuperLie**.

■ Matrix algebras

■ General matrix algebra

Algebra $\mathfrak{gl}$(m|n):

```
glAlgebra[name, Dim → {m, n}]
```

Algebra $\mathfrak{gl}$(par) with given format par $= \{p_1, \ ...\}$, $p_i \in \{0, 1\}$:

```
glAlgebra[name, PList → par]
```

Algebra of linear operators on given superspace V with basis V[1], ..., V[n]:

```
glAlgebra[name, V]
```

■ Special matrix algebra

Algebra $\mathfrak{sl}$(m|n):

```
slAlgebra[name, Dim → {m, n}]
```

Algebra $\mathfrak{sl}$(par) with given format par $= \{p_1, \ ...\}$, $p_i \in \{0, 1\}$:

```
slAlgebra[name, PList → par]
```

Algebra of supertraceless linear operators on given superspace V with basis V[1], ..., V[n]:

```
slAlgebra[name, V]
```

Algebra $\mathfrak{sl}$(m|n) given by means of Cartan matrix, for example,
Algebra $\mathfrak{sl}$(2|2):

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0}, {-1, 0, 1}, {0, -1, 2}}, ∞, PList → {0, 1, 0}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 0, 1}, {0, -1, 0}}, ∞, PList → {1, 1, 1}]
```

Algebra $\mathfrak{sl}$(1|3):

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 2, -1}, {0, -1, 2}}, ∞, PList → {1, 0, 0}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 0, 1}, {0, -1, 2}}, ∞, PList → {1, 1, 0}]
```

■ Algebra $\mathfrak{psl}$(m|m)

Algebra $\mathfrak{psl}$(m|m):

```
pslAlgebra[name, Dim → {m, m}]
```

Algebra $\mathfrak{psl}$(par) with given format par = $\{p_1, ...\}$, $p_i \in \{0, 1\}$, where the number of 0's and 1's should be equal:

```
pslAlgebra[name, PList → par]
```

Algebra of supertraceless linear operators on the given superspace *V* with basis *V*[1], ..., *V*[n]; the operators are given modulo scalar operators:

```
pslAlgebra[name, V]
```

■ Algebra $\mathfrak{osp}$(m|2n)

Not implemented as matrix algebra. May be defined as Lie superalgebra with Cartan matrix:

Algebra $\mathfrak{osp}$(3|2)

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1}, {-2, 2}}, ∞, PList → {1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1}, {-1, 1}}, ∞, PList → {1, 1}]
```

Algebra $\mathfrak{osp}$(2|4)

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 2, -2}, {0, -1, 2}}, ∞, PList → {1, 0, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 0, 2}, {0, -1, 2}}, ∞, PList → {1, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, -2, 1}, {-2, 0, 1}, {-1, -1, 2}}, ∞, PList → {1, 1, 0}]
```

Algebra $\mathfrak{osp}$(3|4)

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0}, {-1, 0, 1}, {0, -2, 2}}, ∞, PList → {0, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 0, 1}, {0, -1, 1}}, ∞, PList → {1, 1, 1}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 2, -1}, {0, -1, 1}}, ∞, PList → {1, 0, 1}]
```

Algebra $\mathfrak{osp}$(5|2)

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0}, {-1, 0, 1}, {0, -1, 1}}, ∞, PList → {0, 1, 1}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 0, 1}, {0, -2, 2}}, ∞, PList → {1, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 2, -1}, {0, -2, 2}}, ∞, PList → {1, 0, 0}]
```

Algebra $\mathfrak{osp}(6|2)$

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0, 0}, {-1, 2, -1, -1}, {0, -1, 2, 0}, {0, -1, 0, 2}}, ∞, PList → {1, 0, 0, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0, 0}, {-1, 0, 1, 1}, {0, -1, 2, 0}, {0, -1, 0, 2}}, ∞, PList → {1, 1, 0, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-1, 0, 1, 1}, {0, 1, 0, -2}, {0, 1, -2, 0}}, ∞, PList → {0, 1, 1, 1}]
```

Algebra $\mathfrak{osp}(4|4)$

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-1, 0, 1, 1}, {0, -1, 2, 0}, {0, -1, 0, 2}}, ∞, PList → {0, 1, 0, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0, 0}, {-1, 0, 1, 1}, {0, 1, 0, -2}, {0, 1, -2, 0}}, ∞, PList → {1, 1, 1, 1}]
```

■ Exceptional finite dimensional algebras

■ Algebra $\mathfrak{osp}(4 \mid 2; \alpha)$

(also named $\mathfrak{d}(\alpha)$)

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0}, {α, 0, -1 - α}, {0, -1, 2}}, ∞, PList → {0, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0}, {-1, 0, -α}, {0, -1, 2}}, ∞, PList → {0, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, -1 - α}, {-1, 0, -α}, {-1 - α, α, 0}}, ∞, PList → {1, 1, 1}]
```

■ Algebra $\mathfrak{ag}_2$

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 2, -3}, {0, -1, 2}}, ∞, PList → {1, 0, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0}, {-1, 0, 3}, {0, -1, 2}}, ∞, PList → {1, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, -3, 1}, {-3, 0, 2}, {-1, -2, 2}}, ∞, PList → {1, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0}, {-3, 0, 2}, {0, -1, 1}}, ∞, PList → {0, 1, 1}]
```

■ Algebra $\mathfrak{ab}_3$

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-3, 0, 1, 0}, {0, -1, 2, -2}, {0, 0, -1, 2}}, ∞, PList → {0, 1, 0, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, -3, 1, 0}, {-3, 0, 2, 0}, {1, 2, 0, -2}, {0, 0, -1, 2}}, ∞, PList → {1, 1, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-1, 2, -1, 0}, {0, -2, 0, 3}, {0, 0, -1, 2}}, ∞, PList → {0, 0, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-2, 0, 2, -1}, {0, 2, 0, -1}, {0, -1, -1, 2}}, ∞, PList → {0, 1, 1, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0, 0}, {-1, 0, 2, 0}, {0, -1, 2, -1}, {0, 0, -1, 2}}, ∞, PList → {1, 1, 0, 0}]
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-1, 2, -1, 0}, {0, -2, 2, -1}, {0, 0, -1, 0}}, ∞, PList → {0, 0, 0, 1}]
```

■ Algebra $\mathfrak{as}(4|4)$

It is implemented as a subalgebra in $\mathfrak{vect}(4|4)$ with basis

$a_{i,j} = x_i \frac{\partial}{\partial x_j} - \xi_j \frac{\partial}{\partial \xi_i}$, $i \neq j$
$a_i = a_{i,i} - a_{i+1,i+1}$
$b_{i,j} = x_i \frac{\partial}{\partial \xi_j} + x_j \frac{\partial}{\partial \xi_i}$
$c_{i,j} = \xi_i \frac{\partial}{\partial x_j} - \xi_j \frac{\partial}{\partial x_i} - \lambda\left(x_k \frac{\partial}{\partial \xi_l} - x_l \frac{\partial}{\partial \xi_k}\right)$, where $(i, j, k, l) \in A_4$
$d = \lambda \sum_{i=1}^{4} x_i \frac{\partial}{\partial x_i} + \xi_i \frac{\partial}{\partial x_i}$

See the notebook as–4_4.nb for details of implementation.

■ Infinite dimensional algebras with Cartan matrix

All computations will be made up to the degree *r*.

■ Algebra $\mathfrak{ag}_2^{(1)}$

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {4, 0, -1, 0}, {0, -1, 2, -3}, {0, 0, -1, 2}}, r, PList → {0, 1, 0, 0}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, -4, 3, 0}, {-4, 0, 1, 0}, {3, 1, 0, -3}, {0, 0, -1, 2}}, r, PList → {1, 1, 1, 0}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-1, 2, -1, 0}, {0, -3, 0, 2}, {0, 0, -1, 1}}, r, PList → {0, 0, 1, 1}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-3, 0, 3, -1}, {0, 3, 0, -2}, {0, -1, -2, 2}}, r, PList → {0, 1, 1, 0}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{1, -1, 0, 0}, {-2, 0, 3, 0}, {0, -1, 2, -1}, {0, 0, -1, 2}}, r, PList → {1, 1, 0, 0}]
```

■ Algebra $\mathfrak{osp}(4\,|\,2)^{(2)}$

```
CartanMatrixAlgebra[name, {x, h, y}, {{1, -1, 0}, {-1, 0, 1}, {0, -2, 2}}, r, PList → {1, 1, 0}]
```

■ Algebra $\mathfrak{sl}(3\,|\,3)^{(4)}$

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -2, 0}, {-1, 0, 1}, {0, -2, 2}}, r, PList → {0, 1, 0}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -2, 0}, {-1, 0, 1}, {0, -2, 2}}, r, PList → {1, 1, 1}]
```

■ Algebra $\mathfrak{svect}_{\alpha}^{L}(1\,|\,2)$

```
$SNormal = Together
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, -1}, {1 - α, 0, α}, {1 + α, -α, 0}}, r, PList → {0, 1, 1}]
```

■ Algebra $\mathfrak{psq}(3)^{(2)}$

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, -1}, {-1, 0, 1}, {1, -1, 0}}, r, PList → {1, 1, 1}]
```

■ Algebra $\mathfrak{psq}(4)^{(2)}$

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, -1}, {-1, 2, -1, 0}, {0, -1, 2, -1}, {-1, 0, 1, 0}}, r, PList → {0, 0, 0, 1}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, 1, 0, -1}, {-1, 2, -1, 0}, {0, -1, 0, 1}, {-1, 0, 1, 0}}, r, PList → {1, 0, 1, 1}]
```

■ Algebra $\mathfrak{psl}(3\,|\,3)^{(4)}$

```
CartanMatrixAlgebra[name, {x, h, y}, {{1, -1, 0}, {-1, 0, -1}, {0, 1, 1}}, r, PList → {1, 1, 1}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0}, {-1, 0, -1}, {0, 1, 2}}, r, PList → {0, 1, 0}]
```

■ Algebra $\mathfrak{sl}(2\,|\,4)^{(2)}$

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, -1, 0}, {-1, 0, 2, -1}, {-1, 2, 0, -1}, {0, -1, -1, 2}}, r, PList → {0, 1, 1, 0}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-2, 0, 1, 1}, {0, -1, 0, 2}, {0, -1, 2, 0}}, r, PList → {0, 1, 1, 1}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, -1, 0, 0}, {-2, 2, -1, 0}, {0, -1, 0, 2}, {0, 0, -1, 2}}, r, PList → {0, 0, 1, 0}]
```

■ Algebra $\mathfrak{osp}(4\,|\,2;\,\alpha)^{(1)}$

(also called $\mathfrak{d}(\alpha)^{(1)}$)

```
CartanMatrixAlgebra[name, {x, h, y}, {{2, 0, 0, -1}, {0, 2, 0, -α}, {0, 0, 2, 1 + α}, {-1, -1, -1, 0}}, r, PList → {0, 0, 0, 1}]
```

```
CartanMatrixAlgebra[name, {x, h, y}, {{0, -1, -α, 1 + α}, {-1, 0, 1 + α, -α}, {-α, 1 + α, 0, -1}, {1 + α, -α, -1, 0}}, r, PList → {1, 1, 1, 1}]
```

■ Vectorial algebras

■ Lie Superalgebra of polynomial vector fields vect(m|n)

Algebra vect(m|n)

```
VectorSpace[x, Dim → {m, n}, CoLeft → v];
Symmetric[VTimes];
VectorLieAlgebra[g, x];
```

Graded algebra vect(m|n; r)

```
ReGrade[g, r]
```

■ Lie Superalgebra of divergence–free polynomial vector fields svect(m|n)

Use `VectorLieAlgebra` and use the divergence–free elements only.

■ Algebra svect^0 (1|n)

<Not implemented yet>

■ Algebra svect$^\sim$ (0|n)

<Not implemented yet>

■ Poisson algebra po(2n|m)

The Poisson algebra is implemented for different forms of the Poisson bracket.

The vector spaces x, p, q, ζ, η, θ should be defined with `VectorSpace`[x, ...] or `TrivialSpace`[x, ...] .

The vector multiplication `VTimes` is used as supersymmetric multiplication of polynomials, so it should be declared symmetric.

Note that the odd variables are denoted here as ζ and η. The common notation ξ and η is not convenient because the variables are sorted alphabetically.

In a Poisson algebra g, the following function are defined:

`Pb`$[x, y] = \{x, y\}_{P.b.}$ is the Poisson bracket;
`pb`[x,y] is the unevaluated expression of the Poisson bracket;
`EulerOp`$_g = \sum_{i=1}^{n} x_i \frac{\partial}{\partial x_i}$ is the Euler operator E:g⟶g (here $\{x_1, \ldots x_n\}$ is the basis of the base space);
$\Delta_g = 2 -$ `EulerOp`$_g$ is a convenient operator;
`HamiltonianH`$_{g,x}[y] = -\{x, y\}_{P.b.}$ is the Hamilnonian g⟶der(g).

The function names (except Δ) may be changed using options, e.g., `PoissonAlgebra[ ..., EulerOp → Eu]`

Use `PoissonAlgebra`[*name*, *x*, `Variables` → {*v*, ...}] to define a Poisson bracket on *x* extended to polynomials in *x* and *v* (the bracket and the Euler operator will not depend on *v*).

□ po({p,q})

$\{f, g\}_{P.b.} = \sum_{i=1}^{n} (-1)^{P(f)\,P(p_i)} \left(\frac{\partial f}{\partial p_i} \frac{\partial g}{\partial q_i} - (-1)^{P(p_i)} \frac{\partial f}{\partial q_i} \frac{\partial g}{\partial p_i} \right)$ where Par(p) = Par(q):

```
dim = ...
VecorSpace[p, Dim → dim]
VecorSpace[q, Dim → dim]
Symmetric[VTimes];
PoissonAlgebra[name, {p, q}]
```

□ po({θ})

$\{f, g\}_{P.b.} = (-1)^{\mathrm{P}[f]} \sum_{i=1}^{n} \frac{\partial f}{\partial \theta_i} \frac{\partial g}{\partial \theta_i}$, where $\mathrm{P}(\theta_i) = 1$:

```
VecorSpace[θ, Dim → {0, n}]
Symmetric[VTimes];
PoissonAlgebra[name, {θ}]
```

□ po({p,ζ,...,η,q}) and po({p,ζ,...,θ,...,η,q})

This is a combination of the two forms above (where p, q, ζ, η are arbitrary superspaces, Par(p) = Par(q) and Par(ζ) = Par(η); here any number of such pairs is possible; $P(\theta_i) = 1$):

```
VectorSpace[...];
...;
Symmetric[VTimes];
PoissonAlgebra[name, {p, ξ, η, q}];
PoissonAlgebra[name, {p, ξ, θ, η, q}];
```

□ po(x)

$\{f, g\}_{P.b.} = -(-1)^{P(f)} \sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i} \frac{\partial g}{\partial x_{2n+1-i}} - (-1)^{P(x_i)} \frac{\partial f}{\partial x_{2n+1-i}} \frac{\partial g}{\partial x_i} \right)$ where $P(x_i) = P(x_{2n+1-i})$:

```
VecorSpace[x, Dim → ...]
Symmetric[VTimes];
PoissonAlgebra[name, x]
```

□ po(x;c)

Poisson bracket of the general form

$$\{f, g\}_{P.b.} = -(-1)^{P[f]} \sum_{k=1}^{m} c_k \frac{\partial f}{\partial x_{i_k}} \frac{\partial g}{\partial x_{j_k}}.$$

No checking.

```
VecorSpace[x, Dim → ...]
Symmetric[VTimes];
PoissonAlgebra[name, x, {{c1, i1, j1}, ..., {cm, im, jm}}]
```

■ Hamilton algebra $\mathfrak{h}$(2n|m)

Algebra $\mathfrak{h}$(2n|m) is implemented as the quotient $\mathfrak{po}$(2n|m)/⟨const⟩;

Grading r=0,...,m/2

Grade($v_1\ v_2\ ...\ v_k$) = gr(v_1) + gr(v_2) + ... + gr(v_k) − 2,
gr(p_i) = gr(q_i) = gr(θ_j) = 1
gr(ζ_i) = 0 for $i = 1..r$ and 1 for $i = (r+1)\,..\,n$
gr(η_i) = 2 − gr(ζ_i)

■ Contact algebra $\mathfrak{k}$(2m+1|n)

The contact bracket is defined with the help of the Poisson bracket. All forms of the Posson bracket described above, except $\mathfrak{po}$(x;c), are supported.
Here we show only the version $\mathfrak{po}$({p,q}):

```
dim = ...
VecorSpace[p, Dim → dim]
VecorSpace[q, Dim → dim]
Symmetric[VTimes];
ContactAlgebra[name, {p, q}, t]
```

No need to declare t as the trivial space; this is done by the function `ContactAlgebra`.

In a contact algebra g, the following functions are defined (in addition to `PoissonAlgebra` functions):

`Kb`$[x, y] = \{x, y\}_{K.b.} = \Delta_g[x]\,\frac{\partial y}{\partial t} - \frac{\partial x}{\partial t}\,\Delta_g[y] + \{x, y\}_{P.b.}$ is the contact bracket

`kb`[x,y] is the unevaluated expression of the contact bracket

`Contact`$K_g[x] = \Delta_g[x]\,\partial_t$ − `Hamiltonian`$H_g[x] + \frac{\partial x}{\partial t}$ `EulerOp`$_g$ is the contact operator `Contact`$K_g : g \to \mathfrak{der}(g)$

The function names may be changed using options, e.g., `ContactAlgebra`[..., `Kb`→`Cb`]

The option `Variables` −> {v, ...} extends the algebra to polynomials in v (the bracket does not depend on v).

■ Buttin algebra $\mathfrak{b}(n)$

The function `ButtinAlgebra` defines the Buttin antibracket on polynomial in x_1, ..., x_n, y_1, ..., y_n, where $P(y_i) = 1 - P(x_i)$:

```
VecorSpace[x, ...]
VecorSpace[x, ...]
Symmetric[VTimes];
ButtinAlgebra[name, {x, y}, t]
```

On the Buttin algebra b, the following functions are defined:

`Bb`$[f, g] = \{f, g\}_{B.b.} = (-1)^{P(f)\,P(x_i)}\,\frac{\partial f}{\partial x_i}\,\frac{\partial g}{\partial y_i} + (-1)^{P(f)\,P(y_i)}\,\frac{\partial f}{\partial y_i}\,\frac{\partial g}{\partial x_i}$ is the Buttin bracket;

`bb`[x,y] is the unevaluated expression of Poisson bracket;

`EulerOp`$_b = \sum_{i=1}^{n} x_i\,\frac{\partial}{\partial x_i} + y_i\,\frac{\partial}{\partial y_i}$ $\qquad E : b \longrightarrow b;$

$\Delta_b = 2 -$ `EulerOp`$_b$

$\mathtt{Br}[f, g] = \{f, g\}_{B.b.} = (-1)^{P(f)\,P(x_i)} \frac{\partial f}{\partial x_i} \frac{\partial g}{\partial y_i} + (-1)^{P(f)\,P(y_i)} \frac{\partial f}{\partial y_i} \frac{\partial g}{\partial x_i}$

$\mathtt{EulerOp}_b = \sum_{i=1}^{n} x_i \frac{\partial}{\partial x_i} + y_i \frac{\partial}{\partial y_i}$ is the Euler operator $E : b \longrightarrow b$;

$\Delta_b = 2 - \mathtt{EulerOp}_b$.

The function names (except Δ) may be changed using options, e.g., `ButtinAlgebra`[..., `EulerOp`→Eu].

The option `Variables` –> {*v*, ...} extends the algebra to polynomials in *v* (the bracket does not depend on *v*).

Grading r=0,...,n (grading $r = n - 1$ is not the Weisfeiler one)

Grade($v_1\ v_2\ ...\ v_k$) = gr(v_1) + gr(v_2) + ... + gr(v_k) – 2,
gr(x_i) = 1 for $i = 1..r$ and 2 for $i = (r + 1) .. n$
gr(y_i) = 2 – gr(x_i)

■ "Odd" contact algebra $\mathfrak{m}(n)$

The function `OKAlgebra` defines the Buttin antibracket and the "odd contact bracket" on polynomials in $x_1, \ ... x_n, \ y_1, \ ... y_n, \tau$, where $P(y_i) = 1 - P(x_i)$ and $P(\tau) = 1$:

```
VecorSpace[x, ...]
VecorSpace[x, ...]
Symmetric[VTimes];
OKAlgebra[name, {x, y, τ}]
```

No need to declare τ as an odd trivial space; this is done by the function OKAlgebra.

On the odd contact algebra m, the following functions are defined (in addition to the functions in the Buttin algebra):

Ob[f, g] = $\{f, g\}_{O.b.} = \Delta_b[f] \frac{\partial g}{\partial t} + (-1)^{P(f)} \frac{\partial f}{\partial t} \Delta_b[g] - \{f, g\}_{B.b.}$ is the odd contact bracket;

ob[x,y] is the unevaluated expression of the odd contact bracket;

The names of the functions may be changed using options, e.g., `OKAlgebra`[..., Ob→Ok].

The option `Variables` –> {*v*, ...} extends the algebra to polynomials in *v* (the bracket does not depend on *v*).

Grading r=0,...,n (grading $r = n - 1$ is not the Weisfeiler one)

Grade($v_1\ v_2\ ...\ v_k$) = gr(v_1) + gr(v_2) + ... + gr(v_k) – 2,
gr(x_i) = 1 for $i = 1..r$ and 2 for $i = (r + 1) .. n$
gr(y_i) = 2 – gr(x_i)
gr(τ) = 2

■ The deformation of the Buttin algebra $\mathfrak{b}_\lambda(n)$

The algebra $\mathfrak{b}_\lambda(n)$ is implemented as a subalgebra in $\mathfrak{m}(n)$, namely, as

$$\mathfrak{b}_\lambda(n) = \left\{ f \in \mathfrak{m}(n) : (b\,n - a\,\mathtt{EulerOp}_{\mathfrak{m}})\,\frac{\partial f}{\partial \tau} = a \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i\,\partial y_i} \right\},\ \text{where } \lambda = \frac{2\,a}{n(a-b)}$$

Special cases to be treated with extra care:
λ=0: $\mathfrak{b}_0(n) = \mathfrak{b}(n)$ is not simple; $\mathfrak{b}(n)/\langle \text{const} \rangle = \mathfrak{le}(n)$;

λ=1: $\mathfrak{b}_1(n)$ contains an ideal $\mathfrak{b}_1^0(n)$ of codimension 1;

λ=∞: $\mathfrak{b}_\infty(n)$ contains an ideal $\mathfrak{b}_\infty^0(n)$ of codimension 1;

$\lambda = \frac{2}{n-1}$: $\mathfrak{b}_\lambda(n)$ preserves the volume; $\mathfrak{b}_\lambda(n) = \mathfrak{sm}(n)$;

$\lambda = \frac{1}{n};\ \frac{2}{n}$.

For n=2, additionally, $\lambda = -\frac{1}{2}$ or $-\frac{3}{2}$: at these points there are extra deformations.

Grading r=0,...,n (grading $r = n - 1$ is not the Weisfeiler one)

Grade$(v_1\ v_2\ ...\ v_k)$ = gr(v_1) + gr(v_2) + ... + gr(v_k) − 2,
gr$(x_i) = 1$ for $i = 1..r$ and 2 for $i = (r+1)\,..\,n$
gr(y_i) = 2 − gr(x_i)
gr$(\tau) = 2$

Grading E (for $\mathfrak{b}_\lambda(2)$ only)

Grade$(v_1\ v_2\ ...\ v_k)$ = Grade(v_1) + Grade(v_2) + ... + Grade(v_k),
Grade$(x_i) = 1$
Grade$(y_i) = -1$
Grade$(\tau) = 0$

See the notebook b_lambda−n.nb for implementation details.

■ Leites algebras $\mathfrak{le}(n)$, $\mathfrak{sle}(n)$, $\mathfrak{sle}^o(n)$

Algebra $\mathfrak{le}(n)$ is implemented as the quotient $\mathfrak{b}(n)/\langle \text{const} \rangle$;
Algebra $\mathfrak{sle}(n) \subset \mathfrak{le}(n)$ is determines as $\mathfrak{sle}(n) = \left\{ f \in \mathfrak{le}(n) \,\middle|\, \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i\,\partial y_i} = 0 \right\}$;
Algebra $\mathfrak{sle}^0(n) \subset \mathfrak{sle}(n)$ is an ideal, the codimension 1 complement to $\langle y_1\ ...\ y_n \rangle$.

Grading r=0,...,n (grading $r = n - 1$ is not Weisfeiler one)

Grade$(v_1\ v_2\ ...\ v_k)$ = gr(v_1) + gr(v_2) + ... + gr(v_k) − 2,
gr$(x_i) = 2$ for $i = 1..r$ and 1 for $i = (r+1)\,..\,n$;
gr(y_i) = 2 − gr(x_i).

See the notebook le.nb for the details of implementation.

■ Exceptional algebras

□ Algebra $\mathfrak{kas}$

The algebra is implemented as a subalgebra in $\mathfrak{k}(1|6)$:

```
VectorSpace[ξ, Dim → {0, 3}];
VectorSpace[η, Dim → {0, 3}];
Symmetric[VTimes];
ContactAlgebra[k16, {ξ, η}, t]
```

The basis of subalgebra $\mathfrak{kas}$ is:

$$t^n - n\,(n-1)\,(n-2)\,t^{n-3}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3$$

$$t^n\,\xi_i + n\,(n-1)\,t^{n-2}\,\frac{\partial}{\partial \eta_i}\,(\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3)$$

$$t^n\,\eta_i + n\,(n-1)\,t^{n-2}\,\frac{\partial}{\partial \xi_i}\,(\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3)$$

$$t^n\,\xi_i\,\xi_j + n\,\frac{\partial}{\partial \eta_i}\,\frac{\partial}{\partial \eta_j}\,(\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3)$$

$$t^n\,\xi_i\,\eta_j + n\,\frac{\partial}{\partial \eta_i}\,\frac{\partial}{\partial \xi_j}\,(\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3)$$

$$t^n\,\eta_i\,\eta_j + n\,\frac{\partial}{\partial \xi_i}\,\frac{\partial}{\partial \xi_j}\,(\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3)$$

$$t^n\,\xi_1\,\xi_2\,\xi_3$$

$$t^n\,(\xi_1\,\xi_3\,\eta_1 + \xi_2\,\xi_3\,\eta_2),$$

$$t^n\,(-\xi_1\,\xi_2\,\eta_1 + \xi_2\,\xi_3\,\eta_3)$$

$$t^n\,(-\xi_1\,\xi_2\,\eta_2 - \xi_1\,\xi_3\,\eta_3)$$

$$t^n\,\xi_3\,\eta_1\,\eta_2$$

$$t^n\,(-\xi_2\,\eta_1\,\eta_2 + \xi_3\,\eta_1\,\eta_3)$$

$$t^n\,(\xi_1\,\eta_1\,\eta_2 + \xi_3\,\eta_2\,\eta_3),$$

$$t^n\,\xi_2\,\eta_1\,\eta_3$$

$$t^n\,(\xi_1\,\eta_1\,\eta_3 - \xi_2\,\eta_2\,\eta_3)$$

$$t^n\,\xi_1\,\eta_2\,\eta_3$$

See the notebook kas.nb for the details of implementation.

□ Algebra $\mathfrak{hsle}$(5|10)

First implementation: in $\mathfrak{svect}(x_1, \ldots x_5) \oplus \Pi d\Omega^1(x_1, \ldots, x_5)$:

$[\Pi\omega_1, \Pi\omega_2] = \omega_1\ \omega_2\ \mathrm{dx}_1{}^{-1}\ \mathrm{dx}_2{}^{-1}\ \mathrm{dx}_3{}^{-1}\ \mathrm{dx}_4{}^{-1}\ \mathrm{dx}_5{}^{-1}$ where $\mathrm{dx}_i{}^{-1} = \frac{\partial}{\partial x_i}$
$[\xi_1, \xi_2]$ is Lie bracket on $\mathfrak{vect}$(x)
$[\xi, \Pi\omega] = \Pi\, L_\xi\, \omega$

Standard grading $\mathfrak{e}$(5|10)

Grade$[x_i] = 2$, Grade$[\mathrm{dx}_i] = -\frac{1}{2}$ (Grade$\left[\frac{\partial}{\partial x_i}\right] = -$Grade$[\mathrm{dx}_i]$ and Grade$[\Pi] = 0$ for all gradings)

Regrading $\mathfrak{e}$(9|6)

Grade$[x_i] = 1$, Grade$[\mathrm{dx}_i] = -\frac{1}{2}$ $(i \le 4)$
Grade$[x_5] = 2$, Grade$[\mathrm{dx}_5] = \frac{1}{2}$

Regrading $\mathfrak{e}$(11|9)

Grade$[x_i] = 2$, Grade$[\mathrm{dx}_i] = 0$ $(i \le 3)$
Grade$[x_i] = 1$, Grade$[\mathrm{dx}_i] = -1$ $(i \ge 4)$

Regrading $\mathfrak{ck}(9|11)$

$$\text{Grade}[x_i] = 3, \text{Grade}[\text{dx}_i] = 0 \ (i \le 2)$$
$$\text{Grade}[x_i] = 2, \text{Grade}[\text{dx}_i] = -1 \ (i \ge 3)$$

Second implementation: as a submodule in $\mathfrak{svect}(5|10)$ genearated as the Cartan prolongation of the pair $(g_{-1},\ g_0)$, where

$g_{-1} = \langle n_{i,j},\ i \le i < j \le 5\rangle$, $n_{i,j} = \frac{\partial}{\partial\theta_{i,j}} + \sum \theta_{k,l}\ \frac{\partial}{\partial x_m}$, the sum over (k, l, m) such as $(i, j, k, l, m) \in A_5$;

$g_0 = \langle z_{i,j},\ 1 \le i \ne j \le 5 \bigcup z_{i,i} - z_{i+1,i+1},\ 1 \le i \le 4\rangle$, where $z_{i,j} = x_i\ \frac{\partial}{\partial x_j} + \sum_{k=1}^{5} \theta_{j,k}\ \frac{\partial}{\partial\theta_{k,i}}$.

See the notebook ksle−5_10.nb for the details of implementation.

Regrading $\mathfrak{e}(11|9)$:

$$\text{Grade}[x_i] = \begin{cases} 2, & i \le 3 \\ 1, & i \ge 4 \end{cases}$$
$$\text{Grade}[\theta_{i,j}] = 4 - \text{Grade}[x_i] - \text{Grade}[x_j]$$

Regrading $\mathfrak{ck}(11|9)$

$$\text{Grade}[x_i] = \begin{cases} 2, & i \le 2 \\ 1, & i \ge 3 \end{cases}$$

□ Algebra $\mathfrak{mb}(4|5)$

This algebra is implemented as a subalgebra in the odd−contact algebra $\mathfrak{m}(4|5)$ generated as a generalized Cartan prolongation of $(g_{-2},\ g_{-1}, g_0)$,where $g_{-2} = \langle 1\rangle$ and $g_{-1} = \langle q_0,\ \ldots q_3, \xi_0,\ \ldots \xi_3\rangle$ coincide with the corresponding components in $\mathfrak{m}(4|5)$, and where for the basis of g_0 we take

$-q_0\ q_i + \xi_j\ \xi_k$ for all $(i,\ j,\ k) \in A_3$
$q_i\ \xi_j$ for all $1 \le i \le 3, 0 \le j \le 3$
$q_0\ \xi_0 - q_i\ \xi_i, 1 \le i \le 3$
$q_i\ q_j$ for all $1 \le i \le j \le 3$
$\xi_i\ \xi_j$ for all $1 \le i < j \le 3$

Standard grading:

$$\text{Grade}[q_i] = \text{Grade}[\xi_i] = 1,$$
$$\text{Grade}[\tau] \text{ ^= } 2$$

Regrading K

$$\text{Grade}[q_0] = 0$$
$$\text{Grade}[\xi_0] = 3$$
$$\text{Grade}[q_i] = 2, i > 0$$
$$\text{Grade}[\xi_i] = 1, i > 0$$
$$\text{Grade}[\tau] \text{ ^= } 3$$

See the notebook mb−4_5.nb for the details of implementation.

□ Algebra vas(4|4)

This algebra is implemented as subalgebra in $\mathfrak{vect}(4|4)$. The subalgebra is singled out by the following equations on the vector field $\xi = \sum_{i=1}^{4} f_i \frac{\partial}{\partial p_i} + g_i \frac{\partial}{\partial \theta_i}$:

(1) $\frac{\partial f_i}{\partial p_j} + (-1)^{P(\xi)} \frac{\partial g_j}{\partial \theta_i} = 0,\ 1 \le i \ne j \le 4;$
(1a) $\frac{\partial f_i}{\partial p_i} + (-1)^{P(\xi)} \frac{\partial g_i}{\partial \theta_i} = \frac{1}{2} \sum_{j=1}^{4} \frac{\partial f_j}{\partial p_j},\ 1 \le i \le 4;$
(2) $\frac{\partial f_i}{\partial \theta_j} + \frac{\partial f_j}{\partial \theta_i} = 0,\ 1 \le i,\ j \le 4;$
(3) $\frac{\partial g_i}{\partial p_j} - \frac{\partial g_j}{\partial p_i} - (-1)^{P(\xi)}\ \lambda\ \frac{\partial f_k}{\partial \theta_l} + (-1)^{P(\xi)}\ \lambda\ \frac{\partial f_l}{\partial \theta_k} = 0$ for $(i, j, k, l) \in A_4$ and $\lambda \ne 0$

Standard grading:

$$\text{Grade}[p_i] = \text{Grade}[\theta_i] = 1$$

See the notebook vas−4_4.nb for the details of implementation.

□ Algebra vle(4|3)

This algebra is implemented as a subalgebra in $\mathfrak{vect}(4|3)$. The subalgebra is singled out by the following equations on the vector field $\xi = \sum_{i=0}^{3} f_i \frac{\partial}{\partial p_i} + \sum_{i=1}^{3} g_i \frac{\partial}{\partial \theta_i}$:

(1) $\frac{\partial f_i}{\partial p_j} + (-1)^{P(\xi)} \frac{\partial g_j}{\partial \theta_i} = 0$ for $1 \le i \ne j \le 3;$
(1a) $\frac{\partial f_i}{\partial p_i} + (-1)^{P(\xi)} \frac{\partial g_i}{\partial \theta_i} = \frac{1}{2} \sum_{j=0}^{3} \frac{\partial f_j}{\partial p_j},\ 1 \le i \le 3;$
(2) $\frac{\partial f_i}{\partial \theta_j} + \frac{\partial f_j}{\partial \theta_i} = 0$ for $1 \le i,\ j \le 4;$
(3) $\frac{\partial g_i}{\partial p_j} - \frac{\partial g_j}{\partial p_i} + (-1)^{P(\xi)} \frac{\partial f_0}{\partial \theta_k} = 0,\ (i, j, k) \in A_3;$
(4) $\frac{\partial f_i}{\partial p_0} = 0$ for $1 \le i \le 3;$
(5) $\frac{\partial f_i}{\partial \theta_j} - \frac{\partial f_j}{\partial \theta_i} - (-1)^{P(\xi)}\ 2\ \frac{\partial g_k}{\partial p_0} = 0$ for $(i, j, k) \in A_3$.

Standard grading:

$$\text{Grade}[p_i] = \text{Grade}[\theta_i] = 1$$

Regrading r=1

$$\text{Grade}[p_0] = 0,$$
$$\text{Grade}[p_i] = 2 \quad \text{for } i > 0,$$
$$\text{Grade}[\theta_i] = 1.$$

Regrading r=2

$$\text{Grade}[p_0] = 0,$$
$$\text{Grade}[p_1] = 2,$$
$$\text{Grade}[\theta_1] = 0,$$
$$\text{Grade}[p_i] = \text{Grade}[\theta_i] = 1 \text{ for } i > 1.$$

See the notebook vle–4_3.nb for the details of implementation.

■ Stringy algebras

■ Moebius-Poisson Algebra

The Moebius-Posson bracket is defined in terms of the Poisson bracket. All forms of Poisson bracket described above, except for po(x;c), are supported.

Here we show only the version po({p,q}).

The function MoebiusAlgebra defines the Poisson bracket and the Moebius-Posson bracket on the space of polynomials in p_i, q_i, θ, t, t^{-1}:

```
dim = . . .;
VecorSpace[p, Dim → dim]
VecorSpace[q, Dim → dim]
Symmetric[VTimes];
MoebiusAlgebra[name, {{p, q}, θ, t}]
```

No need to declare θ and t as an trivial spaces; this is done by the function MoebiusAlgebra.

On the Moebius-Posson algebra m, the following function are defined (in addition to the functions in the Poisson algebra; the functions EulerOp and Δ are redefined):

Mb$[f, g] = \{f, g\}_{M.b.} = \{f, g\}_{P.b.} + (-1)^{P(f)} \frac{1}{t} \frac{\partial f}{\partial \theta} \frac{\partial g}{\partial \theta}$ is the Moebius-Posson bracket;

mb$[f, g]$ is the unevaluated expression of Moebius-Posson bracket;

$\text{EulerOp}_m = \text{EulerOp}_{\text{po}} + \theta \frac{\partial}{\partial \theta}$ is the Euler operator $E : m \longrightarrow m$;
$\Delta_m = 2 - \text{EulerOp}_m$.

The names of the functions (except Δ) may be changed using options, e.g., MoebiusAlgebra[..., EulerOp→Eu].

The option Variables –> {v, ...} extends the algebra to polynomials in v (the bracket does not depend on v).

■ Moebius Contact (Ramond) algebra $\mathfrak{k}^M(1 \mid n)$

The Ramond bracket is defined with the help of the Moebius-Poisson bracket which, in turn, is defined with the help of the Poisson bracket. All forms of the Poisson bracket described above, except po(x;c), are supported. Here we show only the version po({p,q}).

The function RamondAlgebra defines the Poisson bracket, the Moebius-Poisson bracket, and the Ramond bracket on the space of polynomial in p_i, q_n, θ, t, t^{-1}, where $P(p_i) = P(q_i)$, $P(\theta) = 1$ and $P(t) = 0$:

```
dim = . . .;
VecorSpace[p, Dim → dim]
VecorSpace[q, Dim → dim]
Symmetric[VTimes];
RamondAlgebra[name, {{p, q}, θ, t}]
```

No need to declare θ and t as trivial spaces; this is done by the function `RamondAlgebra`.

On the Ramond algebra, r, the following functions are defined (in addition to the functions on Poisson and Moebius-Poisson algebras):

`Rb`$[f, g] = \{f, g\}_{O.b.} = \Delta_r[f]\,\mathcal{D}(g) - \mathcal{D}(f)\,\Delta_r[g] - \{f, g\}_{M.b.}$ is the Ramond bracket, where $\mathcal{D} = \frac{\partial}{\partial t} - \frac{\theta}{2\,t}\,\frac{\partial}{\partial \theta}$;

`rb`[x,y] is the unevaluated expression of the Ramond bracket;

`HamiltonianH`$_r[f]$ = `HamiltonianH`$_{\text{po}}[f] - (-1)^{P(f)}\,\frac{1}{t}\,\frac{\partial f}{\partial \theta}\,\frac{\partial}{\partial \theta}$
`RamondK`$_r[f] = \Delta_r[f]\,\mathcal{D}$ − `HamiltonianH`$_r[f] + \mathcal{D}(f)$ `EulerOp`$_r$ is the operator R: r⟶der(r).

The function names `Rb` and `rb` may be changed using options, e.g., `RamondAlgebra`[..., `Rb`→ Bk].

■ Other algebras

■ Algebra $\mathfrak{gl}(\lambda)$

The algebra is implemented as the subalgebra in $\mathfrak{diff}(K^1)$ generated by

$$x^+ = u^2\,\frac{\partial}{\partial u} - (\lambda - 1)\,u, \quad z = \frac{\partial^2}{\partial u^2}, \quad x^- = -\frac{\partial}{\partial u}$$

See the notebook gll.nb for the details of implementation.

■ Engel Algebra

In the notebook Engel.nb, there are two implementations of the Engel algebra: as a subalgebra of $\mathfrak{vect}(K^1)$ and manually by giving the basis and the multiplication table.

Package SuperLie`Cohom`: Calculation of cohomology

This package helps to calculate cohomology of Lie [super]algebas with coefficients in any module (splitted into a sum of finite−dimensional subspaces).

The main algebra $\mathfrak{a}$ should be defined in `Tabular` mode, i.e., with basis $\mathfrak{a}_1$,... $\mathfrak{a}_n$ and the bracket defined via multiplication table. If it is not the case, one should first define $\mathfrak{a}$ as `SubAlgebra` of the original algebra.

If another algebra $\mathfrak{g}_0$ acts on both the main algebra $\mathfrak{a}$ and on the module of coefficients $\mathfrak{m}$ so that $\mathfrak{a} + \mathfrak{g}_0$ acts on $\mathfrak{m}$ and $d(g(f)) == g(d(f))$, the action of $\mathfrak{g}_0$ may be used in calculations. For example, it suffices to calculate the highest vectors in cohomologies (with respect to an even semi-simple subalgebra of $\mathfrak{g}_0$).

See example below.

■ Reference

■ Variables

■ Input data

ch$raise

■ ch$raise should hold the list of rising vectors in even semisimple part of $\mathfrak{g}_0$.

ch$lower

■ ch$lower should hold the list of lowering vectors in even semisimple part of $\mathfrak{g}_0$.

ch$gen

■ ch$gen holds the list of generators of algebra $\mathfrak{g}_0$. This list is used to speed up the calculation of different $\mathfrak{g}_0$ −modules by skipping calculating the full action table. Set ch$gen=All to calculate full action table (this is the default value).

ch$basis

■ The value of ch$basis if the name of the function *fn*[*r*,*d*] that returns the list of all *r*−forms of degree *d*. The default name is chBasis.

ch$Split

■ The value of ch$Split is the name of the function that should be passed to `SplitList` or `SplitSum` when splitting expressions to homogenous parts. The default name is chSplit.

■ The values of this function are used for indexing the results of calculations. They are referred to as *selectors* in this documents.

ch$Wt

■ The value of ch$Wt is a (possibly user−defined) weight function relative to a fixed Cartan subalgebra of $\mathfrak{g}_0$.

■ The default value is `Weight`.

ch$Out

■ The value of ch$Out is a function that is applied to cohomology when printing. The default value is `Identity`.

■ Storage

These symbols are used to store the information required for calculation. Normally their values are assigned automatically, but the user can change them if required.

ch$alg

■ ch$alg holds the name of the algebra $\mathfrak{a}$ those cohomoligy are calculated.

ch$d

■ ch$d holds the name of the module `DLeft`[$\mathfrak{a}$] of differential 1−forms on the main algebra $\mathfrak{a}$ with trivial coefficients.

ch$g0

■ ch$g0 holds the name of the algebra $\mathfrak{g}_0$ acting on the main algebra $\mathfrak{a}$. This action is used to simplify the calculations: it suffices to calculate only cohomology of the highest weight with respect to the even semisimple subalgebra of $\mathfrak{g}_0$.

■ See also **ch$raise** and **chSplit**.

ch$M

■ ch$M denotes the module of differential forms.

ch$ex

■ ch$ex denotes the $\mathfrak{g}$−module of exact forms (the last component calculated by **chExMod**).

■ Results

The results of calculations are stored as values of `ch$tab`[...], `ch$res`[...], and `ch$book`[...].

To avoid reusing the same storage for solutions of different problems, bind the variables chtab, chres, and/or ch$book to different symbols, as follow:

```
ch$book = prob1answer;
(*solve problem 1*)
ch$book = prob2answer;
(*solve problem 2*)
(*compare the results*)
```

ch$tab

■ `ch$tab`[*degree*]={*selector*→{{`Dim`[ker0],`Dim`[im0]},...},...} stores the dimensions of the spaces stores in `ch$res`.

ch$res

■ `ch$res`[*degree*, *rank*]={*selector*→{*ker*,*im*}, ...} stores the results of `chCoHom`[*s*, *r*].

ch$book

- The final results are stored as `ch$book`[*degree*, *rank*,*selector*]=cohomology or a list of cohomologies.
- To see all stored results, invoke `DownValues[ch$book]`.

■ Functions

chSetAlg

- chSetAlg[$\mathfrak{a}$, $\mathfrak{d}$] defines $\mathfrak{d}$ as **DLeft**[$\mathfrak{a}$] and stores $\mathfrak{a}$ as algebra for calculation of cohomology.
- chSetAlg[$\mathfrak{a}$, $\mathfrak{d}$, $\mathfrak{g}_0$] specifies also the algebra $\mathfrak{g}_0$ that acts on $\mathfrak{a}$.
- Algebras $\mathfrak{a}$ and $\mathfrak{g}_0$ should be already defined as well as the action of $\mathfrak{g}_0$ on $\mathfrak{a}$.

chScalars

- chScalars[*b*,*c*] declares the names of scalar coefficients to be used in calculations.

chHVect

- chHVect[*f*] calculates the highest vectors in the space given as the `general sum` *f*. The result is also a general sum.

chLVect

- chLVect[*f*] calculates the lowest vectors in the space given as the `general sum` *f*. The result is also a general sum.

chGenDim

- chGenDim[*f*] returns the number of indeterminate coefficients in the `general sum` *f*.

chCoHom

- chCoHom[*s*, *r*] calculates the component of grade *s* of the kernel and the image of the operator $d:\Omega^r \longrightarrow \Omega^{r+1}$ .The result is stored in ch$res[*s*, *r*] as list of two `general sums`.
- This function is called from **chCalc**.

chRes

- `chRes`[*s*,*r*] prints the results of calculations of (<=*r*)–cohomologies of grade *s*.

chEqu

- `chEqu`[*s*, *r*, *w*] solves equations of exactness for the coefficients of a generic closed *r*–form of grade *s* and weight *w*.

chCalc

- `chCalc`[*s*,*r*] calculates (<=*r*)–cohomologies of grade *s* and prints a short summary.
- `chCalc`[*s*,{*p*,*q*}] calculates *r*–cohomologies ($p \le r \le q$) of grade *s* and prints a short summary.
- The summary contains the total number of independent highest vectors in the kernel and image of the exterior derivatives.
- Use **chNext[]** to enumerate the calculated cohomologies.

chNext

- `chNext`[] shows the next calculated cohomology of the current degree. If the cohomology is pure, the result is stored (using **chBook**[]) and returned. Otherwise the function returns the conditions of exactness (see **chEqu**) and the rates of coefficients (see **chRate**) for the next cohomology. To store the results in this case, the user should manually invoke **chBook** with appropriate arguments. Even for pure cohomologies, one can call **chBook** to overwrite the results stored automatically (e.g., `chBook`[*c*[1]→2] will store the cohomology in different scale).
- `chNext`[*degree*] begins enumerating the cohomologies of the given degree (they should be already calculated using **chCalc**).

- `chNext[]` shows the next calculated cohomology of the current degree. If the cohomology is pure, the result is stored (using **`chBook`**`[]`) and returned. Otherwise the function returns the conditions of exactness (see **`chEqu`**) and the rates of coefficients (see **`chRate`**) for the next cohomology. To store the results in this case, the user should manually invoke **`chBook`** with appropriate arguments. Even for pure cohomologies, one can call **`chBook`** to overwrite the results stored automatically (e.g., `chBook[`*c*`[1]→2]` will store the cohomology in different scale).
- `chNext[`*degree*`]` begins enumerating the cohomologies of the given degree (they should be already calculated using **`chCalc`**).

chBook

- `chBook[`*rule*`]` takes the last calculated cohomology (pointed by **`chNext`**), replaces the indeterminate scalar coefficients (first, using the given replacement rule (it may be also a list of rules) and then the remaining ones with 0), and stores the result in **`ch$book`**`[`*grade*, *rank*, *selector*`]`.
- `chBook[`*rule1*,*rule2*,...`]` stores a list of cohomologies, one for each argument. The number of rules should be equal to the multiplicity of the cohomology.
- `chBook[]` stores a cohomology for every indeterminate scalar coefficient. This form should be used for pure cohomology only.

chPos

- `chPos[]` takes the current position in the enumeration of cohomologies, i.e., the list {*grade*, *rank*, *selector*}. This position is advanced by function chNext[]

chExMod

- `chExMod[`*s*,*r*`]` builds the $\mathfrak{g}_0$ −module of exact *r*−forms of grade *s* using the default list **`ch$gen`** of generators of $\mathfrak{g}_0$.
- `chExMod[`*s*,*r*,*gen*`]` uses the given list of generators of $\mathfrak{g}_0$.
- `chExMod[`*s*,*r*,`All]` builds the module of exact forms and the full table of the $\mathfrak{g}_0$ −action on this module. This is slower and should be used only if the $\mathfrak{g}_0$ −action is required for calculations.
- The name of the module of exact forms is `ch$ex`.

chMod

- `chMod[`*m*,*v*`]` builds the $\mathfrak{g}_0$ −module *m* generated by the form (or list of forms) *v* and returns the dimension of the module modulo the module of exact forms, see **`chDim`**. The module of exact forms should be already calculated for the required grade and "arity", see **`chExMod`**. To simplify the calculations, the default list **`ch$gen`** of generators of $\mathfrak{g}_0$ is used.
- `chMod[`*m*,*v*,*gen*`]` uses the given list of generators of $\mathfrak{g}_0$.
- `chMod[`*m*,*v*,`All]` builds the module generated by *v* and the full table of the $\mathfrak{g}_0$ −action on this module. This is slower and should be used only if the $\mathfrak{g}_0$ −action is required for calculations.

chDim

- `chDim[`$\mathfrak{m}$, ...`]` calculates the dimension of the [sum of] the given $\mathfrak{g}_0$ −module(s) of forms modulo the module of the exact forms.
- The module of exact forms should be already calculated for the required grade and "arity", see **`chExMod`**.
- This function is used to determine the $\mathfrak{g}_0$ −structure on the space of cohomology in case where $\mathfrak{g}_0$ is not a semisimple Lie algebra.

chInMod

- chInMod[$\mathfrak{m}$, *v*] checks if the vector *v* is inside module $\mathfrak{m}$ (modulo the module of exact forms).

chInSol

- chInSol[$\mathfrak{m}$, *v*] solves condition $v \in \mathfrak{m}$ (modulo the module of exact forms).

chRate

- chRate[v] returns the table of ratings for indeterminate coefficients in general sum v. The *rating* is the number of occurrences of the coefficients in the expression. The ratings may help to find a relatively simple expression for basis vectors of a given mixed cohomology.

■ User–defined functions

chBasis

- The function chBasis[*r*,*s*] should return the list of all *r*–forms of degree *s*.
- The default definition builds this list from the list of 0–forms, i.e., elements of the module of coefficients.
- The user should either redefine chBasis[r,s], or define chBasis[*s*] that should return the list of 0–forms of degree *s*.

chSplit

- chSplit it the user–defined function that is passed to SplitList and SplitSum when splitting expressions into homogenous parts.
- The values of this function are used for indexing the results of calculations. They are referred to as *selectors* in this documents.

■ Example: Cohomology related with $\mathfrak{sl}(2\,|\,3)$

Let $\mathfrak{g} = \mathfrak{sl}(2\,|\,3)$. Calculate the cohomology of the lower odd block of $\mathfrak{g}$ (in some format) with coefficients in $\mathfrak{g}$.

■ First, load packages

```
Needs["SuperLie`"]
Needs["SuperLie`Cohom`"]
```

```
SuperLie Package Version 2.01 installed
Disclaimer: This software is provided "AS IS", without a warranty of any kind
```

■ Set operation properties

For our task, we need automatic expansions of
(1) linear expressions in tensor products (i.e., x**(a+b)→x**a+x**b) and
(2) action of any algebra on tensor products and exterior products:

```
Linear[Tp];
Jacobi[Act -> Tp];
Jacobi[Act -> wedge];
```

■ Define algebras

Next, build the algebra $\mathfrak{g}$ and subalgebras $y = \mathfrak{g}_-$ and $h = \mathfrak{g}_0 = \mathfrak{sl}(2) \oplus \mathfrak{sl}(3) \oplus \mathfrak{c}$.

The subalgebra $\mathfrak{g}_0$ acts on $\mathfrak{g}$ and $\mathfrak{g}_-$; this action commutes with the exterior derivative, so the cohomologies are g_0–modules and it suffices to find the highest weight vectors.

We should also redefine the grading on $\mathfrak{g}$ so that the subalgebra $\mathfrak{g}_0$ has grade 0.

```
slAlgebra[g, Dim → {2, 3}];
Grade[g[i_, j_]] ^:= If[i ≤ 2, -1, 0] + If[j ≤ 2, 1, 0];
SubAlgebra[y, g, Select[Basis[g], Grade[#] < 0 &]];
SubAlgebra[h, g, Select[Basis[g], Grade[#] == 0 &]];
RestrictModule[y, h];
```

The function `RestrictModule` defines action of *h* on *y*.

■ Set the task

Now we explain our task to **Cohom** package.
The forms on *y* will be denoted as *dy*.
The function `chBasis` should build the basis of coefficients of the given degree.
The weight in $\mathfrak{g}$ is defined as a $\mathfrak{gl}$–weight with 5 weight marks. A weight *w* may be $\mathfrak{g}_0$ –highest only if $w_1 \geq w_2$ and $w_3 \geq w_4 \geq w_5$. Therefore we ignore all other weights (value `SkipVal` in `chSplit` function).

```
chSetAlg[y, dy, h];
chScalars[b, c];
ch$Wt = Weight;
chBasis[d_] := Select[Basis[g], Grade[#] == d &];
chSplit[x_] := With[{w = Weight[x]},
  If[w[[1]] ≥ w[[2]] ∧ w[[3]] ≥ w[[4]] ≥ w[[5]], w, SkipVal]];
```

To find the highest vectors, we need the list of raising operator in h. These are elements whose images in g are $g_{1,2}$, $g_{3,4}$, $g_{4,5}$:

```
pos = Position[Image[h], g[i_, j_] /; j == i + 1]
```

```
{{5}, {6}, {7}}
```

```
ch$raise = {h[5], h[6], h[7]}
```

```
{h5, h6, h7}
```

■ Output format

For the output, replace *dy*[*i*] with *d*[*g*[..]]:

```
ch$Out = OutMap;
OutMap[v_] := v /. dy[i_] :→ d[Image[y][[i]]]
```

Now we are ready to calculate cohomology.
All the *dy*'s have degree 1, the degrees of coefficients range from −1 to 1, so the degrees of *r*–forms are from *r*−1 to r+1.

■ Calculations

Degree −1: only 0–cohomologies are possible:

```
chCalc[-1, 0]
```

```
Total: {{1, 0}}
```

```
chNext[]
```

```
{1, 0, 0, 0, -1} → g1,5
```

```
chNext[]
```

```
No more cohomologies
```

Degree 0: 0– and 1–cohomologies:

```
chCalc[0, 1]
```

```
Total: {{0, 3}, {4, 0}}
```

```
chNext[]
```

$$\{1, -1, 1, 0, -1\} \to g_{1,5} ** d[g_{2,3}]$$

```
chNext[]
```

```
No more cohomologies
```

Degree 1: 0– , 1–, and 2–cohomologies:

```
chCalc[1, 2]
```

```
Total: {{0, 1}, {1, 5}, {6, 0}}
```

```
chNext[]
```

$$\{1, -2, 2, 0, -1\} \to g_{1,5} ** (d[g_{2,3}] \wedge d[g_{2,3}])$$

```
chNext[]
```

```
No more cohomologies
```

Degree 2: 1– , 2–, and 3–cohomologies:

```
chCalc[2, {1, 3}]
```

```
Total: {{0, 4}, {4, 9}, {10, 0}}
```

```
chNext[]
```

$$\{1, -3, 3, 0, -1\} \to g_{1,5} ** (d[g_{2,3}] \wedge d[g_{2,3}] \wedge d[g_{2,3}])$$

```
chNext[]
```

```
No more cohomologies
```

Degree 3: 2– , 3–, and 4–cohomologies:

```
chCalc[3, {2, 4}]
```

```
Total: {{0, 6}, {6, 11}, {12, 0}}
```

```
chNext[]
```

$$\{1, -4, 4, 0, -1\} \to g_{1,5} ** (d[g_{2,3}] \wedge d[g_{2,3}] \wedge d[g_{2,3}] \wedge d[g_{2,3}])$$

```
chNext[]
```

```
No more cohomologies
```

Package `SuperLie`\`Sing`\`: Calculating singular vectors

■ Introduction

Let $\mathfrak{g} = \oplus \mathfrak{g}_i$ be a $\mathbb{Z}-$graded Lie superalgebra, $\mathcal{L}_k = \oplus_{i \geq k} \mathfrak{g}_i$ and V a $\mathfrak{g}_0-$module. The action of $\mathfrak{g}_0$ on V may be extended with 0 to $\mathcal{L}_1$ and then an induced $\mathfrak{g}-$module is defined to be $I(V) = U(\mathfrak{g}_-) \otimes V$.

The problem is to find all possible homomorphisms $I(V_1) \to I(V_2)$ for a certain class of $\mathfrak{g}_0-$modules.

This problem is related to the similar problem for modules dual to induced ones, namely, to the modules of tensor fields.

The image of V under such a homomorphism is a $\mathfrak{g}_0-$module annihilated by $\mathcal{L}_1$. Such modules are called *singular* and their elements are called *singular vectors*.

We consider the case when $\mathfrak{g}_0$ has a Cartan decomposition with raising generators x_1, ..., x_p and lowering generators y_1, ..., y_q, and take the class of irreducible $\mathfrak{g}_0-$modules V^λ with highest weight. In this situation, it suffices to find all *highest* weight singular vectors in $I(V^\lambda)$ for all possible λ's.

To find the highest singular vectors, it suffices to solve the system $\{x_i\, v = 0,\ z_j\, v = 0\}$, where the x_i are the raising generators of $\mathfrak{g}_0$ and z_j are the generators of $\mathcal{L}_1$ considered as an ideal in $\mathcal{L}_1 \oplus \mathfrak{g}_0^+$. Usually (for $\mathfrak{g}$ simple or close to simple), the z_j are only the lowest weight vectors in $\mathfrak{g}_1$, but there are exceptions.

The calculations are performed separately for every `degree` (with respect to the grading of $U(\mathfrak{g}_-)$; this degree is equal to the degree of the corresponding invariant differential operators acting between the dual modules of tensor fields) and `depth` (the difference between the highest weight and the weight of the given vector in Verma module).

First, we calculate the action of x_i and z_j on the module $I(M^\lambda)$ induced from a Verma module M^λ with indefinite highest weight λ, and next we determine the condition on λ when all the images are induced from a submodule in M^λ.

If $\mathfrak{g}_0$ is not a semisimple finite dimensional complex or real Lie algebra, the submodules in $I(V^\lambda)$ may contain multiple highest vectors, so a "gluing" is required.

■ Functions

■ Preparation

These functions are used to describe the problem.

To build the main algebra **g**, use functions defined in **SuperLie** (this step is not described here, see the documentation to **SuperLie**).

To use the enveloping algebras and Verma modules, the vector multiplication should be declared non−symmetric.

□ svSetAlg

svSetAlg[*g*,{*neg*, *zero*, *pos*}] declares the algebra and subalgebras to be used in calculations. The main algebra **g** is divided, according to the grade, in the positive, zero, and negative parts.

The "positive" part may be represented by generators only (as an ideal over the "raising=positive" part of the "zero" component). In this case, *pos* is supplied as *name*→{g1, ...}

□ svScalars

svScalars[*c*,...] declares the indefinite scalar parameters to be used in calculations. The first parameter will be used as an indefinite coefficient of the vectors; other parameters are available for the user. The second parameter is by convention used for the heights weight of the Verma module.

□ svCheckRL

svCheckRL[r, l, d] checks that the proposed lists of the raising, lowering, and diagonal elements agrees with the weight defined in the algebra.

□ svCart

svCart[*x*, *h*, *y*] builds the Cartan decomposition of g_0.

svCart[*x*,*h*,*y*,{*gx*,0,*gy*}] builds the Cartan decomposition with the non−standard gradings of the generators. Here *gx* and *gy* are either lists of generator gradings, or their common grading (if all should have the same degree).

To build the Verma module, the Cartan decomposition should agree with the weight defined on g_0 : $[h_i, x] = w_i[x]\, x$. Use the function svCheckRL to check this.

□ svVerma

svVerma[*m*, λ, *grade*] builds the Verma module over g_0 with an indefinite highest weight λ and the basis of $U({g_0}^+)$; all computations are performed up to the given grade.

To use enveloping algebras and Verma module, the vector multiplication should be declared non−symmetric before calling svVerma:

```
If[SymmetricQ[VTimes], UnSymmetric[VTimes]]
```

□ svLess

The ordering function svLess is used to sort the terms in products in the enveloping algebra $U(\mathfrak{g}_-)$.

If svLess[*x*,*y*] returns True, the terms *x* and *y* are sorted in the order indicated.

The user may add extra definitions to function svLess, or use a different function by assigning sv$Less=*ownLess*.

■ For every degree/depth

□ svDefEq

`svDefEq`[*deg*] builds and partially solves the system of equations for singular vectors of given degree in $I(V)$, with indefinite g_0 −module V and returns the list of depths for which the highest vectors may exist.

The tensor product `Tp` should not be automatically linearly expanded; use `UnLinear`[`Tp`] if it was.

□ svEq

`svEq`[*w*] builds the system of equations for singular vectors in $I(M^\lambda)$ of the degree defined by `svDefEq` and weight $\lambda - w$ (i.e. depth w). The equations are stored as values of **sv'eqHi** ("highness equations") and **sv'eqZ** ("singularity equations").

The function returns the list of substitution rules that implements the map of the indefinite module V to M^λ . The rules contain indefinite coefficients.

While solving the equations, the rules will be specialized by reducing the amount of indefinite coefficients and adding restrictions on the highest weight λ.

■ Calculations

□ svH

`svH` converts the highness conditions **eqHi** using current map. It also assigns the result to **e** and returns the result converted for printing. All printed elements of $U(\mathfrak{g}_-)$ should annihilate the highest weight vector of V^λ (= map the highest weight vector of M^λ to an element of a submodule of M^λ).

□ svZ

`svZ`[*i*] converts the highness conditions **eqZ**⟦*i*⟧ using current map. The function `svZ`[] does the same with all **eqZ**. It assigns the result to **e** and returns the result converted for printing.

□ svSolve

`svSolve` examines the list of vectors **e** (stored by `svH` and `svZ`[]) , tries to solve equations $e_i == 0$ and prints the solutions together with conditions on λ whenever the equations are satisfied. Use `svSub`[*i*] to add the solution of the *i*−th equation to the current map.

□ svAct

`svAct`[*u*,*m*] returns the result of action of $u \in U(\mathfrak{g}_0^+)$ on $m \in I(M^\lambda)$.

□ svSp

`svSp`[*u*, *m*] returns the scalar product of $u \in U(\mathfrak{g}_0^+)$ on $m \in I(M^\lambda)$.
Defined if $w(u) + w(m) = \lambda$.

□ svRep

svRep[*expr*] applies the current map to the expression *expr* and tries to simplify the result.

□ svSub

svSub[*sol*,...] adds the solutions to the current map. Here *sol* may be
(a) a rule λ[_]–>._;
(b) a rule c[__]–>_;
(c) an integer, see svSolve;
(d) an expression *e*, in this case the solution of *e*==0 is added .

□ svExcl

svSub[*expr*] adds the expression *expr* to the list of non–zero expressions. Such expressions will be cancelled in equations. The list of non–zero expressions is valid for the current branch.

□ svBranch

svBranch[*level*] starts new logical branch of the solution. If level is <= the current level, the current map and exclusion list are restored as they were when the level was created.

□ svHiCf

svHiCf returns the coefficient at m_λ of the vector in question, **hiD**.
Since m_λ cannot belong to any submodule of M^λ, this coefficient should be non–zero.
The indeterminate coefficients that are not present in this coefficient are irrelevant.

□ svResult

svResult returns the solution(s) of the current replacement list, as element(s) of M^λ.

□ svImg

svImg[*f*] substitutes the elements of g_-, g_0, and g_+ in the expression f with their images in the main algebra g.

■ Variables

□ sv$g

The value of sv$g is the name of the main algebra g. Assigned by **svDefAlg**.

□ svn, sva, sv$p

The values of svn, sva, and sv$p are the names of the negative (g_-), zeroth (g_0) and positive (g_+) subalgebras of the main algebra. Assigned by **svDefAlg**.

□ svy, svh, sv$x

The names of the negative, zeroth and positive subalgebras in the Cartan decomposition of the algebra $g_0 = g_0{}^- \oplus g_0{}^0 \oplus g_0{}^+$. Assigned by **svCart**.

□ svr, svl, sv$d

The values of `sv$r`, `sv$l`, and `sv$d` are lists of Chevalley–like generators of g_0 (the raising, lowering and diagonal, respectively). Required relations: $[d_i,\ r_j] = w_i(r_j)\, r_j$, where $w(x)$ is the weight of x, and similarly for the l_j. Assigned by **svCart**. The proposed generators may be checked by **svCheckRL**.

□ svm, svλ

The values of `sv$m` and `sv$λ` are the name and the highest weight of the Verma module over g_0. Assigned by **svVerma**.

□ sv$z

The value of `sv$z` is the list of generators of g_+ as $g_0{}^+$–module. May be assigned by **svDefAlg**; otherwise should be assigned manually.

□ sv$v

This is an indefinite g_0–module used in calculations.

□ sv$c

The value of `sv$c` is the name of the scalar coefficient used in calculations.

■ Private part

■ Functions

svRaise[f] for f∈Ind(V), calculates `svAct`[sv$r,f], decomposes the result with respect to the basis of $U(g_-)$ and returns the list of coefficients (elements of V)

uCoords[f] for f∈Ind(V), decomposes f with respect to the basis of $U(g_-)$ and returns the list of coefficients (elements of V)

modBas[d] returns the basis of the d–th component of Ind(M^λ).

eqMm[e] for $e \in M^\lambda$, returns the list of scalar equations equivalent to the condition e∈ maximal submodule. It uses current substitutions.

eqPr[f] for $f \in$ Ind(M^λ) returns the list of scalar equations equivalent to the condition f∈ Ind(maximal submodule). It uses current substitutions.

gPr[f] for $f \in \mathrm{Ind}(M^\lambda)$ calculates the list of scalar equations equivalent to the condition f∈ Ind(maximal submodule), assigns the result to **e** and returns the result converted for printing. Uses current substitutions.

mComp[w] returns the list of element of M^λ with weight $\lambda-$w.

uxComp[x] returns the list of element of $U(g_0{}^+)$ of required weight.
If $x \in M^\lambda$, then the weight is λ−w(x); if x is a list, then the weight is −x.

eqZ returns the equations of singularity.
eqH returns the equations of highness.

svImg[v] returns the image of the element v in the algebra g.

envBas[d] returns the list of elements of grade −d in $U(g_-)$.

■ Data objects

svArep is the decomposition operator of g_0 in the form of a replacement rule.

Package SuperLie`Domain`: Object Oriented Programming in Mathematica

■ Properties

SetProperty

■ SetProperty[*obj*, *property*] defines the property of the object.

■ Both parameters can be lists.

■ Either objects or properties can be with parameters, as *obj*→*value* or *property*→*value* (but not both).

■ Each property must be previously defined using the functions NewProperty, NewValue, NewList or NewDomain. Only one of the properties in the list can be a domain.

■ Another way to set the property of the objects is to call *property*[*obj*, *obj*, ...] or *property*[*obj*–>*value*, ...]. Only the second form is valid for Value and List types of properties.

■ CancelProperty[*obj*, *property*] cancels the property of the object.

■ Both parameters can be lists.

■ Objects as well as property can be with parameters, e.g., *obj*→*value* or *property*→*value* (but not both simultaneously).

■ Another way to cancel a property of an objects is to call Un*property*[*obj*, *obj*, ...] or Un*property*[*obj*→*value*, ...].

Define

■ Define[*obj*, *prop*] clears all settings of the object and calls SetProperty.

■ Define[*obj*, *prop*, {*attr*}] sets attributes of the object before the properties.

ClearDef

■ ClearDef[*name*] clears all settings associated with the symbol *name*.

■ Keys

In *Mathematica*, there are two forms for holding the properties of objects. One is Attributes[obj] whose value is {key, key, ... }, the other one is Options[obj] with value of the form {key–>value, key–>value, ... }.

The Domain package allows one to define other lists of properties, which can contain terms of both types: key or key–>value.

NewList

■ NewList[*name*] defines a new list of properties. The value of *name*[*obj*] will be the list of the properties of the object, *name*[*obj*, *key*] is the value of the property *key* (or True if *key* has no value) and False if *key* has not a term of *name*[*obj*].

■ The expression *name*[*obj*→*list*] is used to add or change the properties, Un*name*[*obj*→*list*] to remove the properties.

■ One function can set properties of several objects: *name*[*obj*→*list*, *obj*→*list*, ...] or *name*[{*obj*,*ob*j,..}→*list*].

■ Each member of the list must be either *key* or *key*→*value*. If the *name*[*obj*] already contains the member with the same key, it will be replaced.

- Another way to set/reset the list of properties (together with properties of other kinds) is `SetProperty`[*obj*, {..., *name*→*list*, ..}] and `CancelProperty`[*obj*, {...,*name*→*list*, ..}].

NewValue

- `NewValue`[*name*,..] declares new properties of the type Value.

- After declaration one can use *name*[*obj*→*val*, *obj*→*val*, ...] to set (or change) the value of the property name and `Un`*name*[*obj*, *obj*, ...] to discard the property.
- The expression *name*[*obj*] returns the value of the property *name* of the object *obj*.
- Another way to set/reset the value of the property (together with properties of other kinds) is `SetProperty`[*obj*, {..., *name*→*val*, ..}] and `CancelProperty`[*obj*, {..., *name*, ..}].

NewProperty

- `NewProperty`[*name*] declares a new property of objects.
- `NewProperty`[*name*, {*method*,...}] defines also the method of setting and resetting the property.

- The predefined methods are:
- `Flag`: *name*[*obj*] sets the flag *name*`Q`[*obj*] = `True`, `Un`*name*[*obj*] cancels this definition. This is the default method, if it is used, then `NewProperty` is called without the second parameter.
- `Flag`→*value* is the same as `Flag` with given value used instead of `True`.
- `Rule`: *name*[*obj*] transforms the replacement rule *name*`Rule`[*obj*] into a definition attached to *obj*; *name*[*obj*→*parm*] uses *name*-`Rule`[*obj*, *parm*]. The function `Un`*name*[*obj*] (`Un`*name*[*obj*→*parm*]) cancels this definition.
- `Rule`→*parm* gives the default value of the parameter for *name*`Rule`[*obj*].
- `VarRule` is the same as `Rule`, but adds to the name of the rule a prefix depending on the domain of the object (see below about domains). So the property may differ from one domain to another.
- `VarRule`→*parm* gives the default value of the parameter.
- `Option`: *name*[*obj*→*val*] sets the option *name*→*val* of the given object; *name*[*obj*] sets *name*→`True`; `Un`*name*[*obj*] deletes the option.
- `Option`→*value* gives the default value of the option (instead of `True`).
- `Also`→{*prop*,..} shows that *name*[*obj*] sets also the properties from the list by calling *parm*[*obj*]. Each of them may have a parameter: *prop*→*parm*, then it is used in the call *prop*[*obj*→*parm*].
- `Format`: shows that *name*[*obj*→*func*] defines the output format as `Format`[*expr_obj*, *name*`Form`] := *func*[*expr*]. Example: `TeX`[*x*→`Subscripted`] defines `Format`[*expr_x*, `TeXForm`] := `Subscripted`[*expr*].

NewDomain

- `NewDomain`[*name*] declares a new domain.

- To attach an object to this domain, enter *name*[*obj*,...] or *name*[*obj*→*parm*,...] if the domain requires a parameter.
- To detach the object, type `Un`*name*[*obj*,...] or `Un`*name*[*obj*→*parm*,...].
- The names of domains can be used also in `SetProperty` and `CancelProperty`.
- `NewDomain`[*name*,{*method*,...},"*prefix*"] defines also the method of attaching and detaching an object to the domain. The methods are the same as for `NewProperty`. The third parameter is the prefix used in `VarRule` method for this domain.

Domain

- `Domain`[*expr*] returns the domain of the value of expression *expr*.
- `Domain`[*op*, *n*, *tot*] returns the domain of the *n*–th argument of the function *op* when it is called with *tot* arguments.

Operation

- `Operation`[*name*, *generic*[*domain*, ...]→*domain*] defines a new operation *name* as a restriction of the *generic* operation if its parameters belong to certain domains.

- In case of operations with a variable number of operands, the repeated operands may be written as *domain*.. (two dots) or *domain*... (three dots; the last version allows zero operands).

- In the case when a repeated group of operands enters a new operation not directly but as a result of another operation, this condition may be written as *domain*..→ *opname*. An example of such a condition is the restriction of the generic `Times` operation (`GTimes`) to the product of a scalar coefficient and the vector: `Operation[SVTimes, GTimes[Scalar..→Times, Vector]→ Vector]`.

■ Tools

SPrint

- `SPrint`[*format*, *val*, ...] converts the arguments to a string. All arguments are the same as in `Print` function.

AutoRule

- `AutoRule`[*rule*] converts the replacement rule or list of rules into definition(s), so the rule will be automatically applied whenever possible. The value of the *rule* is changed to {} (if rule is non–protected) to prevent the repeated attempts to apply the rule.
- `AutoRule`[*rule*, *tag*] attaches the definitions to the given tag.

UnAutoRule

- `UnAutoRule`[*rule*] cancels the definitions made by `AutoRule`[rule] or `AutoRule`[*rule*, *tag*] and restores the value of the *rule*.

NameSuffix

- `NameSuffix`[*name*, "*suffix*"] builds a new name, appending the suffix to the old name.

PrefixName

- `PrefixName["prefix"`, *name*] builds a new name, prepending the prefix to the old name.

Tag

- `Tag`[*expr*] returns the `HoldPattern`[*tag*], where *tag* is the first symbol in the sequence *expr*, `Head`[*expr*], `Head`[`Head`[*expr*]],... .

- Only the head of *expr* is evaluated (as in the left hand side of "=")."

Target

- `Target`[*expr*] evaluates the head and the arguments of the expression and returns the result enclosed in `HoldPattern`[] to prevent further evaluation of the expression. The result may be used as target for assignment, in replacement rules, and so on.
- `Target`[*expr*, *head*] returns *head*[*value*] rather than `HoldPattern`[*value*]. For example, `Print`[`Target`[*expr*, `HoldForm`]] prints partially evaluated expression.

AddHead

- `AddHead`[*head*, *expr*] returns *expr* if `Head`[*expr*]≡*head* and *head*[*expr*] if this expression *expr* has different header.

InfixFormat

- `InfixFormat`[*sep*][*f*] defines the infix output format for the expressions with header *f*: f[*x*1,*x*2,..] ⟶ x1 *sep x*2 ... .
- `InfixFormat`[*sep*, *options*][*f*] defines the infix format with the options:

`Prec` is the precedence level (the default is 100),

`Group` is grouping (the default is None),

Empty is the format for *f*[] (the default is 1),

Type is the format type (the default is OutputForm).

SetFormat

■ SetFormat[*type*, *f*, *fun*] defines the output format for the expressions with header *f*: *f*[*x*,...] ⟶*fun*[*f*[*x*, ...]] .

■ SetFormat[StandardForm, *f*, Subscripted] defines also the interpretation of the subscripted input.

ClearFormat

■ ClearFormat[*type*, *f*] cancels the definition given by SetFormat.

SetToTag

■ SetToTag[*symb*] redefines assignments of expressions with header *symb*, so that assignments *symb*[arg,..] = ... and *symb*[*arg*,..] := ... will be attached to *arg* rather than to *symb*.

Compound

■ Compound[{*f1*,...*fn*}] returns compound function which, if applied to arguments *args,* evaluates (*f1*[*args*]; ...; *fn*[*args*]).

SortKeys

■ SortKeys[*list*] sorts the list in the alphabetical order of the keys.

OrderKeys

■ OrderKeys[arg1,arg2] returns -1, 0, or 1 depending on the order of the keys in arguments (like function Order).

OrderedKeysQ

■ OrderedKeysQ[*list*] returns True if keys in the list are ordered and False otherwise (like function OrderedQ).

SameKeysQ

■ SameKeysQ[*arg*,...] returns True if all arguments have the same keys and False otherwise (like function SameQ).

Merge

■ Merge[*list*1,*list*2,...] is similar to Union, but supports more options:

SameTest→*test* specifies the function used to determine equivalent elements (like in Union);

Sort→*fn* specifies the function to be used for sort the union (like in Sort).

Merge→*fn* specifies the function that is used to merge sets of equivalent elements.

■ See also Union, Sort.

Union

■ Domain‘Union supports the option Sort→*fn* which specifies the function to be used to sort the union (like in Sort).

■ See also System‘Union, Sort.

UnionKeys

■ UnionKeys[*list*, ...] returns the joined list, sorted by the key order, with repeated keys dropped.

DeleteSame

- `DeleteSame`[*list*] deletes from the list the adjacent repeated terms leaving only the first one. `DeleteSame`[*list*, *test*] uses function test instead of `SameQ` to check the equivalence of terms.

KeyValue

- `KeyValue`[*list*,*key*] returns `True` if the list contains the member key, the value if the list contains the member *key*→*value*, and `False` otherwise.

Package SuperLie`Enum`: The package for enumerating the sets

■ Creating enumerated sets

EnumSet

■ EnumSet[*set*, *range*→*comp*, ...] builds a new enumerated set from given components. The result is attached to the symbol in the first argument.

■ The arguments should have format {*start*, *end*, *step*}→{*deg*:→*list*, ... }.

■ This component will be enumerated in the following order: for every *degree* in the range {*start*, *end*, *step*}, the list of the elements is calculated as *degree* /. {*deg*:→*list*, ... }.

■ The value of *end* may be Infinity.

EnumAddTo

■ EnumAddTo[*set*, *range*→*comp*, ...] adds more components to the enumerated set.

■ See EnumSet for the argument format

EnumJoin

■ EnumJoin[*new, set1*, ...] builds new enumerated set joining the enumerated sets *set*1, *set*2, ..., *setn*.

■ EnumJoin[*new*, *old*] builds a duplicate of an old enumerated set.

■ Iteration over enumerated sets

EnumFor

■ EnumFor[*var*, *set*, *options...*, *body*] executes body repeatedly for any *var* from *set*.

■ Options:

■ Range→{*from*, *to*, *step*} restricts the range of the elements' degree (*step* is optional)

■ From→*elt* starts the iteration from the given element. This option is valid only when an earlier iteration reaches this element (using To or Until option)

■ FromNext→*elt* same as From→*elt* but skips the given element.

■ To→*elt* ends the iteration at the given element. The iteration may be resumed from this point using options From or FromNext.

■ Until→*elt* same as To→*elt* but stops before the given element.

EnumTable

■ EnumTable[*expr*, {*var*,*set*,*options...*}] generates a list of the values of *expr* when var runs over all elements of the enumerated set.

■ EnumTable[*expr*, *iter* ...] with several iterators gives a nested list, the first iterator is outermost.

■ The options (see EnumFor) restricts the range of elements.

EnumList

■ EnumList[*set*, *options* ...] generates the list of elements of the enumerated set.

■ The options (see `EnumFor`) restricts the range of the elements.

EnumPoint

■ `EnumPoint`[*var*, *set*, *elt*, *options* ...] finds the location of the *elt* in the set and assigns the *var* to point out the result.

■ The options (see `EnumFor`) restrict the range of the elements.

■ Accessing enumerated sets

Enum

■ `Enum`[*set*, *i*] returns the *i*–th component of the enumeration of the set.

■ Enum[*set*] returns the number of the components in the enumeration of the set.

EnumRange

■ `EnumRange`[*set*, *i*] returns the range of degrees in the *i*–th component of the enumerated set.

■ The format of the result is {*stars*, *end*, *step*}.

TestRange

■ `TestRange`[*value*, *range*] tests whether the value is in the given range.

■ The *range* may be {*end*}, {*start*, *end*} or {*start*, *end*, *step*}

■ The result may be `True`, `False`, `Greater` or `Less`.

Examples

Every example is described "from scratch" as if it is executed in a fresh session. The function `Off[...]` is called to suppress the message `Equations may not give solutions for all "solve" variables`.

■ 1. Classical Lie superalgebras

■ 1.1. Defining relations for $\mathfrak{g}(A)$

In these examples we will find the defining relations between the positive generators of two algebras with Cartan matrix.

□ 1.1.1. Defining relations for $\mathfrak{ag}_2$

```
Needs["SuperLie`"]
Off[Solve::svars]
```

```
CartanMatrixAlgebra[ag, {x, h, y}, {{0, 1, 0}, {-1, 2, -3}, {0, -1, 2}}, ∞, PList -> {1, 0, 0}]
```

```
17|14
```

This is the basis of $x = (\mathfrak{ag}_2)_+$ in terms of generators x_1, x_2, x_3:

```
GenBasis[ag] // ColumnForm
```

```
x1
x2
x3
[x1, x2]
[x2, x3]
[x2, [x2, x3]]
[x3, [x1, x2]]
[x2, [x2, [x2, x3]]]
[[x1, x2], [x2, x3]]
[[x1, x2], [x2, [x2, x3]]]
[[x2, x3], [x2, [x2, x3]]]
[[x2, [x2, x3]], [x3, [x1, x2]]]
[[x3, [x1, x2]], [x2, [x2, [x2, x3]]]]
[[x3, [x1, x2]], [[x1, x2], [x2, [x2, x3]]]]
```

These are the defining relations between generators:

```
GenRel[ag] // ColumnForm
```

```
[x1, x1] → 0
[x1, x3] → 0
[x2, [x1, x2]] → 0
[x3, [x2, x3]] → 0
[x2, [x2, [x2, [x2, x3]]]] → 0
```

□ 1.1.2. Defining relations for $\mathfrak{d}(\alpha)^{(1)}$

All calculations are done up to degree 12

```
Needs["SuperLie`"]
Off[Solve::svars]
```

Declare a scalar parameter α.

```
Scalar[α];
```

Build the algebra up to degree 12:

```
CartanMatrixAlgebra[g, {x, h, y}, {{2, 0, 0, -1}, {0, 2, 0, -1}, {0, 0, 2, -1}, {-1, -α, 1 + α, 0}}, 12, PList → {0, 0, 0, 1}]
```

```
46|40
```

Here are the relations:

```
GenRel[g] // ColumnForm
```

$$[x_4, x_4] \to 0$$
$$[x_1, x_2] \to 0$$
$$[x_1, x_3] \to 0$$
$$[x_2, x_3] \to 0$$
$$[x_1, [x_1, x_4]] \to 0$$
$$[x_2, [x_2, x_4]] \to 0$$
$$[x_3, [x_3, x_4]] \to 0$$
$$[[x_2, x_4], [[x_1, x_4], [x_3, x_4]]] \to \frac{\alpha}{-1-\alpha} [[x_3, x_4], [[x_1, x_4], [x_2, x_4]]]$$

■ 1.2. Defining relations in vectorial algebras

In the following examples we will show how to find defining relations between elements of an existing (i.e., already defined) algebra.

□ 1.2.1. Algebra of polynomial vector fields $\mathfrak{vect}(2|1)$

Let us find the relations between the positive generators of $\mathfrak{vect}(2|1)$ with respect to the standard grading, see [GL].

```
Needs["SuperLie`"]
Off[Solve::svars]
```

```
SuperLie Package Version 2.03 installed
Disclaimer: This software is provided "AS IS", without a warranty of any kind
```

First let us define the algebra $\mathfrak{vect}(2|1)$. We will define it as algebra of vector fields on a space x of dimension (2|1).

We will use the operation `VTimes` for multiplication of polynomials, so it should be declared `Symmetric` (i.e., supercommutative). The tensor product `Tp` should be declared `Linear` (i.e., automatically expanded via linearity).

```
Symmetric[VTimes];
Linear[Tp];
VectorSpace[x, Dim → {2, 1}, CoLeft → v];
VectorLieAlgebra[g, x]
```

```
g = vect(x)
```

Find the lowest vectors in g_1 :

```
GeneralZero[{x2 ** v1 , x3 ** v2 }, Basis[g, 1], c, Lb]
```

```
c[2] (x1 x3 ) ** v1 + c[1] (x2 x3 ) ** v1 + c[2] (x2 x3 ) ** v2
```

Build a subalgebra with generators $e_1 = x_1 ** v_2$, $e_2 = x_2 ** v_3$, $z_1 = x_2\, x_3 ** v_1$, $z_2 = x_1\, x_3 ** v_1 + x_2\, x_3 ** v_2$.

We will restrict the calculations to elements of grade ≤4 (note that the first two generators has grade 0).

```
SubAlgebra[s, g,
 {e1 → x1 ** v2 , e2 → x2 ** v3 , z1 → (x2 x3 ) ** v1 , z2 → (x1 x3 ) ** v1 + (x2 x3 ) ** v2 }, Grade → 4]
```

```
s is a sublagebra in g
```

Here are the relations (to start every relation with a fresh line, we print the results in the infix format with the newline symbol as separator; `ColumnForm` does not work because some relations does not fit in one line):

```
GenRel[s] // InfixFormat["\n", Prec -> 1000]
```

```
[e2 , e2 ] == 0
[e1 , [e1 , e2 ]] == 0
[e1 , z2 ] == 0
[e1 , [e1 , [e1 , z1 ]]] == 0
[[e1 , e2 ], [e2 , z2 ]] == 0
[z1 , z1 ] == 0
[z1 , z2 ] == 0
[z2 , z2 ] == 0
[z1 , [e1 , [e1 , z1 ]]] == 0
[[e1 , z1 ], [e2 , z1 ]] == -3 [z2 , [e2 , z1 ]]
[[e2 , z1 ], [e1 , [e1 , z1 ]]] == 2 [z2 , [[e1 , e2 ], z1 ]] - 2 [[e1 , z1 ], [e2 , z2 ]]
[[e2 , z1 ], [[e1 , e2 ], z1 ]] == - [[e2 , z1 ], [e2 , z2 ]]
[[[e1 , e2 ], z1 ], [[e1 , e2 ], z2 ]] == 1/2 [[e2 , z2 ], [e2 , [e1 , [e1 , z1 ]]]] -
   1/2 [[e2 , z2 ], [[e1 , e2 ], [e1 , z1 ]]] + 1/2 [[e2 , [e1 , z1 ]], [[e1 , e2 ], z2 ]]
[[[e1 , e2 ], z1 ], [e2 , [e1 , [e1 , z1 ]]]] ==
  [[e2 , z2 ], [[e1 , e2 ], [e1 , z1 ]]] - [[e2 , [e1 , z1 ]], [[e1 , e2 ], z2 ]]
[[e2 , z1 ], [z2 , [e2 , z1 ]]] == 0
[[[e1 , e2 ], z1 ], [z2 , [e2 , z1 ]]] == 0
[[[e1 , e2 ], [e1 , z1 ]], [z2 , [e2 , z1 ]]] ==
  1/2 [[e2 , z2 ], [[e1 , z1 ], [e2 , z2 ]]] - [[[e1 , e2 ], z2 ], [z2 , [e2 , z1 ]]]
[[[e1 , z1 ], [e2 , z2 ]], [[e2 , z1 ], [e2 , z2 ]]] == -1/3 [[e2 , z2 ], [[e2 , z2 ], [z2 , [e2 , z1 ]]]]
[[[e2 , z1 ], [e2 , z2 ]], [[e2 , z2 ], [[e1 , e2 ], z1 ]]] ==
  1/6 [[e2 , z2 ], [[e2 , z2 ], [[e2 , z1 ], [e2 , z2 ]]]]
```

□ 1.2.2. Algebra ℏ(2|1)

Let us find the relations between the positive generators of H(2|1) with respect to the standard grading, see [GL].

```
Needs["SuperLie`"]
Off[Solve::svars]
```

First, let us define the algebra H(2|1). We will define it on polynomials of p, q, θ.

We will use the operation `VTimes` for multiplication of polynomials, so it should be declared `Symmetric`.

```
TrivialSpace[p];
TrivialSpace[q];
TrivialSpace[θ, 1];
Symmetric[VTimes];
HamiltonAlgebra[g, {p, θ, q}]
```

```
g is a Hamiltonian algebra over {p, θ, q}
```

Find the lowest vectors in g_1 :

```
GeneralZero[{q^2, qθ}, DegreeBasis[3, {p, θ, q}], c, Hb]
```

```
c[1] q^3
```

Build a subalgebra with generators $e_1 = p^2$, $e_2 = p\theta$, $z = q^3$, up to grade 5.

Note that `HamiltonAlgebra` does not define grading on polynomials, so we should specify the grading explicitly.

```
SubAlgebra[s, g, {e1 → p^2, e2 → pθ, z → q^3}, Grade → {Deg[#, BasisPattern[g]] - 2 &, 5}]
```

```
s is a sublagebra in g
```

Here are the relations:

```
GenRel[s] // ColumnForm
```

```
[e1, e2] == 0
[e2, e2] == - e1
[e1, [e1, [e1, [e1, z]]]] == 0
[e2, [e1, [e1, [e1, z]]]] == 0
[z, [e2, z]] == 0
[[e1, z], [e1, [e1, z]]] == [z, [e1, [e1, [e1, z]]]]
[[e2, z], [e1, [e1, z]]] == 4 [[e1, z], [e2, [e1, z]]]
[z, [z, [e1, z]]] == 0
[[e1, [e1, [e1, z]]], [z, [e1, z]]] == 3/5 [[e1, [e1, z]], [z, [e1, [e1, z]]]]
[[z, [e1, [e1, z]]], [z, [e1, [e1, [e1, z]]]]] == 2/3 [[z, [e1, z]], [[e1, z], [e1, [e1, [e1, z]]]]]
```

□ 1.2.3. Algebra ħ(4|0)

Let us find the relations between the positive generators of ħ(4|0) with respect to the standard grading, see [GL].

```
Needs["SuperLie`"]
Off[Solve::svars]
```

First, let us define the algebra ħ(4). We will define it on polynomials of p_i, q_i.

We will use the operation `VTimes` for multiplication of polynomials, so it should be declared `Symmetric`.

```
VectorSpace[p, Dim → 2];
VectorSpace[q, Dim → 2];
Symmetric[VTimes];
HamiltonAlgebra[g, {p, q}]
```

```
g is a Hamiltonian algebra over {p, q}
```

Find the lowest vectors in $\mathfrak{g}_1$:

```
GeneralZero[{p2 q1 , q2^2 }, DegreeBasis[3, {p1 , p2 , q1 , q2 }], c, Hb]
```

```
c[1] q1^3
```

Build a subalgebra with generators $e_1 = p_1\, q_2$, $e_2 = p_2^2$, $z = q_1^3$, up to grade 5.

Note that `HamiltonAlgebra` does not define grading on polynomials, so we should specify the grading explicitly.

```
SubAlgebra[s, g, {e1 → p1 q2 , e2 → p2^2 , z → q1^3 }, Grade → {Deg[#, BasisPattern[g]] - 2 &, 5}]
```

Here are the relations:

```
GenRel[s] // InfixFormat["\n", Prec -> 1000]
```

```
[e2 , [e1 , e2 ]] == 0
[e1 , [e1 , [e1 , e2 ]]] == 0
[e2 , z] == 0
[e1 , [e1 , [e1 , [e1 , z]]]] == 0
[z, [e1 , z]] == 0
[[e1 , z], [e1 , [e1 , z]]] == 0
[[[e1 , e2 ], z], [e1 , [e1 , [e1 , z]]]] == -3/5 [[e1 , [e1 , z]], [[e1 , e2 ], [e1 , z]]]
[[[e1 , [e1 , e2 ]], z], [[e1 , [e1 , e2 ]], [e1 , z]]] ==
  -4 [[[e1 , e2 ], [e1 , z]], [[e1 , e2 ], [e1 , [e1 , z]]]] -
   2 [[[e1 , e2 ], [e1 , z]], [[e1 , [e1 , e2 ]], [e1 , z]]]
[[[e1 , [e1 , e2 ]], [e1 , z]], [[e1 , [e1 , e2 ]], [[e1 , e2 ], z]]] ==
  -1/4 [[[e1 , e2 ], [e1 , [e1 , z]]], [[e1 , e2 ], [[e1 , e2 ], [e1 , z]]]] -
   3/8 [[[e1 , e2 ], [[e1 , e2 ], z]], [[e1 , [e1 , e2 ]], [e1 , [e1 , z]]]] -
   5/4 [[[e1 , [e1 , e2 ]], [e1 , z]], [[e1 , e2 ], [[e1 , e2 ], [e1 , z]]]]
[[[e1 , [e1 , e2 ]], [[e1 , [e1 , e2 ]], [[e1 , [e1 , e2 ]], z]]], [[e1 , z], [[e1 , e2 ], z]]] ==
  -3 [[[e1 , [e1 , e2 ]], [[e1 , [e1 , e2 ]], [e1 , z]]], [[[e1 , e2 ], z], [[e1 , [e1 , e2 ]], z]]] -
   3 [[[e1 , [e1 , e2 ]], [[e1 , [e1 , e2 ]], [[e1 , e2 ], z]]], [[e1 , [e1 , z]], [[e1 , e2 ], z]]]
```

■ 1.3. Defining relations for $\mathfrak{gl}(\lambda)$

```
Needs["SuperLie`"]
Off[Solve::svars]
```

The algebra $\mathfrak{gl}(\lambda)$ is defined as a subalgebra in $\mathfrak{diff}(1)$ with generators shown below.

Since the *grades* of the generators have different signs, it is not possible to restrict the calculations to some grade. Instead, we restrict the computation to some *degree* with respect to the generators assuming that the degree of each generator is equal to 1.

```
TrivialSpace[u];
DiffAlgebra[Diff, u];
Scalar[λ];
SubAlgebra[s, Diff, {x^+ → u^2 d[u] - (λ - 1) u, x^- → -d[u], z → d[u]^2 }, GRange → 10]
```

```
s is a sublagebra in Diff
```

Here are the relations:

```
GenRel[s] // ColumnForm
```

$$[x^-, z] = 0$$
$$[x^+, [x^+, x^-]] = -2\,(x^+)$$
$$[x^-, [x^+, x^-]] = 2\,(x^-)$$
$$[z, [x^+, x^-]] = 4\,z$$
$$[z, [z, [x^+, z]]] = 0$$
$$[[x^+, z], [x^+, [x^+, z]]] = 8\,(-4+\lambda^2)\,(x^-) + \frac{2}{3}\,[z, [x^+, [x^+, [x^+, z]]]]$$
$$[x^+, [x^+, [x^+, [x^+, [x^+, z]]]]] = 0$$
$$[[z, [x^+, z]], [x^+, [x^+, [x^+, z]]]] = -144\,(-9+\lambda^2)\,z - \frac{3}{4}\,[[x^+, [x^+, z]], [z, [x^+, [x^+, z]]]]$$

■ 1.4. Defining relations for $\mathfrak{diff}(1)$

```
Needs["SuperLie`"]
Off[Solve::svars]
```

Case n=1:

```
TrivialSpace[q];
DiffAlgebra[Diff, q];
SubAlgebra[s, Diff,
 {e → VTimes[], q → q, dq → d[q], x → d[q]^2, y → q^2, h → q d[q], z → q^3}, GRange → 16]
```

```
s is a sublagebra in Diff
```

Here are the relations:

```
GenRel[s] // InfixFormat["\n", Prec -> 1000]
```

$$[e, q] = 0$$
$$[e, dq] = 0$$
$$[e, x] = 0$$
$$[e, y] = 0$$
$$[e, h] = 0$$
$$[e, z] = 0$$
$$[q, dq] = -e$$
$$[q, x] = -2\,dq$$
$$[q, y] = 0$$
$$[q, h] = -q$$
$$[q, z] = 0$$
$$[dq, x] = 0$$
$$[dq, y] = 2\,q$$
$$[dq, h] = dq$$
$$[dq, z] = 3\,y$$
$$[x, y] = 2\,e + 4\,h$$
$$[x, h] = 2\,x$$
$$[y, h] = -2\,y$$
$$[y, z] = 0$$
$$[h, z] = 3\,z$$
$$[z, [z, [x, z]]] = 0$$
$$[x, [x, [x, [x, z]]]] = 0$$
$$[[x, z], [x, [x, z]]] = 144\,e + [z, [x, [x, [x, z]]]]$$
$$[[z, [x, z]], [x, [x, [x, z]]]] = \frac{41472}{5}\,q - \frac{3}{5}\,[[x, [x, z]], [z, [x, [x, z]]]]$$
$$[[z, [x, [x, z]]], [z, [x, [x, [x, z]]]]] = 435456\,y + \frac{4}{5}\,[[x, [x, [x, z]]], [[x, z], [z, [x, z]]]]$$

Case n=2:

```
VectorSpace[q, Dim → 2];
DiffAlgebra[Diff, q];
SubAlgebra[s, Diff, {e → VTimes[], q1 → q1, q2 → q2, dq1 → d[q1],
  dq2 → d[q2], x1 → q2 d[q1], x2 → d[q2]^2, y1 → q1 d[q2], y2 → q2^2,
  y3 → q1 q2, y4 → q1^2, h1 → q1 d[q1], h2 → q2 d[q2], z → q1^3}, GRange → 15]
```

```
s is a sublagebra in Diff
```

Here are the relations:

```
GenRel[s] // InfixFormat["\n", Prec -> 1000]
```

```
[e, q1] == 0
[e, q2] == 0
[e, dq1] == 0
[e, dq2] == 0
[e, x1] == 0
[e, x2] == 0
[e, y1] == 0
[e, y2] == 0
[e, y3] == 0
[e, y4] == 0
[e, h1] == 0
[e, h2] == 0
[e, z] == 0
[q1, q2] == 0
[q1, dq1] == - e
[q1, dq2] == 0
[q1, x1] == - q2
[q1, x2] == 0
[q1, y1] == 0
[q1, y2] == 0
[q1, y3] == 0
[q1, y4] == 0
[q1, h1] == - q1
[q1, h2] == 0
[q1, z] == 0
[q2, dq1] == 0
[q2, dq2] == - e
[q2, x1] == 0
[q2, x2] == -2 dq2
[q2, y1] == - q1
[q2, y2] == 0
[q2, y3] == 0
[q2, y4] == 0
[q2, h1] == 0
[q2, h2] == - q2
[q2, z] == 0
[dq1, dq2] == 0
[dq1, x1] == 0
[dq1, x2] == 0
[dq1, y1] == dq2
[dq1, y2] == 0
[dq1, y3] == q2
[dq1, y4] == 2 q1
[dq1, h1] == dq1
[dq1, h2] == 0
[dq1, z] == 3 y4
[dq2, x1] == dq1
[dq2, x2] == 0
[dq2, y1] == 0
[dq2, y2] == 2 q2
[dq2, y3] == q1
[dq2, y4] == 0
[dq2, h1] == 0
[dq2, h2] == dq2
[dq2, z] == 0
[x1, y1] == - h1 + h2
[x1, y2] == 0
[x1, y3] == y2
[x1, y4] == 2 y3
[x1, h1] == x1
[x1, h2] == - x1
[x2, y1] == 0
[x2, y2] == 2 e + 4 h2
[x2, y3] == 2 y1
[x2, y4] == 0
[x2, h1] == 0
[x2, h2] == 2 x2
[x2, z] == 0
```

```
[y1, y2] == 2 y3
[y1, y3] == y4
[y1, y4] == 0
[y1, h1] == - y1
[y1, h2] == y1
[y1, z] == 0
[y2, y3] == 0
[y2, y4] == 0
[y2, h1] == 0
[y2, h2] == -2 y2
[y2, z] == 0
[y3, y4] == 0
[y3, h1] == - y3
[y3, h2] == - y3
[y3, z] == 0
[y4, h1] == -2 y4
[y4, h2] == 0
[y4, z] == 0
[h1, h2] == 0
[h1, z] == 3 z
[h2, z] == 0
[x2, [x1, x2]] == 0
[z, [x1, z]] == 0
[x1, [x1, [x1, x2]]] == 0
[x1, [x1, [x1, [x1, z]]]] == 0
[[x1, z], [x1, [x1, z]]] == 0
```

$$[[z, [x1, x2]], [x1, [x1, [x1, z]]]] == \frac{3}{5} [[x1, [x1, z]], [[x1, x2], [x1, z]]]$$

$$[[[x1, x2], [x1, z]], [[x1, z], [x1, [x1, x2]]]] == \frac{1}{2} [[z, [x1, [x1, x2]]], [[x1, z], [x1, [x1, x2]]]] + 2 [[[x1, x2], [x1, z]], [[x1, x2], [x1, [x1, z]]]]$$

$$[[[x1, z], [x1, [x1, x2]]], [[x1, [x1, x2]], [z, [x1, x2]]]] == -144 e - \frac{1}{4} [[[x1, x2], [x1, [x1, z]]], [[x1, x2], [[x1, x2], [x1, z]]]] + \frac{3}{8} [[[x1, x2], [z, [x1, x2]]], [[x1, [x1, x2]], [x1, [x1, z]]]] + \frac{5}{4} [[[x1, z], [x1, [x1, x2]]], [[x1, x2], [[x1, x2], [x1, z]]]]$$

$$[[[x1, [x1, z]], [[x1, x2], [x1, z]]], [[x1, [x1, x2]], [[x1, x2], [z, [x1, x2]]]]] == 25920 q1 - \frac{5}{2} [[[x1, [x1, x2]], [[x1, x2], [x1, z]]], [[z, [x1, [x1, x2]]], [[x1, x2], [x1, z]]]]$$

■ 2. Singular vectors

■ 2.1 $\mathfrak{k}(1|6)$

Not written yet; for the answer, see [GLS]

■ 2.2 $\mathfrak{kas}$

Not written yet; for the answer, see [GLS]

■ 3. Cohomology with various coefficients

■ 3.1 Trivial coefficients

□ 3.1.1. $H^i(\mathfrak{g})$ for $\mathfrak{g} = \mathfrak{h}^0(0 \mid n)$, n=4

```
Needs["SuperLie`"]
Off[Solve::svars]
```

```
SuperLie Package Version 2.03 installed
Disclaimer: This software is provided "AS IS", without a warranty of any kind
```

Define $\mathfrak{h}^0$ as subalgebra in the Hamiltonian algebra

```
n = 4;
VectorSpace[θ, Dim → (0 | n)];
Symmetric[VTimes];
HamiltonAlgebra[ham, {θ}]
```

```
ham is a Hamiltonian algebra over {θ}
```

```
bas = Rest[UpToDegreeBasis[n - 1, Array[θ, n]]]
SubAlgebra[g, ham, bas, Grade → (Deg[#, _θ] - 2 &)]
```

```
{θ1, θ2, θ3, θ4, θ1 θ2, θ1 θ3, θ1 θ4, θ2 θ3, θ2 θ4, θ3 θ4, θ1 θ2 θ3, θ1 θ2 θ4, θ1 θ3 θ4, θ2 θ3 θ4}
```

```
g is a sublagebra in ham
```

Prepare calculations:

```
Needs["SuperLie`Cohom`"]
chSetAlg[g, dg, None, 1]
chScalars[b, c];
Jacobi[Act - wedge];
ch$Out := (# /. {g[i_] :> Image[g][[i]], dg[i_] :> d[Image[g][[i]]]}) &
```

Calculations (the grades of $\mathfrak{g}_i$ are from −1 to 1, so the grade of r−cohomology may be from −r to r)

```
chCalc[-4, 4]
```

```
Total: {0, 0, 0, 0, {1, 34}}
```

The result shows the dimension of Im d and Ker d. One 4−cohomology is found. Print it.

```
chNext[]
```

```
→ d[θ1 θ2 θ3] ∧ d[θ1 θ2 θ3] ∧ d[θ1 θ2 θ3] ∧ d[θ1 θ2 θ3] +
  2 d[θ1 θ2 θ3] ∧ d[θ1 θ2 θ3] ∧ d[θ1 θ2 θ4] ∧ d[θ1 θ2 θ4] +
  2 d[θ1 θ2 θ3] ∧ d[θ1 θ2 θ3] ∧ d[θ1 θ3 θ4] ∧ d[θ1 θ3 θ4] +
  2 d[θ1 θ2 θ3] ∧ d[θ1 θ2 θ3] ∧ d[θ2 θ3 θ4] ∧ d[θ2 θ3 θ4] +
  d[θ1 θ2 θ4] ∧ d[θ1 θ2 θ4] ∧ d[θ1 θ2 θ4] ∧ d[θ1 θ2 θ4] +
  2 d[θ1 θ2 θ4] ∧ d[θ1 θ2 θ4] ∧ d[θ1 θ3 θ4] ∧ d[θ1 θ3 θ4] +
  2 d[θ1 θ2 θ4] ∧ d[θ1 θ2 θ4] ∧ d[θ2 θ3 θ4] ∧ d[θ2 θ3 θ4] +
  d[θ1 θ3 θ4] ∧ d[θ1 θ3 θ4] ∧ d[θ1 θ3 θ4] ∧ d[θ1 θ3 θ4] +
  2 d[θ1 θ3 θ4] ∧ d[θ1 θ3 θ4] ∧ d[θ2 θ3 θ4] ∧ d[θ2 θ3 θ4] +
  d[θ2 θ3 θ4] ∧ d[θ2 θ3 θ4] ∧ d[θ2 θ3 θ4] ∧ d[θ2 θ3 θ4]
```

```
chCalc[-3, 4]
```

```
Total: {0, 0, 0, {0, 20}, {20, 100}}
```

No cohomology of degree −3.

```
chCalc[-2, 4]
```

```
Total: {0, 0, {1, 9}, {9, 51}, {52, 178}}
```

There are: one 2−cohomology and one 4−cohomology of degree −2. To save the volume of this book, we do not print all cohomologies.

```
chCalc[-1, 4]
```

```
Total: {0, {0, 4}, {4, 20}, {20, 80}, {80, 240}}
```

```
chCalc[0, 4]
```

```
Total: {{1, 0}, {0, 6}, {7, 24}, {25, 91}, {92, 263}}
```

```
chNext[]
```

```
→ 1
```

```
chNext[]
```

```
→ {{{c[1] → 0}}, {4, 4, 4, 4, 4, 4, 4}}
```

Only the coefficient c[1] gives a cohomology. Print it.

```
chBook[c[1] → 1]
```

$$-d[\Theta_1]\wedge d[\Theta_2\,\Theta_3\,\Theta_4]+d[\Theta_2]\wedge d[\Theta_1\,\Theta_3\,\Theta_4]-d[\Theta_3]\wedge d[\Theta_1\,\Theta_2\,\Theta_4]+d[\Theta_4]\wedge d[\Theta_1\,\Theta_2\,\Theta_3]$$

```
chNext[]
```

```
→ {{{c[16] → -c[18] - c[22] - c[25]}},
  {8, 6, 6, 6, 6, 8, 8, 6, 6, 6, 6, 6, 6, 6, 6, 4, 8, 4, 8, 6, 8, 4, 6, 6, 4}}
```

```
chBook[c[16] → 1]
```

$$-d[\Theta_1]\wedge d[\Theta_1\,\Theta_2]\wedge d[\Theta_1\,\Theta_3\,\Theta_4]+d[\Theta_1]\wedge d[\Theta_1\,\Theta_3]\wedge d[\Theta_1\,\Theta_2\,\Theta_4]-d[\Theta_1]\wedge d[\Theta_1\,\Theta_4]\wedge d[\Theta_1\,\Theta_2\,\Theta_3]+d[\Theta_1\,\Theta_2]\wedge d[\Theta_1\,\Theta_3]\wedge d[\Theta_1\,\Theta_4]$$

```
chNext[]
```

```
→ {{{c[25] → 0}}, {12, 8, 9, 9, 8, 12, 18, 14, 16, 15, 15, 22, 14, 14, 8, 18, 8, 9, 18, 23, 20,
   19, 14, 6, 10, 18, 23, 15, 9, 9, 8, 8, 23, 15, 20, 9, 9, 8, 8, 27, 10, 9, 9, 9, 8, 9,
   8, 21, 23, 19, 18, 10, 9, 10, 9, 22, 17, 9, 14, 10, 9, 9, 10, 9, 10, 9, 10, 9, 18, 10,
   18, 18, 8, 8, 9, 27, 17, 14, 20, 9, 21, 23, 19, 8, 16, 12, 19, 8, 12, 12, 12, 12}}
```

```
chBook[c[25] → 1]
```

$$\begin{aligned}&d[\Theta_1]\wedge d[\Theta_1]\wedge d[\Theta_2\,\Theta_3\,\Theta_4]\wedge d[\Theta_2\,\Theta_3\,\Theta_4]-2\,d[\Theta_1]\wedge d[\Theta_2]\wedge d[\Theta_1\,\Theta_3\,\Theta_4]\wedge d[\Theta_2\,\Theta_3\,\Theta_4]+\\&2\,d[\Theta_1]\wedge d[\Theta_3]\wedge d[\Theta_1\,\Theta_2\,\Theta_4]\wedge d[\Theta_2\,\Theta_3\,\Theta_4]-2\,d[\Theta_1]\wedge d[\Theta_4]\wedge d[\Theta_1\,\Theta_2\,\Theta_3]\wedge d[\Theta_2\,\Theta_3\,\Theta_4]+\\&d[\Theta_2]\wedge d[\Theta_2]\wedge d[\Theta_1\,\Theta_3\,\Theta_4]\wedge d[\Theta_1\,\Theta_3\,\Theta_4]-2\,d[\Theta_2]\wedge d[\Theta_3]\wedge d[\Theta_1\,\Theta_2\,\Theta_4]\wedge d[\Theta_1\,\Theta_3\,\Theta_4]+\\&2\,d[\Theta_2]\wedge d[\Theta_4]\wedge d[\Theta_1\,\Theta_2\,\Theta_3]\wedge d[\Theta_1\,\Theta_3\,\Theta_4]+d[\Theta_3]\wedge d[\Theta_3]\wedge d[\Theta_1\,\Theta_2\,\Theta_4]\wedge d[\Theta_1\,\Theta_2\,\Theta_4]-\\&2\,d[\Theta_3]\wedge d[\Theta_4]\wedge d[\Theta_1\,\Theta_2\,\Theta_3]\wedge d[\Theta_1\,\Theta_2\,\Theta_4]+d[\Theta_4]\wedge d[\Theta_4]\wedge d[\Theta_1\,\Theta_2\,\Theta_3]\wedge d[\Theta_1\,\Theta_2\,\Theta_3]\end{aligned}$$

```
chNext[]
```

```
No more cohomologies
```

```
chCalc[1, 4]
```

```
Total: {0, {0, 4}, {4, 20}, {20, 80}, {80, 240}}
```

```
chCalc[2, 4]
```

```
Total: {0, 0, {1, 9}, {9, 51}, {52, 178}}
```

```
chCalc[3, 4]
```

```
Total: {0, 0, 0, {0, 20}, {20, 100}}
```

```
chCalc[4, 4]
```

```
Total: {0, 0, 0, 0, {1, 34}}
```

Final result: one 0–cohomology, three 2–cohomologies, one 3–cohomology, and five 4–cohomologies.

□ 3.1.2. $H^i(\mathfrak{g})$ for $\mathfrak{g} = \mathfrak{h}^0(0 \mid n)$, n=5

```
Needs["SuperLie`"]
Off[Solve::svars]
```

Define $\mathfrak{h}^0$ as subalgebra in the Hamiltonian algebra.
To simplify the problem,

```
k = 2;
VectorSpace[ξ, Dim → (0 | k)];
VectorSpace[η, Dim → (0 | k)];
TrivialSpace[θ, 1];
Symmetric[VTimes];
HamiltonAlgebra[ham, {ξ, θ, η}]
```

```
ham is a Hamiltonian algebra over {ξ, θ, η}
```

```
bas = Rest[UpToDegreeBasis[2 k, Join[{θ}, Array[ξ, k], Array[η, k]]]]
SubAlgebra[g, ham, bas, Grade → (Deg[#, θ | _ξ | _η] - 2 &)]
```

```
{θ, ξ1, ξ2, η1, η2, θ ξ1, θ ξ2, θ η1, θ η2, ξ1 ξ2, ξ1 η1, ξ1 η2, ξ2 η1, ξ2 η2,
 η1 η2, θ ξ1 ξ2, θ ξ1 η1, θ ξ1 η2, θ ξ2 η1, θ ξ2 η2, θ η1 η2, ξ1 ξ2 η1, ξ1 ξ2 η2,
 ξ1 η1 η2, ξ2 η1 η2, θ ξ1 ξ2 η1, θ ξ1 ξ2 η2, θ ξ1 η1 η2, θ ξ2 η1 η2, ξ1 ξ2 η1 η2}
```

```
g is a sublagebra in ham
```

Prepare calculations

```
Needs["SuperLie`Cohom`"]
chSetAlg[g, dg, None, 1]
chScalars[b, c];
Jacobi[Act → wedge];
ch$Out := (# /. {g[i_] :> Image[g][[i]], dg[i_] :> d[Image[g][[i]]]}) &
```

Calculations (the grades of $\mathfrak{g}_i$ are from −1 to 2, so the grade of r–cohomology may be from −2r to r)

```
chCalc[-6, 3]
```

```
Total: {0, 0, 0, {0, 10}}
```

```
chCalc[-5, 3]
```

```
Total: {0, 0, 0, {1, 99}}
```

The result shows the dimension of Im d and Ker d. One 3–cohomology found. Print it.

chNext[]

$$\to d[\theta\,\xi_1\,\xi_2]\wedge d[\theta\,\xi_1\,\eta_1\,\eta_2]\wedge d[\theta\,\xi_2\,\eta_1\,\eta_2]-d[\theta\,\xi_1\,\eta_1]\wedge d[\theta\,\xi_1\,\xi_2\,\eta_1]\wedge d[\theta\,\xi_1\,\eta_1\,\eta_2]-d[\theta\,\xi_1\,\eta_2]\wedge d[\theta\,\xi_1\,\xi_2\,\eta_1]\wedge d[\theta\,\xi_2\,\eta_1\,\eta_2]-d[\theta\,\xi_2\,\eta_1]\wedge d[\theta\,\xi_1\,\xi_2\,\eta_2]\wedge d[\theta\,\xi_2\,\eta_1\,\eta_2]+d[\theta\,\eta_1\,\eta_2]\wedge d[\theta\,\xi_1\,\xi_2\,\eta_1]\wedge d[\theta\,\xi_1\,\xi_2\,\eta_2]+d[\xi_1\,\xi_2\,\eta_1]\wedge d[\theta\,\xi_1\,\eta_1\,\eta_2]\wedge d[\xi_1\,\xi_2\,\eta_1\,\eta_2]+d[\xi_1\,\xi_2\,\eta_2]\wedge d[\theta\,\xi_2\,\eta_1\,\eta_2]\wedge d[\xi_1\,\xi_2\,\eta_1\,\eta_2]+d[\xi_1\,\eta_1\,\eta_2]\wedge d[\theta\,\xi_1\,\xi_2\,\eta_1]\wedge d[\xi_1\,\xi_2\,\eta_1\,\eta_2]+d[\xi_2\,\eta_1\,\eta_2]\wedge d[\theta\,\xi_1\,\xi_2\,\eta_2]\wedge d[\xi_1\,\xi_2\,\eta_1\,\eta_2]$$

chCalc[-4, 3]

```
Total: {0, 0, {0, 10}, {10, 365}}
```

No cohomology of degree –4.

chCalc[-3, 3]

```
Total: {0, 0, {0, 50}, {50, 720}}
```

No cohomology of degree –3.

chCalc[-2, 3]

```
Total: {0, {0, 5}, {5, 100}, {100, 925}}
```

No cohomology of degree –2.

chCalc[-1, 3]

```
Total: {0, {0, 10}, {10, 115}, {115, 860}}
```

No cohomology of degree –1.

chCalc[0, 3]

```
Total: {{1, 0}, {0, 10}, {10, 85}, {85, 610}}
```

chNext[]

$$\to 1$$

chNext[]

```
No more cohomologies
```

A single 0–cohomology found.

chCalc[1, 3]

```
Total: {0, {0, 5}, {5, 45}, {45, 330}}
```

chCalc[2, 3]

```
Total: {0, 0, {1, 14}, {14, 136}}
```

chNext[]

$$\to \frac{1}{2}\,d[\theta]\wedge d[\theta]+d[\xi_1]\wedge d[\eta_1]+d[\xi_2]\wedge d[\eta_2]$$

A single 2–cohomology found.

chCalc[3, 3]

```
Total: {0, 0, 0, {0, 35}}
```

No cohomology of degree 3.

Final result: one 0–cohomology, one 2–cohomology, one 3–cohomology.

■ 3.2 SUGRA (H^2 (g_ ; g) ,where g = sl(4 | n))

Let $\mathfrak{e}$=sl(par), there par is the *format* of the matrix algebra, i.e., list of parities of rows and columns; n = dim($\mathfrak{g}$) = length(par) = $\sum n_i$ and $\mathfrak{h}_i \subset \mathfrak{e}$ are subalgebras of matrices formed by consecutive n_i rows and columns (i.e., $\mathfrak{h}_1$ formed by first n_1 columns and rows, $\mathfrak{h}_2$ - by the next n_2 columns and rows, etc); $\mathfrak{h} = \oplus\mathfrak{h}_i$ is the algebra block–diagonal matrices ; $\mathfrak{h}_0 \subset \mathfrak{h}$ is the even part of $\mathfrak{h}$; $\mathfrak{n} \subset \mathfrak{e}$ is the subalgebra of block underdiagonal matrices; $\mathfrak{b} = \mathfrak{n} + \mathfrak{h}$ (direct sum of vector spaces but not algebras)

Let us find the cohomology (only the first and the second one in this example, but the higher cohomology may be calculated in the same way) of $\mathfrak{n}$ with coefficients in $\mathfrak{b}$ (the generalized Riemann case) and in $\mathfrak{e}$ (the conformal case).

```
Needs["SuperLie`"];
Needs["SuperLie`Cohom`"];
```

First we should define the algebra. We have a series of algebras, so we write a *Mathematica* program that defines the algebra and all the subalgebras mentioned above for given format *par* and block decomposition $\{n_i\}$. The program also will define some other things required for calculation of cohomology. The grading on $\mathfrak{e}$ is defined so that the subalgebra $\mathfrak{h}$ has grade 0, the blocks under diagonal have grade –1 and so on.

```
DefTask[par_, blocksizes_] :=
  Module[{dim, k, raise, lower, cent},
   dim = Length[par];
   If[dim ≠ Plus @@ blocksizes, Message[DefTask::dim]; Return[$Failed]];
   slAlgebra[e, PList → par];
   block$sizes = blocksizes;
   n$blocks = Length[blocksizes];
   par$ = par;
   block$no = Flatten[Table[i, {i, n$blocks}, {blocksizes[[i]]}]];
   (* block number for rows/columns *)
   block$ind = Flatten[Table[j, {i, n$blocks}, {j, blocksizes[[i]]}]];
   (* indices inside blocks *)
   Grade[e[i_, j_]] ^:= block$no[[j]] - block$no[[i]];
   (* the grading defined by the block decomposition *)
   SubAlgebra[n, e, Select[Basis[e], Grade[#] < 0 &]];
   SubAlgebra[h, e, Join[Select[Array[e, dim], block$no[[#[[1]]]] == block$no[[#[[1]] + 1]],
     Select[Basis[e], (Length[#] == 2 ∧ Grade[#] == 0) &]]];
   RestrictModule[n, h]; (* define the action of h on n *)
   (* list of positive and negative generators in h_0 *)
   ch$raise = Select[Basis[h], P[#] == 0 ∧ With[{m = Mapping[h, e][#]}, m[[2]] == m[[1]] + 1] &];
   ch$lower = Select[Basis[h], P[#] == 0 ∧ With[{m = Mapping[h, e][#]}, m[[2]] == m[[1]] - 1] &];
   (* the condition when a weight may be a highest weight of a h_0-module *)
   hi$cond[w_] = And @@ (With[{m = Mapping[h, e][#]}, w[[m[[1]]]] ≥ w[[m[[2]]]] & /@ ch$raise);
   (* initialize the Cohom package *)
   chSetAlg[n, dn, h];
  ];
DefTask::dim = "The sum of block sizes is not equal to the size of the whole matrix";
```

We are calculating cohomology with two sets of coefficients, so we should store the results in separate places. To do this, we assign values to "store" variables `ch$res`, `ch$tab` and `ch$book`. The variable `ansInt` will be also used for storing results. The functions below will initialize calculating in respective case. They return the minimal and the maximal grading of r–cochains for $0 \le r \le 2$.

```
ConformalCase[] :=
(conf$ = True;
```

```
  maxcfdeg = n$blocks - 1;
  mincfdeg = 1 - n$blocks;
  ch$res = resConf;
  ch$tab = tbConf;
  ch$book = ansConf;
  Clear[resConf, tbConf, ansConf];
  tbConf[_] = {};
  ansConf[__] = 0;
  {mincfdeg, maxcfdeg + 2 (n$blocks - 1)})

RiemannCase[] :=
(conf$ = False;
  maxcfdeg = 0;
  mincfdeg = 1 - n$blocks;
  ch$res = resRm;
  ch$tab = tbRm;
  ch$book = ansRm;
  Clear[resRm, tbRm, ansRm, ansInt];
  tbRm[_] = {};
  ansRm[__] = 0;
  ansInt[___] = 0;
  {mincfdeg, 2 (n$blocks - 1)})
```

Here is the maximal and minimal possible grade of d–forms:

```
MaxDeg[d_] := maxcfdeg + d * (n$blocks - 1);
MinDeg[d_] := mincfdeg + d;
```

Definitions required for **Cohom** package. The splitting function is used to decompose modules in sums of homogeneous parts. We will split the modules into parts of fixed weight and "total parity". The "total parity" of an r–form ω is defined as `(P[ω] + r) mod 2`.

```
chScalars[b, c];
chSplit[x_] :=
  With[{w = Weight[x]}, If[hi$cond[w], {Mod[P[x] + Deg[x, _ch$d], 2], w}, SkipVal]];

(*chSplit[x_]:=With[{w=Weight[x]},If[hi$cond[w],w,SkipVal]];*)
```

The function `chBasis[d]` should return the list of the elements of grade d in the basis of the module of coefficients:

```
mComp[d_] := Select[If[conf$, Basis[e], Join[Image[n], Image[h]]], Grade[#] == d &];
chBasis[d_] := If[conf$ || d ≤ 0, mComp[d], {}];

Linear[Tp];
Jacobi[Act -> {wedge, Tp}];
Off[Solve::svars]
```

The function `Der` calculates the derivative but does not normalize the result. So we define `NDer` as derivative with normalization.

```
NDer[f_] := VNormal[Der[f]];
NDer0[f__] := VNormal[Der0[f]];
```

Now we will define the output format. Every matrix element is written as a letter with 4 indices. The subscripts denote the coordinates of the block (row and column) and the superscripts denote the row and the column inside the block. The diagonal elements have only 2 indices. The superscripts that correspond to a block of size 1 are omitted.

The letters used for matrix elements are H for diagonal elements, X for even elements of diagonal blocks, Y for odd elements of diagonal blocks, A for other even elements, Q for other odd elements. The forms are denoted by "hat": $\hat{A}$ and $\hat{Q}$.

For diagonal blocks of size 2×2, the superscripts 12 and 21 at X and Y are replaced with "+" and "–"; the superscript of H is omitted. The diagonal elements that are not elements of blocks are written with two subscript indices that denote the adjacent blocks.

```
WithoutPreSL[
OutFn[v_] := v //.
    {e[i_] :> With[{c = block$no[[i]], j = block$ind[[i]]},
       Which[
        j == block$sizes[[c]], H_StringForm["````",c,c+1],
        block$sizes[[c]] == 2, H_c,
        True, H_c^j]],
     e[i_, j_] :>
      With[{ci = block$no[[i]], cj = block$no[[j]], ki = block$ind[[i]], kj = block$ind[[j]]},
       If[ci == cj,
        (If[P[e[i, j]] == 0, X, Y])_ci^If[block$sizes[[ci]]==2, If[i>j,"-","+"], StringForm["````",ki,kj]],
        (If[P[e[i, j]] == 0, A,
           Q])_StringForm["````",ci,cj]^StringForm["````",If[block$sizes[[ci]]>1,ki,""],If[block$sizes[[cj]]>1,kj,""]]]],
     dn[i_] :> (Image[n][[i]] /. e -> de),
     de[i_, j_] :>
      With[{ci = block$no[[i]], cj = block$no[[j]], ki = block$ind[[i]], kj = block$ind[[j]]},
       (If[P[e[i, j]] == 0, Â,
          Q̂])_StringForm["````",ci,cj]^StringForm["````",If[block$sizes[[ci]]>1,ki,""],If[block$sizes[[cj]]>1,kj,""]]]}];

ch$Out = OutFn;
```

The following function will try to integrate the generalized Riemann cohomology as a cycle in the conformal case. The function defined below works with highest (with respect to the action of $\hbar_0$) cycles only.

```
ConfIntegrate[f_] :=
  With[{ff = If[ListQ[f], GeneralSum[b, f], f]},
   Module[{v, deg = Grade[ff], r = Deg[ff, _ch$d], w = ch$Wt[ff], dv, res},
    v = Select[Block[{conf$ = True}, ch$basis[r - 1, deg]], ch$Wt[#] === w &];
    DPrint[2, "deg=", deg, ", w=", w, ", basis:", v];
    v = chHVect[v]; If[v == 0, Return[0]];
    DPrint[2, "hVect:", v];
    dv = NDer[v]; If[dv == 0, Return[0]];
    DPrint[2, "deg=", deg, ", w=", w, ", dv=", dv];
    res = GeneralSolve[dv == ff, v, c, b];
    If[res === $Failed, 0, VNormal[res], c]]];
```

Store the integral:

```
ires[v_] := With[{r = v /. _c → 1}, ansInt[chPos[]] = r; ch$Out[r]]
ires[v_, rep_] := With[{r = v /. rep}, ansInt[chPos[]] = r; ch$Out[r]]
ires[v_, rep__] := (ansInt[chPos[]] = (With[{r = v /. #}, Print[ch$Out[r]]; r] & /@ {rep});)
```

Here we define short commands for invoking most used functions:

```
integ := ConfIntegrate[ch$book[chPos[]]]
next := chNext[]
```

Now we define the TEX format:

```
Format[OverHat[A_], TeXForm] := StringForm["\\hat ``", A]
WithoutPreSL[
  Unprotect[Power];
  Format[Power[Subscript[a_, i_], j_String], TeXForm] := Subsuperscript[a, i, j];
  Format[Power[Subscript[a_, i_], j_StringForm], TeXForm] := Subsuperscript[a, i, j];
  Protect[Power]];
Format[a_ ** b_, TeXForm] := StringForm["``\cdot ``", a, b]
Format[a_ ∧ b_, TeXForm] := Infix[{a, b}, "\\wedge "]

Format[e_wt, TeXForm] := SequenceForm["(", Infix[e, ",", 0], ")"]

Format[e_label, TeXForm] := SequenceForm[e[[1]], ": \n", e[[2]]]
```

```
tOut[v_] := TeXForm[ch$Out[v /. wedge[e__] :> Infix[{e}, "\\wedge ", 145]]]
```

Now the define some auxiliary functions for working with TEX files:

```
texfile = With[{file = "FileName" /. NotebookInformation[InputNotebook[]]},
   ToFileName[First[file], StringReplace[file[[2]], ".nb" -> ".tex"]]];

OpenTeX[] :=
  (tex = OpenWrite[texfile, FormatType → OutputForm, PageWidth → Infinity];
   Write[tex, "% Cocycles which represent structure functions on M(N)"];
   Write[tex, "\\documentclass[12pt]{amsart}"];
   Write[tex, "\\hoffset=-2cm\\voffset=1cm\\topmargin=-0.5in"];
   Write[tex, "\\textheight=24cm\\textwidth=16.5cm"];
   Write[tex, "\\begin{document}"];);

CloseTeX[final_: False] :=
  (If[final, WriteCR[]; Write[tex, "\\end{document}"]]; Close[tex]);

ReopenTeX[] :=
 (tex = OpenAppend[texfile, FormatType → OutputForm, PageWidth → Infinity];
  WriteCR[];)
```

Print the header and the footer of the table for current format:

```
OutHead[] :=
 (Write[tex, ""];
  Write[tex, "\\vskip 0.3cm"];
  Write[tex, "Par = {\\bf ", Sequence @@ par$, "}"];
  Write[tex, "\\vskip 0.2cm"];
  Write[tex, "\\begin{tabular}{|c|c|l|r|}"];
  Write[tex, "\\hline\r"];
  Write[tex, "$",
   Subscript["\\text{deg}", "\\text{par}"] // TeXForm, "$&weight&dim&note\\\\"];
  Write[tex, "\\hline"];
  comm = 0; mix = False;)

OutTail[] :=
 (Write[tex, "\\end{tabular}"];
  If[mix, Write[tex, ""]; Write[tex, "${}^*$ Mixed cohomologies"]];
  Write[tex, "\\vskip 0.2cm"];
  Write[tex, "\\noindent {\\bf Notes:}"];
  Write[tex, ""];
 )
```

Print rows for given degree [and parity]:

```
OutDeg[deg_] := {OutDeg[deg, 0], OutDeg[deg, 1]}

OutDeg[deg_, par_] :=
 Module[{nres, fres, lst = {}, wt, i, j, nr, fr, nk, ni, fk, fi, nd, fd, nmix, fmix},
  nres = tbRm[deg]; (* result table, Reimann case *)
  fres = tbConf[deg]; (* result table, Conformal case *)
  For[i = 1, i ≤ Length[nres], i++,
   nr = nres⟦i⟧; (* the current element, Reimann case *)
   wt = nr⟦1⟧;
   If[wt⟦1⟧ ≠ par, Continue[]];
   fr = Cases[fres, {wt, ___}]; (* the current element, conformal case *)
   If[fr === {} ∨ Length[fr⟦1⟧] < 4 ∨ fr⟦1, 4⟧ === "", fi = fd = 0,
    (*else*)fr = fr⟦1⟧; fk = fr⟦4, 1⟧; fi = If[ListQ[fr⟦3⟧], fr⟦3, 2⟧, 0]; fd = fk - fi];
   If[Length[nr] < 4 ∨ nr⟦4⟧ === "", ni = nd = 0,
    (*else*)nk = nr⟦4, 1⟧; ni = If[ListQ[nr⟦3⟧], nr⟦3, 2⟧, 0]; nd = nk - ni];
   If[nd > 0, Print[{nd, wt, deg}]];
   If[nd == fd == 0, Continue[]];
   comm++;
   mix = mix ∨ ni > 0 ∨ fi > 0;
   lst = {lst, wt → {nd, fd, ni > 0, fi > 0, deg, comm}}];
  For[j = 1, j ≤ Length[fres], j++,
   fr = fres⟦j⟧;
   wt = fr⟦1⟧;
   If[wt⟦1⟧ ≠ par, Continue[]];
```

```
    If[Cases[nres, {wt, ___}] =!= {}, Continue[]];
    If[Length[fr] < 4, fi = fd = 0,
     (*else*)fk = fr[[4, 1]]; fi = If[ListQ[fr[[3]]], fr[[3, 2]], 0]; fd = fk - fi];
    If[fd == 0, Continue[]];
    comm++;
    mix = mix ∨ fi > 0;
    lst = {lst, wt → {0, fd, False, fi > 0, deg, comm}}];
   If[lst =!= {},
    lst = Flatten[lst];
    Write[tex, "$", Subscript[deg, par] // TeXForm, "$"];
    For[i = 1, i ≤ Length[lst], i++,
     nr = lst[[i]];
     wt = nr[[1]];
     nr = nr[[2]];
     Write[tex,
      " & $(", Infix[wt[[2]], ",", 0], ")$ & $", nr[[1]], If[nr[[1]] > 0 ∧ nr[[3]], "^{*}", ""],
      "/", nr[[2]], If[nr[[2]] > 0 ∧ nr[[4]], "^{*}", ""], "$ & ", nr[[6]], " \\\\"];
     Write[tex, If[i < Length[lst], "\\cline{2-4}", "\\hline"]]]];
   lst]
```

Select the information to be printed as notes:

```
FilterNote[wt_ → {__, deg_, note_Integer}] :=
  (DPrint[1, "Filter notes for w=", wt, ",  deg=", deg];
   note → {wt[[2]], ansRm[deg, 2, wt], ansConf[deg, 2, wt], ansInt[deg, 2, wt]});
FilterNote[wt_ → {__, ""}] := Unevaluated[];
```

Compose and print a note to the table of cohomology. Use the following information: note number, weight, cohomology in the generalized Riemann case and in the conformal case, the integral of the Riemann cohomology as a conformal cochain:

```
OutNot[no_ → {wt_, rm_, conf_, intg_}] :=
 Module[{v, both, ng, fl, n = no},
  DPrint[1, "Rm: ", rm, ", Conf: ", conf];
  Which[
   rm === 0 ∨ rm === conf, OutVect[no, conf],
   conf === 0, OutVect[no, rm],
   True,
   DPrint[1, "rm: ", rm, ", conf: ", conf];
   fl = If[ListQ[conf], conf, {conf}];
   ng = If[ListQ[rm], rm, {rm}];
   both = Intersection[fl, ng];
   DPrint[1, "both: ", both];
   If[both =!= {}, fl = Complement[fl, both]; ng = Complement[ng, both];
    OutVect[n, both]; n = 0];
   If[ng =!= {}, OutVect[n, ng, "(Riemann case)"]; n = 0];
   If[fl =!= {}, OutVect[n, fl, "(Conformal case)"]]];
  If[intg =!= 0,
   OutCR[];
   OutNote[If[ListQ[intg],
     "Conformally invariant 1-cochains:", "Conformally invariant 1-cochain:"]];
   OutVect[intg]];
 ]
```

Print a vector or a list of vectors in a note with eventual comment:

```
OutVect[n_Integer, v_List, text_: ""] :=
 (If[n > 0, Write[tex, n, ") $", tOut[v[[1]]], "$ ", text],
   Write[tex, "$", tOut[v[[1]]], "$ ", text]];
  Do[Write[tex, "\\vskip 0.05cm"]; Write[tex, "$", tOut[v[[i]]], "$ ", text]
   , {i, 2, Length[v]}];
  Write[tex, "\\vskip 0.1cm"];)

OutVect[v_List, text_: ""] :=
 (OutCR[];
  Do[Write[tex, "\\vskip 0.05cm"]; Write[tex, "$", tOut[v[[i]]], "$ ", text]
   , {i, 1, Length[v]}];
  Write[tex, "\\vskip 0.1cm"];)

OutNote[text_] :=
 (Write[tex, text];)
```

```
OutCR[] := (Write[tex]; Write[tex])

OutNote[n_Integer, text_] :=
 (OutCR[];
  Write[tex, "\\vskip 0.1cm"];
  Write[tex, "\\noindent {\\bf ", n, "}. ", text]; Write[tex];)

OutVect[n_Integer, v_, txt_: ""] :=
 (OutCR[];
  If[n > 0, Write[tex, n, ") $", tOut[v], "$ ", txt],
   Write[tex, "$", tOut[v], "$ ", txt]];
  Write[tex, "\\vskip 0.1cm"];
  v)

OutVect[v_, txt_: ""] :=
 (Write[tex, "$", tOut[v], "$ ", txt];
  Write[tex, "\\vskip 0.1cm"];
  v)

OutnVect[n_Integer, v_, txt_: ""] :=
 (OutCR[];
  Write[tex, n, ") $", tOut[v], "$ ", txt];
  Write[tex, "\\vskip 0.1cm"];
  v)
```

Print the results of calculations as a table with notes:

```
OutRes[] :=
 Module[{lst = {}},
  OutHead[];
  Do[lst = {lst, OutDeg[deg]}, Evaluate[Prepend[minmax[tbRm, tbConf], deg]]];
  OutTail[];
  DPrint[1, "Writing notes ..."];
  OutNot /@ (FilterNote /@ Flatten[lst])]

minmax[tb__] :=
 Module[{min = ∞, max = -∞, i, j},
  For[j = 1, j ≤ Length[{tb}], j++,
   With[{tt = {tb}[[j]]},
    ((i = #[[1, 1, 1]]; If[NumberQ[i], min = Min[min, i]; max = Max[max, i]]) &) /@
     DownValues[tt]]];
  {min,
   max}]
```

■ Algebra $\mathfrak{sl}$({0,0,1,1,0,0}), blocks {2,2,2}

```
DefTask[{0, 0, 1, 1, 0, 0}, {2, 2, 2}]
```

□ Cohomologies of $\mathfrak{n} = \mathfrak{g}_-$ with coefficients in $\mathfrak{g}$

```
ConformalCase[]
```

```
{-2, 6}
```

Grade r=−2

```
chCalc[-2, 2]
```

```
Total: {{1, 0}, 0, 0}
```

```
next
```

$$\{0, \{0, -1, 0, 0, 1, 0\}\} \to A_{31}^{12}$$

Grade r=−1

chCalc[-1, 2]

```
Total: {{0, 2}, {4, 0}, 0}
```

next

$$\{1, \{0, -1, 1, 0, 1, -1\}\} \to A_{31}^{12} ** \hat{Q}_{32}^{21}$$

next

$$\{1, \{1, -1, 0, -1, 1, 0\}\} \to A_{31}^{12} ** \hat{Q}_{21}^{21}$$

next

```
No more cohomologies
```

Grade r=0

chCalc[0, 2]

```
Total: {{0, 5}, {5, 11}, {14, 0}}
```

next

$$\{0, \{0, -1, 2, 0, 1, -2\}\} \to A_{31}^{12} ** \left(\hat{Q}_{32}^{21} \wedge \hat{Q}_{32}^{21}\right)$$

next

$$\{0, \{1, -1, 1, -1, 1, -1\}\} \to A_{31}^{12} ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{32}^{21}\right)$$

next

$$\{0, \{2, -1, 0, -2, 1, 0\}\} \to A_{31}^{12} ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{21}^{21}\right)$$

next

```
No more cohomologies
```

Grade r=1

chCalc[1, 2]

```
Total: {{0, 2}, {2, 16}, {16, 24}}
```

next

```
No more cohomologies
```

Grade r=2

chCalc[2, 2]

```
Total: {{0, 1}, {1, 18}, {20, 56}}
```

next

$$\{0, \{1, -1, 1, 1, -1, -1\}\} \to X_1^{+} ** \left(\hat{Q}_{32}^{12} \wedge \hat{Q}_{32}^{21}\right) - X_1^{+} ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{32}^{22}\right) + Q_{21}^{12} ** \left(\hat{Q}_{32}^{12} \wedge \hat{A}_{31}^{21}\right) - Q_{21}^{12} ** \left(\hat{Q}_{32}^{22} \wedge \hat{A}_{31}^{11}\right) - Q_{21}^{22} ** \left(\hat{Q}_{32}^{11} \wedge \hat{A}_{31}^{21}\right) + Q_{21}^{22} ** \left(\hat{Q}_{32}^{21} \wedge \hat{A}_{31}^{11}\right)$$

next

$$\{0, \{1, 1, -1, -1, 1, -1\}\} \to -Q_{32}^{11} ** \left(\hat{Q}_{21}^{22} \wedge \hat{A}_{31}^{21}\right) + Q_{32}^{11} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{22}\right) + Q_{32}^{12} ** \left(\hat{Q}_{21}^{12} \wedge \hat{A}_{31}^{21}\right) - Q_{32}^{12} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{22}\right) - X_3^+ ** \left(\hat{Q}_{21}^{12} \wedge \hat{Q}_{21}^{21}\right) + X_3^+ ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{21}^{22}\right)$$

next

```
No more cohomologies
```

Grade r=3

chCalc[3, 2]

```
Total: {0, {0, 8}, {8, 58}}
```

Grade r=4

chCalc[4, 2]

```
Total: {0, {0, 4}, {4, 52}}
```

Grade r=5

chCalc[5, 2]

```
Total: {0, 0, {0, 18}}
```

Grade r=6

chCalc[6, 2]

```
Total: {0, 0, {0, 4}}
```

□ Cohomologies of $\mathfrak{n} = \mathfrak{g}_-$ with coefficients in $\mathfrak{g}_- + \mathfrak{g}_0$

RiemannCase[]

```
{-2, 4}
```

Grade r=−2

chCalc[-2, 2]

```
Total: {{1, 0}, 0, 0}
```

next

$$\{0, \{0, -1, 0, 0, 1, 0\}\} \to A_{31}^{12}$$

Grade r=−1

chCalc[-1, 2]

```
Total: {{0, 2}, {4, 0}, 0}
```

next

$$\{1, \{0, -1, 1, 0, 1, -1\}\} \to A_{31}^{12} ** \hat{Q}_{32}^{21}$$

next

$$\{1, \{1, -1, 0, -1, 1, 0\}\} \to A_{31}^{12} ** \hat{Q}_{21}^{21}$$

Grade r=0

```
chCalc[0, 2]
```

```
Total: {{0, 3}, {5, 11}, {14, 0}}
```

```
next
```

$$\{0, \{0, -1, 2, 0, 1, -2\}\} \to A^{12}_{31} ** \left(\hat{Q}^{21}_{32} \wedge \hat{Q}^{21}_{32}\right)$$

```
integ
```

```
0
```

```
next
```

$$-Q^{11}_{21} ** \hat{Q}^{11}_{21} - Q^{12}_{21} ** \hat{Q}^{12}_{21} - Q^{21}_{21} ** \hat{Q}^{21}_{21} - Q^{22}_{21} ** \hat{Q}^{22}_{21} + Q^{11}_{32} ** \hat{Q}^{11}_{32} + Q^{12}_{32} ** \hat{Q}^{12}_{32} + Q^{21}_{32} ** \hat{Q}^{21}_{32} + Q^{22}_{32} ** \hat{Q}^{22}_{32}$$

$$-Q^{11}_{21} ** \hat{Q}^{11}_{21} - Q^{12}_{21} ** \hat{Q}^{12}_{21} - Q^{21}_{21} ** \hat{Q}^{21}_{21} - Q^{22}_{21} ** \hat{Q}^{22}_{21} + A^{11}_{31} ** \hat{A}^{11}_{31} + A^{12}_{31} ** \hat{A}^{12}_{31} + A^{21}_{31} ** \hat{A}^{21}_{31} + A^{22}_{31} ** \hat{A}^{22}_{31}$$

$$\{0, \{0, 0, 0, 0, 0, 0\}\} \to 2$$

```
next
```

$$\{0, \{1, -1, 1, -1, 1, -1\}\} \to A^{12}_{31} ** \left(\hat{Q}^{21}_{21} \wedge \hat{Q}^{21}_{32}\right)$$

```
integ
```

```
0
```

```
next
```

$$\{0, \{2, -1, 0, -2, 1, 0\}\} \to A^{12}_{31} ** \left(\hat{Q}^{21}_{21} \wedge \hat{Q}^{21}_{21}\right)$$

```
integ
```

```
0
```

```
next
```

```
No more cohomologies
```

Grade r=1

```
chCalc[1, 2]
```

```
Total: {0, {0, 14}, {16, 24}}
```

```
next
```

$$\{1, \{0, 0, 1, 0, 0, -1\}\} \to \{\{\{c[1] \to c[2] - c[3]\}\}, \{6, 8, 6, 8\}\}$$

Here we have a mixed cohomology. The dimension of closed cochains is 4 while the dimension of exact cochains is 3. The chain is exact if the coefficients in the expression (not shown here) satisfies $c_1 = c_2 - c_3$. So we should choose some set of c_i that do not satisfy the equation. The list {6,8,6,8} shows the number of times the c_i occurs in the expression, so we select $c_1 = 1$ and $c_i = 0$ for $i \neq 0$ as, hopefully, simplest expression:

```
chBook[c[1] → 1]
```

$$-Q_{21}^{11} ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{21}^{21}\right) - Q_{21}^{12} ** \left(\hat{Q}_{21}^{22} \wedge \hat{Q}_{32}^{22}\right) + Q_{21}^{21} ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{32}^{21}\right) +$$
$$Q_{21}^{22} ** \left(\hat{Q}_{21}^{22} \wedge \hat{Q}_{32}^{21}\right) - Q_{32}^{12} ** \left(\hat{Q}_{32}^{12} \wedge \hat{Q}_{32}^{21}\right) + Q_{32}^{12} ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{32}^{22}\right)$$

Check if this cohomology is a derivative of some 1−cochain in conformal case:

integ

$$-\frac{1}{2} e_1 ** dn_8 - e_2 ** dn_8 + \frac{1}{2} (1 - 2\,c[1])\, e_3 ** dn_8 +$$
$$(-1 - c[1])\, e_4 ** dn_8 + \frac{1}{2} (1 + 2\,c[1])\, e_5 ** dn_8 + c[1]\, e_{3,1} ** dn_{12} +$$
$$c[1]\, e_{3,2} ** dn_{11} + (1 - c[1])\, e_{3,4} ** dn_6 - c[1]\, e_{5,6} ** dn_5$$

Store the integral:

ires[VNormal[2 %], c[1] → 0]

$$-H_1 ** \hat{Q}_{32}^{21} - 2\,H_{12} ** \hat{Q}_{32}^{21} + H_2 ** \hat{Q}_{32}^{21} - 2\,H_{23} ** \hat{Q}_{32}^{21} + H_3 ** \hat{Q}_{32}^{21} + 2\,X_2^+ ** \hat{Q}_{32}^{22}$$

next

```
{1, {1, 0, 0, -1, 0, 0}} → {{{c[1] → c[2] + c[3]}}, {8, 6, 6, 8}}
```

chBook[c[2] → 1]

$$-Q_{21}^{12} ** \left(\hat{Q}_{21}^{12} \wedge \hat{Q}_{21}^{21}\right) + Q_{21}^{12} ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{21}^{22}\right) + Q_{32}^{11} ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{21}^{21}\right) -$$
$$Q_{32}^{12} ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{11}\right) + Q_{32}^{21} ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{32}^{21}\right) - Q_{32}^{22} ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{21}\right)$$

integ

$$\frac{1}{2} (1 + 2\,c[1])\, e_1 ** dn_7 + (1 + c[1])\, e_2 ** dn_7 + \frac{1}{2} (1 - 2\,c[1])\, e_3 ** dn_7 + e_4 ** dn_7 -$$
$$\frac{1}{2} e_5 ** dn_7 + c[1]\, e_{1,2} ** dn_4 + (-1 + c[1])\, e_{3,4} ** dn_3 + c[1]\, e_{5,4} ** dn_{10} + c[1]\, e_{6,4} ** dn_{12}$$

ires[VNormal[2 %], c[1] → 0]

$$H_1 ** \hat{Q}_{21}^{21} + 2\,H_{12} ** \hat{Q}_{21}^{21} + H_2 ** \hat{Q}_{21}^{21} + 2\,H_{23} ** \hat{Q}_{21}^{21} - H_3 ** \hat{Q}_{21}^{21} - 2\,X_2^+ ** \hat{Q}_{21}^{11}$$

next

```
No more cohomologies
```

Grade r=2

chCalc[2, 2]

```
Total: {0, {0, 5}, {12, 52}}
```

next

$$\{0, \{0, 0, 1, 1, 0, -2\}\} \to$$
$$H_2 ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{32}^{21}\right) - Q_{21}^{11} ** \left(\hat{Q}_{32}^{22} \wedge \hat{A}_{31}^{21}\right) - Q_{21}^{12} ** \left(\hat{Q}_{32}^{22} \wedge \hat{A}_{31}^{22}\right) + X_2^+ ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{32}^{22}\right) +$$
$$Q_{21}^{21} ** \left(\hat{Q}_{32}^{21} \wedge \hat{A}_{31}^{21}\right) + Q_{21}^{22} ** \left(\hat{Q}_{32}^{21} \wedge \hat{A}_{31}^{22}\right) - X_2^- ** \left(\hat{Q}_{32}^{21} \wedge \hat{Q}_{32}^{21}\right) - X_3^+ ** \left(\hat{Q}_{32}^{12} \wedge \hat{Q}_{32}^{21}\right) + X_3^+ ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{32}^{22}\right)$$

integ

$$e_{3,6} ** dn_6 - e_{4,6} ** dn_8$$

ires[%]

$$Q_{23}^{12} ** \hat{Q}_{32}^{22} - Q_{23}^{22} ** \hat{Q}_{32}^{21}$$

next

$$\{0, \{0, 0, 2, 0, -1, -1\}\} \to$$
$$- H_3 ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{32}^{21}\right) - Q_{21}^{11} ** \left(\hat{Q}_{32}^{11} \wedge \hat{A}_{31}^{21}\right) + Q_{21}^{11} ** \left(\hat{Q}_{32}^{21} \wedge \hat{A}_{31}^{11}\right) - Q_{21}^{12} ** \left(\hat{Q}_{32}^{11} \wedge \hat{A}_{31}^{22}\right) +$$
$$Q_{21}^{12} ** \left(\hat{Q}_{32}^{21} \wedge \hat{A}_{31}^{12}\right) - X_2^{+} ** \left(\hat{Q}_{32}^{12} \wedge \hat{Q}_{32}^{21}\right) + X_2^{+} ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{32}^{22}\right) + X_3^{+} ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{32}^{11}\right) - X_3^{-} ** \left(\hat{Q}_{32}^{21} \wedge \hat{Q}_{32}^{21}\right)$$

integ

$$- e_{3,5} ** dn_8 + e_{3,6} ** dn_5$$

ires[%]

$$- Q_{23}^{11} ** \hat{Q}_{32}^{21} + Q_{23}^{12} ** \hat{Q}_{32}^{11}$$

next

$$\{0, \{1, -1, 1, 1, -1, -1\}\} \to X_1^{+} ** \left(\hat{Q}_{32}^{12} \wedge \hat{Q}_{32}^{21}\right) - X_1^{+} ** \left(\hat{Q}_{32}^{11} \wedge \hat{Q}_{32}^{22}\right) +$$
$$Q_{21}^{12} ** \left(\hat{Q}_{32}^{12} \wedge \hat{A}_{31}^{21}\right) - Q_{21}^{12} ** \left(\hat{Q}_{32}^{22} \wedge \hat{A}_{31}^{11}\right) - Q_{21}^{22} ** \left(\hat{Q}_{32}^{11} \wedge \hat{A}_{31}^{21}\right) + Q_{21}^{22} ** \left(\hat{Q}_{32}^{21} \wedge \hat{A}_{31}^{11}\right)$$

integ

$$0$$

next

$$\{0, \{1, 0, 0, 0, 0, -1\}\} \to \left\{\left\{c[1] \to \frac{3\, c[2]}{8} + \frac{c[3]}{8}\right\}\right\}, \{24, 21, 21\}\}$$

chBook[c[3] → 1]

$$\frac{1}{4} H_1 ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{21}\right) + \frac{1}{4} H_1 ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{21}^{21}\right) - \frac{1}{4} H_2 ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{21}\right) +$$
$$\frac{1}{4} H_2 ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{21}^{21}\right) + \frac{1}{4} H_3 ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{21}\right) + \frac{1}{4} H_3 ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{21}^{21}\right) +$$
$$\frac{1}{2} X_1^{+} ** \left(\hat{Q}_{21}^{12} \wedge \hat{Q}_{32}^{21}\right) + \frac{1}{2} X_1^{+} ** \left(\hat{Q}_{21}^{22} \wedge \hat{Q}_{32}^{22}\right) + \frac{1}{2} Q_{21}^{11} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{21}\right) -$$
$$\frac{1}{2} Q_{21}^{12} ** \left(\hat{Q}_{21}^{12} \wedge \hat{A}_{31}^{21}\right) + Q_{21}^{12} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{22}\right) - \frac{1}{2} X_2^{+} ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{22}\right) + \frac{1}{2} Q_{21}^{21} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{21}\right) -$$
$$\frac{1}{2} Q_{21}^{22} ** \left(\hat{Q}_{21}^{22} \wedge \hat{A}_{31}^{21}\right) + Q_{21}^{22} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{22}\right) - \frac{1}{2} X_2^{-} ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{32}^{21}\right) + \frac{1}{2} A_{31}^{12} ** \left(\hat{A}_{31}^{12} \wedge \hat{A}_{31}^{21}\right) -$$
$$\frac{1}{2} A_{31}^{12} ** \left(\hat{A}_{31}^{11} \wedge \hat{A}_{31}^{22}\right) - \frac{1}{2} X_3^{+} ** \left(\hat{Q}_{32}^{12} \wedge \hat{Q}_{21}^{21}\right) - \frac{1}{2} X_3^{+} ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{11}\right) + A_{31}^{22} ** \left(\hat{A}_{31}^{22} \wedge \hat{A}_{31}^{21}\right)$$

To avoid fractions, repeat `chBook` with coefficient 4:

chBook[c[3] → 4]

$$H_1 ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{21}\right) + H_1 ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{21}^{21}\right) - H_2 ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{21}\right) + H_2 ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{21}^{21}\right) + H_3 ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{21}\right) +$$
$$H_3 ** \left(\hat{Q}_{32}^{22} \wedge \hat{Q}_{21}^{21}\right) + 2 X_1^{+} ** \left(\hat{Q}_{21}^{12} \wedge \hat{Q}_{32}^{21}\right) + 2 X_1^{+} ** \left(\hat{Q}_{21}^{22} \wedge \hat{Q}_{32}^{22}\right) + 2 Q_{21}^{11} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{21}\right) -$$
$$2 Q_{21}^{12} ** \left(\hat{Q}_{21}^{12} \wedge \hat{A}_{31}^{21}\right) + 4 Q_{21}^{12} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{22}\right) - 2 X_2^{+} ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{22}\right) + 2 Q_{21}^{21} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{21}\right) -$$
$$2 Q_{21}^{22} ** \left(\hat{Q}_{21}^{22} \wedge \hat{A}_{31}^{21}\right) + 4 Q_{21}^{22} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{22}\right) - 2 X_2^{-} ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{32}^{21}\right) + 2 A_{31}^{12} ** \left(\hat{A}_{31}^{12} \wedge \hat{A}_{31}^{21}\right) -$$
$$2 A_{31}^{12} ** \left(\hat{A}_{31}^{11} \wedge \hat{A}_{31}^{22}\right) - 2 X_3^{+} ** \left(\hat{Q}_{32}^{12} \wedge \hat{Q}_{21}^{21}\right) - 2 X_3^{+} ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{32}^{11}\right) + 4 A_{31}^{22} ** \left(\hat{A}_{31}^{22} \wedge \hat{A}_{31}^{21}\right)$$

integ

$$(1 - c[1])\, e_1 ** dn_{12} - c[1]\, e_2 ** dn_{12} + (1 + c[1])\, e_3 ** dn_{12} + (2 + c[1])\, e_4 ** dn_{12} + (-1 - c[1])\, e_5 ** dn_{12} + (2 - c[1])\, e_{1,2} ** dn_{11} + c[1]\, e_{1,3} ** dn_8 + c[1]\, e_{1,4} ** dn_6 + (-2 - c[1])\, e_{3,6} ** dn_3 + (-2 - c[1])\, e_{4,6} ** dn_7 + c[1]\, e_{5,6} ** dn_{10}$$

ires[%, c[1] → 0]

$$H_1 ** \hat{A}_{31}^{21} + H_2 ** \hat{A}_{31}^{21} + 2\, H_{23} ** \hat{A}_{31}^{21} - H_3 ** \hat{A}_{31}^{21} + 2\, X_1^+ ** \hat{A}_{31}^{22} - 2\, Q_{23}^{12} ** \hat{Q}_{21}^{11} - 2\, Q_{23}^{22} ** \hat{Q}_{21}^{21}$$

next

$$\{0, \{1, 1, -1, -1, 1, -1\}\} \to - Q_{32}^{11} ** \left(\hat{Q}_{21}^{22} \wedge \hat{A}_{31}^{21}\right) + Q_{32}^{11} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{22}\right) + Q_{32}^{12} ** \left(\hat{Q}_{21}^{12} \wedge \hat{A}_{31}^{21}\right) - Q_{32}^{12} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{22}\right) - X_3^+ ** \left(\hat{Q}_{21}^{12} \wedge \hat{Q}_{21}^{21}\right) + X_3^+ ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{21}^{22}\right)$$

integ

0

next

$$\{0, \{1, 1, 0, -2, 0, 0\}\} \to - H_1 ** \left(\hat{Q}_{21}^{22} \wedge \hat{Q}_{21}^{21}\right) - X_1^+ ** \left(\hat{Q}_{21}^{22} \wedge \hat{Q}_{21}^{22}\right) + X_1^- ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{21}^{21}\right) + X_2^+ ** \left(\hat{Q}_{21}^{12} \wedge \hat{Q}_{21}^{21}\right) - X_2^+ ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{21}^{22}\right) - Q_{32}^{12} ** \left(\hat{Q}_{21}^{22} \wedge \hat{A}_{31}^{11}\right) + Q_{32}^{12} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{12}\right) - Q_{32}^{22} ** \left(\hat{Q}_{21}^{22} \wedge \hat{A}_{31}^{21}\right) + Q_{32}^{22} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{22}\right)$$

integ

$$- e_{1,4} ** dn_4 + e_{2,4} ** dn_7$$

ires[%]

$$- Q_{12}^{12} ** \hat{Q}_{21}^{22} + Q_{12}^{22} ** \hat{Q}_{21}^{21}$$

next

$$\{0, \{2, 0, -1, -1, 0, 0\}\} \to H_2 ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{21}^{21}\right) + X_1^+ ** \left(\hat{Q}_{21}^{12} \wedge \hat{Q}_{21}^{21}\right) - X_1^+ ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{21}^{22}\right) - X_2^+ ** \left(\hat{Q}_{21}^{11} \wedge \hat{Q}_{21}^{11}\right) + X_2^- ** \left(\hat{Q}_{21}^{21} \wedge \hat{Q}_{21}^{21}\right) + Q_{32}^{11} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{11}\right) - Q_{32}^{12} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{11}\right) + Q_{32}^{21} ** \left(\hat{Q}_{21}^{21} \wedge \hat{A}_{31}^{21}\right) - Q_{32}^{22} ** \left(\hat{Q}_{21}^{11} \wedge \hat{A}_{31}^{21}\right)$$

integ

$$e_{1,3} ** dn_7 - e_{1,4} ** dn_3$$

ires[%]

$$Q_{12}^{11} ** \hat{Q}_{21}^{21} - Q_{12}^{12} ** \hat{Q}_{21}^{11}$$

next

No more cohomologies

Grade r=3

chCalc[3, 2]

Total: {0, 0, {0, 30}}

Grade r=4

chCalc[4, 2]

```
Total: {0, 0, {0, 10}}
```

□ Output to TEX file

```
OpenTeX[];

OutRes[];

CloseTeX[True];
```

■ 3.3 Coefficients in the adjoint module

□ 3.3.1 osp(4|2;α)

Not written yet

■ 4. The Shapovalov determinants

Not written yet

■ 5. Decomposition of the tensor square

■ 5.1. Algebra sl(1|1)

```
Needs["SuperLie`"];
slAlgebra[g, Dim → {1, 1}]
```

```
SuperLie Package Version 2.03 installed
Disclaimer: This software is provided "AS IS", without a warranty of any kind

g = sl(1|1)
```

We will use operation NonCommutativeMultiply (defined in *Mathematica*) to denote the tensor product. In SuperLie, there is a shorter synonym to this operation, Tp. In order to define a module, we need the following properties of the tensor product:

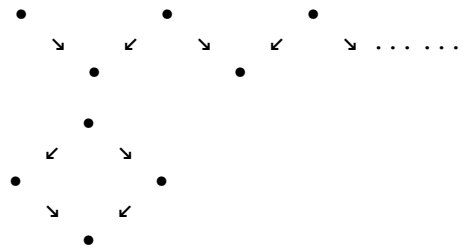
```
Jacobi[Act -> Tp];
Linear[Tp];
P[Tp] ^= 0;
```

Let us define the tensor product as a module M over algebra g:

```
Vector[M];
TheAlgebra[M] ^= g;
BasisPattern[M] ^= _Tp;
Basis[M] ^= Flatten[Outer[Tp, Basis[g], Basis[g]]]
```

```
{g1 ** g1, g1 ** g1,2, g1 ** g2,1, g1,2 ** g1,
 g1,2 ** g1,2, g1,2 ** g2,1, g2,1 ** g1, g2,1 ** g1,2, g2,1 ** g2,1}
```

Note that the central element g_1 acts on M by zero. All such finite–dimensional undecomposible $\mathfrak{sl}(1|1)$–modules are known, see [LInd]. There are two types of modules. One type may be described as a "snake" and the another one as a "square", see below. The bullets denote the elements of the basis and the arrows denotes all non–zero actions of $g_{1,2}$ (down and to the right) and $g_{2,1}$ (down and to the left). The "snake" can have any length and may begin with the arrow directed to the left as well:



Note that all elements in the lower row of the "snake" and "square" (and only they) are annihilated by $\mathfrak{sl}(1|1)$. Let us find them:

```
GeneralZero[Basis[g], Basis[M], c]
```

$$c[1]\, g_1 ** g_1 - c[2]\, g_1 ** g_{1,2} - c[3]\, g_1 ** g_{2,1} + c[2]\, g_{1,2} ** g_1 + c[3]\, g_{2,1} ** g_1$$

```
lv = GeneralBasis[%, c]
```

$$\{g_1 ** g_1,\ -g_1 ** g_{1,2} + g_{1,2} ** g_1,\ -g_1 ** g_{2,1} + g_{2,1} ** g_1\}$$

Now find the elements "glued" to these. We just calculate the action of g on the basis of M. Here is the action of $g_{1,2}$

```
# → VNormal[Act[g1,2, #]] & /@ Basis[M] // ColumnForm
```

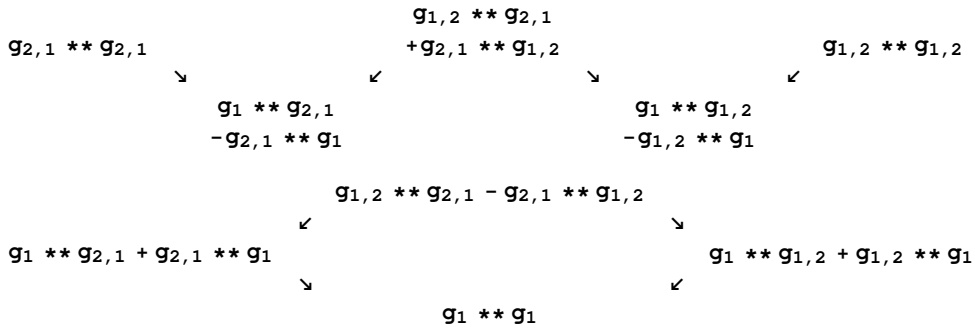
$$\begin{aligned}
&g_1 ** g_1 \to 0\\
&g_1 ** g_{1,2} \to 0\\
&g_1 ** g_{2,1} \to g_1 ** g_1\\
&g_{1,2} ** g_1 \to 0\\
&g_{1,2} ** g_{1,2} \to 0\\
&g_{1,2} ** g_{2,1} \to -g_{1,2} ** g_1\\
&g_{2,1} ** g_1 \to g_1 ** g_1\\
&g_{2,1} ** g_{1,2} \to g_1 ** g_{1,2}\\
&g_{2,1} ** g_{2,1} \to g_1 ** g_{2,1} - g_{2,1} ** g_1
\end{aligned}$$

and here is the action of $g_{2,1}$:

```
# → VNormal[Act[g2,1, #]] & /@ Basis[M] // ColumnForm
```

$$\begin{aligned}
&g_1 ** g_1 \to 0\\
&g_1 ** g_{1,2} \to g_1 ** g_1\\
&g_1 ** g_{2,1} \to 0\\
&g_{1,2} ** g_1 \to g_1 ** g_1\\
&g_{1,2} ** g_{1,2} \to g_1 ** g_{1,2} - g_{1,2} ** g_1\\
&g_{1,2} ** g_{2,1} \to g_1 ** g_{2,1}\\
&g_{2,1} ** g_1 \to 0\\
&g_{2,1} ** g_{1,2} \to -g_{2,1} ** g_1\\
&g_{2,1} ** g_{2,1} \to 0
\end{aligned}$$

From this information we can easily restore the structure of undecomposable submodules:



■ 6. Embeddings in diff

Not written yet

■ 7. Casimir elements

■ 7.1. The Casimir element for $k^L(1 \mid 6)$

The space and the algebra:

```
Needs["SuperLie`"]
Off[Solve::svars]

VectorSpace[ζ, Dim → (0 | 3)];
VectorSpace[η, Dim → (0 | 3)];
Symmetric[VTimes];
ContactAlgebra[g, {ζ, η}, t]
```

```
g is a Contact algebra over {ζ, η} and t
```

Now we define the weight and grading functions that respect the multiplication of polynomials. These functions do not respect the contact bracket (the function Weight–{2,0,0,0} does respect it), but this does not matter: we will use the weight only to order monomials.

```
Weight[ζi_] ^:= WeightMark[4, 1, i + 1]
Weight[ηi_] ^:= WeightMark[4, 1, -i - 1]
Weight[t] ^= {2, 0, 0, 0};
Weight[VTimes[]] ^= {0, 0, 0, 0};
Grade = Weight[#][[1]] &;
```

Now we should list all monomials of degree $2p + 1$ and $2p + 2$ for an indefinite integer p:

```
Scalar[p];
bas = Join[Basis[ζ], Basis[η]];
pComp = Join[{t^(p+1)}, t^p bas, t^p DegreeBasis[2, bas], t^(p-1) DegreeBasis[3, bas],
  t^(p-1) DegreeBasis[4, bas], t^(p-2) DegreeBasis[5, bas], t^(p-2) DegreeBasis[6, bas]]
```

```
{t^(1+p), t^p ζ1, t^p ζ2, t^p ζ3, t^p η1, t^p η2, t^p η3, t^p ζ1 ζ2, t^p ζ1 ζ3, t^p ζ1 η1, t^p ζ1 η2, t^p ζ1 η3,
 t^p ζ2 ζ3, t^p ζ2 η1, t^p ζ2 η2, t^p ζ2 η3, t^p ζ3 η1, t^p ζ3 η2, t^p ζ3 η3, t^p η1 η2, t^p η1 η3, t^p η2 η3,
 t^(-1+p) ζ1 ζ2 ζ3, t^(-1+p) ζ1 ζ2 η1, t^(-1+p) ζ1 ζ2 η2, t^(-1+p) ζ1 ζ2 η3, t^(-1+p) ζ1 ζ3 η1, t^(-1+p) ζ1 ζ3 η2,
 t^(-1+p) ζ1 ζ3 η3, t^(-1+p) ζ1 η1 η2, t^(-1+p) ζ1 η1 η3, t^(-1+p) ζ1 η2 η3, t^(-1+p) ζ2 ζ3 η1, t^(-1+p) ζ2 ζ3 η2,
 t^(-1+p) ζ2 ζ3 η3, t^(-1+p) ζ2 η1 η2, t^(-1+p) ζ2 η1 η3, t^(-1+p) ζ2 η2 η3, t^(-1+p) ζ3 η1 η2, t^(-1+p) ζ3 η1 η3,
 t^(-1+p) ζ3 η2 η3, t^(-1+p) η1 η2 η3, t^(-1+p) ζ1 ζ2 ζ3 η1, t^(-1+p) ζ1 ζ2 ζ3 η2, t^(-1+p) ζ1 ζ2 ζ3 η3,
 t^(-1+p) ζ1 ζ2 η1 η2, t^(-1+p) ζ1 ζ2 η1 η3, t^(-1+p) ζ1 ζ2 η2 η3, t^(-1+p) ζ1 ζ3 η1 η2, t^(-1+p) ζ1 ζ3 η1 η3,
 t^(-1+p) ζ1 ζ3 η2 η3, t^(-1+p) ζ1 η1 η2 η3, t^(-1+p) ζ2 ζ3 η1 η2, t^(-1+p) ζ2 ζ3 η1 η3, t^(-1+p) ζ2 ζ3 η2 η3,
 t^(-1+p) ζ2 η1 η2 η3, t^(-1+p) ζ3 η1 η2 η3, t^(-2+p) ζ1 ζ2 ζ3 η1 η2, t^(-2+p) ζ1 ζ2 ζ3 η1 η3, t^(-2+p) ζ1 ζ2 ζ3 η2 η3,
 t^(-2+p) ζ1 ζ2 η1 η2 η3, t^(-2+p) ζ1 ζ3 η1 η2 η3, t^(-2+p) ζ2 ζ3 η1 η2 η3, t^(-2+p) ζ1 ζ2 ζ3 η1 η2 η3}
```

The Casimir element is defined to be the formal sum $\sum_{g_i \in \mathfrak{g}} g_i^* g_i$, where $\{g_i\}$ is a basis in $\mathfrak{g}$.

The pairing $\langle x, y \rangle$ in $\mathfrak{g}$ is defined as coefficient of $t^{-1}\ \zeta_1\ \zeta_2\ \zeta_3\ \eta_1\ \eta_2\ \eta_3$ in xy.
Let us write the dual elements to `pComp` (up to a sign):

```
pDual = t^-1 ζ1 ζ2 ζ3 η1 η2 η3 (pComp /. v : (t | _ζ | _η) → v^-1)
```

```
{t^(-2-p) ζ1 ζ2 ζ3 η1 η2 η3, -t^(-1-p) ζ2 ζ3 η1 η2 η3, t^(-1-p) ζ1 ζ3 η1 η2 η3, -t^(-1-p) ζ1 ζ2 η1 η2 η3,
 t^(-1-p) ζ1 ζ2 ζ3 η2 η3, -t^(-1-p) ζ1 ζ2 ζ3 η1 η3, t^(-1-p) ζ1 ζ2 ζ3 η1 η2, -t^(-1-p) ζ3 η1 η2 η3,
 t^(-1-p) ζ2 η1 η2 η3, -t^(-1-p) ζ2 ζ3 η2 η3, t^(-1-p) ζ2 ζ3 η1 η3, -t^(-1-p) ζ2 ζ3 η1 η2, -t^(-1-p) ζ1 η1 η2 η3,
 t^(-1-p) ζ1 ζ3 η2 η3, -t^(-1-p) ζ1 ζ3 η1 η3, t^(-1-p) ζ1 ζ3 η1 η2, -t^(-1-p) ζ1 ζ2 η2 η3, t^(-1-p) ζ1 ζ2 η1 η3,
 -t^(-1-p) ζ1 ζ2 η1 η2, -t^(-1-p) ζ1 ζ2 ζ3 η3, t^(-1-p) ζ1 ζ2 ζ3 η2, -t^(-1-p) ζ1 ζ2 ζ3 η1, t^-p η1 η2 η3,
 -t^-p ζ3 η2 η3, t^-p ζ3 η1 η3, -t^-p ζ3 η1 η2, t^-p ζ2 η2 η3, -t^-p ζ2 η1 η3, t^-p ζ2 η1 η2,
 t^-p ζ2 ζ3 η3, -t^-p ζ2 ζ3 η2, t^-p ζ2 ζ3 η1, -t^-p ζ1 η2 η3, t^-p ζ1 η1 η3, -t^-p ζ1 η1 η2,
 -t^-p ζ1 ζ3 η3, t^-p ζ1 ζ3 η2, -t^-p ζ1 ζ3 η1, t^-p ζ1 ζ2 η3, -t^-p ζ1 ζ2 η2, t^-p ζ1 ζ2 η1,
 -t^-p ζ1 ζ2 ζ3, t^-p η2 η3, -t^-p η1 η3, t^-p η1 η2, t^-p ζ3 η3, -t^-p ζ3 η2, t^-p ζ3 η1,
 -t^-p ζ2 η3, t^-p ζ2 η2, -t^-p ζ2 η1, t^-p ζ2 ζ3, t^-p ζ1 η3, -t^-p ζ1 η2, t^-p ζ1 η1,
 -t^-p ζ1 ζ3, t^-p ζ1 ζ2, -t^(1-p) η3, t^(1-p) η2, -t^(1-p) η1, t^(1-p) ζ3, -t^(1-p) ζ2, t^(1-p) ζ1, -t^(1-p)}
```

Now we fix the sign:

```
sgn = pDual * pComp /. SVTimes[-1, _] → -1 /. v_VTimes → 1
```

```
{1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1}
```

```
pDual = sgn * pDual
```

```
{t^(-2-p) ζ1 ζ2 ζ3 η1 η2 η3, -t^(-1-p) ζ2 ζ3 η1 η2 η3, t^(-1-p) ζ1 ζ3 η1 η2 η3, -t^(-1-p) ζ1 ζ2 η1 η2 η3,
 t^(-1-p) ζ1 ζ2 ζ3 η2 η3, -t^(-1-p) ζ1 ζ2 ζ3 η1 η3, t^(-1-p) ζ1 ζ2 ζ3 η1 η2, t^(-1-p) ζ3 η1 η2 η3,
 -t^(-1-p) ζ2 η1 η2 η3, t^(-1-p) ζ2 ζ3 η2 η3, -t^(-1-p) ζ2 ζ3 η1 η3, t^(-1-p) ζ2 ζ3 η1 η2, t^(-1-p) ζ1 η1 η2 η3,
 -t^(-1-p) ζ1 ζ3 η2 η3, t^(-1-p) ζ1 ζ3 η1 η3, -t^(-1-p) ζ1 ζ3 η1 η2, t^(-1-p) ζ1 ζ2 η2 η3, -t^(-1-p) ζ1 ζ2 η1 η3,
 t^(-1-p) ζ1 ζ2 η1 η2, t^(-1-p) ζ1 ζ2 ζ3 η3, -t^(-1-p) ζ1 ζ2 ζ3 η2, t^(-1-p) ζ1 ζ2 ζ3 η1, -t^-p η1 η2 η3,
 t^-p ζ3 η2 η3, -t^-p ζ3 η1 η3, t^-p ζ3 η1 η2, -t^-p ζ2 η2 η3, t^-p ζ2 η1 η3, -t^-p ζ2 η1 η2,
 -t^-p ζ2 ζ3 η3, t^-p ζ2 ζ3 η2, -t^-p ζ2 ζ3 η1, t^-p ζ1 η2 η3, -t^-p ζ1 η1 η3, t^-p ζ1 η1 η2,
 t^-p ζ1 ζ3 η3, -t^-p ζ1 ζ3 η2, t^-p ζ1 ζ3 η1, -t^-p ζ1 ζ2 η3, t^-p ζ1 ζ2 η2, -t^-p ζ1 ζ2 η1,
 t^-p ζ1 ζ2 ζ3, t^-p η2 η3, -t^-p η1 η3, t^-p η1 η2, t^-p ζ3 η3, -t^-p ζ3 η2, t^-p ζ3 η1,
 -t^-p ζ2 η3, t^-p ζ2 η2, -t^-p ζ2 η1, t^-p ζ2 ζ3, t^-p ζ1 η3, -t^-p ζ1 η2, t^-p ζ1 η1,
 -t^-p ζ1 ζ3, t^-p ζ1 ζ2, -t^(1-p) η3, t^(1-p) η2, -t^(1-p) η1, t^(1-p) ζ3, -t^(1-p) ζ2, t^(1-p) ζ1, t^(1-p)}
```

Now we express the Casimir element c_2 as an element of the completed enveloping algebra $\hat{U}(\mathfrak{g})$.

The operation VTimes is already used (as multiplication of polynomials), so we should define another operation in the enveloping algebra. Let it be `ep`. We will not define "power" operation to represent ep[x,x,...], so we write None instead of "power" name.

The terms of `ep` should be sorted in some fixed order. Let us write the elements with higher weight first. To do this, define the ordering function which compares, first, the weights (in decreas-

$$p = 10$$

$$p - 1 > 1 - p$$

ep

ing order) and next, if the weights are equal the elements themselves in the alphabetical order.

It turned out that with indefinite p, the comparison does not work properly, so when comparing the weights depending on p, we will temporarily (only for this comparison) set $p = 10$ (to be sure that, e.g., $p - 1 > 1 - p$).

```
envOrd[f_, g_] := OrderedQ[{{-Weight[f], f} /. p → 10, {-Weight[g], g} /. p → 10}];
EnvelopingOperation[ep, None, Kb, envOrd]
Jacobi[Kb → ep]
```

```
{JacobiRule[Kb, CircleTimes], JacobiRule[Kb, ep], LinearRule[Kb]}
```

The Casimir element is the formal sum of all $\mathtt{ep}[\mathtt{g}_i, \mathtt{g}_i^*]$ (where $\mathtt{g}_i$ are the elements of the basis of $\mathfrak{g}$ such that $\mathtt{g}_i$ with $\mathtt{g}_i^*$ are in the above–defined order) plus a linear term $\sum \mathtt{a}_j \mathtt{g}_j$, where all the $\mathtt{g}_j$ have zero weight and the scalar coefficients $\mathtt{a}_i$ are unknown yet.

Here is the term of the Casimir element corresponding to pComp:

```
pCas = Inner[ep, pComp, pDual, VPlus]
```

$$\begin{aligned}
&\mathtt{ep}[t^{1+p}, t^{-2-p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - \mathtt{ep}[t^p\,\xi_1, t^{-1-p}\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3] + \\
&\mathtt{ep}[t^p\,\xi_2, t^{-1-p}\,\xi_1\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - \mathtt{ep}[t^p\,\xi_3, t^{-1-p}\,\xi_1\,\xi_2\,\eta_1\,\eta_2\,\eta_3] + \mathtt{ep}[t^p\,\eta_1, t^{-1-p}\,\xi_1\,\xi_2\,\xi_3\,\eta_2\,\eta_3] - \\
&\mathtt{ep}[t^p\,\eta_2, t^{-1-p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_3] + \mathtt{ep}[t^p\,\eta_3, t^{-1-p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2] + \mathtt{ep}[t^p\,\xi_1\,\xi_2, t^{-1-p}\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - \\
&\mathtt{ep}[t^p\,\xi_1\,\xi_3, t^{-1-p}\,\xi_2\,\eta_1\,\eta_2\,\eta_3] + \mathtt{ep}[t^p\,\xi_1\,\eta_1, t^{-1-p}\,\xi_2\,\xi_3\,\eta_2\,\eta_3] - \mathtt{ep}[t^p\,\xi_1\,\eta_2, t^{-1-p}\,\xi_2\,\xi_3\,\eta_1\,\eta_3] + \\
&\mathtt{ep}[t^p\,\xi_1\,\eta_3, t^{-1-p}\,\xi_2\,\xi_3\,\eta_1\,\eta_2] + \mathtt{ep}[t^p\,\xi_2\,\xi_3, t^{-1-p}\,\xi_1\,\eta_1\,\eta_2\,\eta_3] - \mathtt{ep}[t^p\,\xi_2\,\eta_1, t^{-1-p}\,\xi_1\,\xi_3\,\eta_2\,\eta_3] + \\
&\mathtt{ep}[t^p\,\xi_2\,\eta_2, t^{-1-p}\,\xi_1\,\xi_3\,\eta_1\,\eta_3] - \mathtt{ep}[t^p\,\xi_2\,\eta_3, t^{-1-p}\,\xi_1\,\xi_3\,\eta_1\,\eta_2] + \mathtt{ep}[t^p\,\xi_3\,\eta_1, t^{-1-p}\,\xi_1\,\xi_2\,\eta_2\,\eta_3] - \\
&\mathtt{ep}[t^p\,\xi_3\,\eta_2, t^{-1-p}\,\xi_1\,\xi_2\,\eta_1\,\eta_3] + \mathtt{ep}[t^p\,\xi_3\,\eta_3, t^{-1-p}\,\xi_1\,\xi_2\,\eta_1\,\eta_2] + \mathtt{ep}[t^p\,\eta_1\,\eta_2, t^{-1-p}\,\xi_1\,\xi_2\,\xi_3\,\eta_3] - \\
&\mathtt{ep}[t^p\,\eta_1\,\eta_3, t^{-1-p}\,\xi_1\,\xi_2\,\xi_3\,\eta_2] + \mathtt{ep}[t^p\,\eta_2\,\eta_3, t^{-1-p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1] - \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\xi_3, t^{-p}\,\eta_1\,\eta_2\,\eta_3] + \\
&\mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\eta_1, t^{-p}\,\xi_3\,\eta_2\,\eta_3] - \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\eta_2, t^{-p}\,\xi_3\,\eta_1\,\eta_3] + \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\eta_3, t^{-p}\,\xi_3\,\eta_1\,\eta_2] - \\
&\mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_3\,\eta_1, t^{-p}\,\xi_2\,\eta_2\,\eta_3] + \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_3\,\eta_2, t^{-p}\,\xi_2\,\eta_1\,\eta_3] - \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_3\,\eta_3, t^{-p}\,\xi_2\,\eta_1\,\eta_2] - \\
&\mathtt{ep}[t^{-1+p}\,\xi_1\,\eta_1\,\eta_2, t^{-p}\,\xi_2\,\xi_3\,\eta_3] + \mathtt{ep}[t^{-1+p}\,\xi_1\,\eta_1\,\eta_3, t^{-p}\,\xi_2\,\xi_3\,\eta_2] - \mathtt{ep}[t^{-1+p}\,\xi_1\,\eta_2\,\eta_3, t^{-p}\,\xi_2\,\xi_3\,\eta_1] + \\
&\mathtt{ep}[t^{-1+p}\,\xi_2\,\xi_3\,\eta_1, t^{-p}\,\xi_1\,\eta_2\,\eta_3] - \mathtt{ep}[t^{-1+p}\,\xi_2\,\xi_3\,\eta_2, t^{-p}\,\xi_1\,\eta_1\,\eta_3] + \mathtt{ep}[t^{-1+p}\,\xi_2\,\xi_3\,\eta_3, t^{-p}\,\xi_1\,\eta_1\,\eta_2] + \\
&\mathtt{ep}[t^{-1+p}\,\xi_2\,\eta_1\,\eta_2, t^{-p}\,\xi_1\,\xi_3\,\eta_3] - \mathtt{ep}[t^{-1+p}\,\xi_2\,\eta_1\,\eta_3, t^{-p}\,\xi_1\,\xi_3\,\eta_2] + \mathtt{ep}[t^{-1+p}\,\xi_2\,\eta_2\,\eta_3, t^{-p}\,\xi_1\,\xi_3\,\eta_1] - \\
&\mathtt{ep}[t^{-1+p}\,\xi_3\,\eta_1\,\eta_2, t^{-p}\,\xi_1\,\xi_2\,\eta_3] + \mathtt{ep}[t^{-1+p}\,\xi_3\,\eta_1\,\eta_3, t^{-p}\,\xi_1\,\xi_2\,\eta_2] - \mathtt{ep}[t^{-1+p}\,\xi_3\,\eta_2\,\eta_3, t^{-p}\,\xi_1\,\xi_2\,\eta_1] + \\
&\mathtt{ep}[t^{-1+p}\,\eta_1\,\eta_2\,\eta_3, t^{-p}\,\xi_1\,\xi_2\,\xi_3] + \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1, t^{-p}\,\eta_2\,\eta_3] - \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\xi_3\,\eta_2, t^{-p}\,\eta_1\,\eta_3] + \\
&\mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\xi_3\,\eta_3, t^{-p}\,\eta_1\,\eta_2] + \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\eta_1\,\eta_2, t^{-p}\,\xi_3\,\eta_3] - \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\eta_1\,\eta_3, t^{-p}\,\xi_3\,\eta_2] + \\
&\mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_2\,\eta_2\,\eta_3, t^{-p}\,\xi_3\,\eta_1] - \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_3\,\eta_1\,\eta_2, t^{-p}\,\xi_2\,\eta_3] + \mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_3\,\eta_1\,\eta_3, t^{-p}\,\xi_2\,\eta_2] - \\
&\mathtt{ep}[t^{-1+p}\,\xi_1\,\xi_3\,\eta_2\,\eta_3, t^{-p}\,\xi_2\,\eta_1] + \mathtt{ep}[t^{-1+p}\,\xi_1\,\eta_1\,\eta_2\,\eta_3, t^{-p}\,\xi_2\,\xi_3] + \mathtt{ep}[t^{-1+p}\,\xi_2\,\xi_3\,\eta_1\,\eta_2, t^{-p}\,\xi_1\,\eta_3] - \\
&\mathtt{ep}[t^{-1+p}\,\xi_2\,\xi_3\,\eta_1\,\eta_3, t^{-p}\,\xi_1\,\eta_2] + \mathtt{ep}[t^{-1+p}\,\xi_2\,\xi_3\,\eta_2\,\eta_3, t^{-p}\,\xi_1\,\eta_1] - \mathtt{ep}[t^{-1+p}\,\xi_2\,\eta_1\,\eta_2\,\eta_3, t^{-p}\,\xi_1\,\xi_3] + \\
&\mathtt{ep}[t^{-1+p}\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{-p}\,\xi_1\,\xi_2] - \mathtt{ep}[t^{-2+p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2, t^{1-p}\,\eta_3] + \\
&\mathtt{ep}[t^{-2+p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_3, t^{1-p}\,\eta_2] - \mathtt{ep}[t^{-2+p}\,\xi_1\,\xi_2\,\xi_3\,\eta_2\,\eta_3, t^{1-p}\,\eta_1] + \\
&\mathtt{ep}[t^{-2+p}\,\xi_1\,\xi_2\,\eta_1\,\eta_2\,\eta_3, t^{1-p}\,\xi_3] - \mathtt{ep}[t^{-2+p}\,\xi_1\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{1-p}\,\xi_2] + \\
&\mathtt{ep}[t^{-2+p}\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{1-p}\,\xi_1] + \mathtt{ep}[t^{-2+p}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{1-p}]
\end{aligned}$$

Let us check that $\mathtt{Kb}[\mathfrak{g}, \sum_p \mathtt{pCas}_p] = 0$. It suffices to check it for generators of $\mathfrak{k}(1|6)$.

```
Kb[ξ1 η2, pCas] // VNormal
```

```
0
```

```
Kb[ξ2 η3, pCas] // VNormal
```

```
0
```

```
Kb[ξ2 ξ3, pCas] // VNormal
```

```
0
```

```
res = Kb[η3 η2 η1, pCas] // VNormal
```

```
- ep[t^(-1+p) ζ1 η1 η2, t^-p ζ2 η1 η2 η3] - ep[t^(-1+p) ζ1 η1 η3, t^-p ζ3 η1 η2 η3] +
 ep[t^(-1+p) ζ2 η1 η2, t^-p ζ1 η1 η2 η3] - ep[t^(-1+p) ζ2 η2 η3, t^-p ζ3 η1 η2 η3] +
 ep[t^(-1+p) ζ3 η1 η3, t^-p ζ1 η1 η2 η3] + ep[t^(-1+p) ζ3 η2 η3, t^-p ζ2 η1 η2 η3] -
 ep[t^(-1+p) η1 η2 η3, t^-p ζ1 ζ2 η1 η2] - ep[t^(-1+p) η1 η2 η3, t^-p ζ1 ζ3 η1 η3] -
 ep[t^(-1+p) η1 η2 η3, t^-p ζ2 ζ3 η2 η3] - p ep[t^(-1+p) η1 η2 η3, t^(-1-p) ζ1 ζ2 ζ3 η1 η2 η3] +
 ep[t^p ζ1 η1 η2, t^(-1-p) ζ2 η1 η2 η3] + ep[t^p ζ1 η1 η3, t^(-1-p) ζ3 η1 η2 η3] -
 ep[t^p ζ2 η1 η2, t^(-1-p) ζ1 η1 η2 η3] + ep[t^p ζ2 η2 η3, t^(-1-p) ζ3 η1 η2 η3] -
 ep[t^p ζ3 η1 η3, t^(-1-p) ζ1 η1 η2 η3] - ep[t^p ζ3 η2 η3, t^(-1-p) ζ2 η1 η2 η3] +
 ep[t^p η1 η2 η3, t^(-1-p) ζ1 ζ2 η1 η2] + ep[t^p η1 η2 η3, t^(-1-p) ζ1 ζ3 η1 η3] +
 ep[t^p η1 η2 η3, t^(-1-p) ζ2 ζ3 η2 η3] + (1 + p) ep[t^p η1 η2 η3, t^(-2-p) ζ1 ζ2 ζ3 η1 η2 η3] -
 ep[t^(-2+p) ζ1 ζ2 η1 η2 η3, t^(1-p) η1 η2] + (1 - p) ep[t^(-2+p) ζ1 ζ2 η1 η2 η3, t^-p ζ3 η1 η2 η3] -
 ep[t^(-2+p) ζ1 ζ3 η1 η2 η3, t^(1-p) η1 η3] + (-1 + p) ep[t^(-2+p) ζ1 ζ3 η1 η2 η3, t^-p ζ2 η1 η2 η3] -
 ep[t^(-2+p) ζ2 ζ3 η1 η2 η3, t^(1-p) η2 η3] + (1 - p) ep[t^(-2+p) ζ2 ζ3 η1 η2 η3, t^-p ζ1 η1 η2 η3] +
 ep[t^(-1+p) ζ1 ζ2 η1 η2 η3, t^-p η1 η2] + p ep[t^(-1+p) ζ1 ζ2 η1 η2 η3, t^(-1-p) ζ3 η1 η2 η3] +
 ep[t^(-1+p) ζ1 ζ3 η1 η2 η3, t^-p η1 η3] - p ep[t^(-1+p) ζ1 ζ3 η1 η2 η3, t^(-1-p) ζ2 η1 η2 η3] +
 ep[t^(-1+p) ζ2 ζ3 η1 η2 η3, t^-p η2 η3] + p ep[t^(-1+p) ζ2 ζ3 η1 η2 η3, t^(-1-p) ζ1 η1 η2 η3]
```

This should annihilate with terms corresponding to a neighboring p. To check this, we split the sum into the sum of components `ep[x, y]` with different grades of `x`:

```
splitRes = SplitSum[res, _ep, Expand[Grade[#[[1]]] &]
```

```
{1 + 2 p → - ep[t^(-1+p) ζ1 η1 η2, t^-p ζ2 η1 η2 η3] - ep[t^(-1+p) ζ1 η1 η3, t^-p ζ3 η1 η2 η3] +
    ep[t^(-1+p) ζ2 η1 η2, t^-p ζ1 η1 η2 η3] - ep[t^(-1+p) ζ2 η2 η3, t^-p ζ3 η1 η2 η3] +
    ep[t^(-1+p) ζ3 η1 η3, t^-p ζ1 η1 η2 η3] + ep[t^(-1+p) ζ3 η2 η3, t^-p ζ2 η1 η2 η3] -
    ep[t^(-1+p) η1 η2 η3, t^-p ζ1 ζ2 η1 η2] - ep[t^(-1+p) η1 η2 η3, t^-p ζ1 ζ3 η1 η3] -
    ep[t^(-1+p) η1 η2 η3, t^-p ζ2 ζ3 η2 η3] - p ep[t^(-1+p) η1 η2 η3, t^(-1-p) ζ1 ζ2 ζ3 η1 η2 η3] -
    ep[t^(-2+p) ζ1 ζ2 η1 η2 η3, t^(1-p) η1 η2] + (1 - p) ep[t^(-2+p) ζ1 ζ2 η1 η2 η3, t^-p ζ3 η1 η2 η3] -
    ep[t^(-2+p) ζ1 ζ3 η1 η2 η3, t^(1-p) η1 η3] + (-1 + p) ep[t^(-2+p) ζ1 ζ3 η1 η2 η3, t^-p ζ2 η1 η2 η3] -
    ep[t^(-2+p) ζ2 ζ3 η1 η2 η3, t^(1-p) η2 η3] + (1 - p) ep[t^(-2+p) ζ2 ζ3 η1 η2 η3, t^-p ζ1 η1 η2 η3],
 3 + 2 p → ep[t^p ζ1 η1 η2, t^(-1-p) ζ2 η1 η2 η3] + ep[t^p ζ1 η1 η3, t^(-1-p) ζ3 η1 η2 η3] -
    ep[t^p ζ2 η1 η2, t^(-1-p) ζ1 η1 η2 η3] + ep[t^p ζ2 η2 η3, t^(-1-p) ζ3 η1 η2 η3] -
    ep[t^p ζ3 η1 η3, t^(-1-p) ζ1 η1 η2 η3] - ep[t^p ζ3 η2 η3, t^(-1-p) ζ2 η1 η2 η3] +
    ep[t^p η1 η2 η3, t^(-1-p) ζ1 ζ2 η1 η2] + ep[t^p η1 η2 η3, t^(-1-p) ζ1 ζ3 η1 η3] +
    ep[t^p η1 η2 η3, t^(-1-p) ζ2 ζ3 η2 η3] + (1 + p) ep[t^p η1 η2 η3, t^(-2-p) ζ1 ζ2 ζ3 η1 η2 η3] +
    ep[t^(-1+p) ζ1 ζ2 η1 η2 η3, t^-p η1 η2] + p ep[t^(-1+p) ζ1 ζ2 η1 η2 η3, t^(-1-p) ζ3 η1 η2 η3] +
    ep[t^(-1+p) ζ1 ζ3 η1 η2 η3, t^-p η1 η3] - p ep[t^(-1+p) ζ1 ζ3 η1 η2 η3, t^(-1-p) ζ2 η1 η2 η3] +
    ep[t^(-1+p) ζ2 ζ3 η1 η2 η3, t^-p η2 η3] + p ep[t^(-1+p) ζ2 ζ3 η1 η2 η3, t^(-1-p) ζ1 η1 η2 η3]}
```

Now check that the two components annihilate with a neighboring term of the Casimir element

```
VNormal[PartSplit[splitRes, 1 + 2 p] + (PartSplit[splitRes, 3 + 2 p] /. p -> p - 1)]
```

```
0
```

Now repeat the same procedure for another positive generator, $t\eta_1$.

```
res = Kb[t η1, pCas] // VNormal;
splitRes = SplitSum[res, _ep, Expand[Grade[#[[1]]] &];
VNormal[PartSplit[splitRes, 1 + 2 p] + (PartSplit[splitRes, 3 + 2 p] /. p -> p - 1)]
```

```
0
```

We should also test the negative generators:

```
Kb[ζ2 η1, pCas] // VNormal
```

```
0
```

```
Kb[ζ3 η2, pCas] // VNormal
```

```
0
```

```
Kb[η2 η3, pCas] // VNormal
```

```
0
```

```
res = Kb[ζ1, pCas] // VNormal
```

```
- ep[t^p, t^(-1-p) ζ1 ζ2 ζ3 η2 η3] + ep[t^(1+p), t^(-2-p) ζ1 ζ2 ζ3 η2 η3] - ep[t^(-1+p) ζ1 ζ2, t^-p ζ3 η2 η3] +
 p ep[t^(-1+p) ζ1 ζ2, t^(-1-p) ζ1 ζ3 η1 η2 η3] + ep[t^(-1+p) ζ1 ζ3, t^-p ζ2 η2 η3] -
 p ep[t^(-1+p) ζ1 ζ3, t^(-1-p) ζ1 ζ2 η1 η2 η3] + p ep[t^(-1+p) ζ1 η1, t^(-1-p) ζ1 ζ2 ζ3 η2 η3] -
 ep[t^(-1+p) ζ1 η2, t^-p ζ2 ζ3 η3] - p ep[t^(-1+p) ζ1 η2, t^(-1-p) ζ1 ζ2 ζ3 η1 η3] +
 ep[t^(-1+p) ζ1 η3, t^-p ζ2 ζ3 η2] + p ep[t^(-1+p) ζ1 η3, t^(-1-p) ζ1 ζ2 ζ3 η1 η2] - ep[t^(-1+p) ζ2 ζ3, t^-p ζ1 η2 η3] +
 ep[t^(-1+p) ζ2 η2, t^-p ζ1 ζ3 η3] - ep[t^(-1+p) ζ2 η3, t^-p ζ1 ζ3 η2] - ep[t^(-1+p) ζ3 η2, t^-p ζ1 ζ2 η3] +
 ep[t^(-1+p) ζ3 η3, t^-p ζ1 ζ2 η2] - ep[t^(-1+p) η2 η3, t^-p ζ1 ζ2 ζ3] + ep[t^p ζ1 ζ2, t^(-1-p) ζ3 η2 η3] +
 (-1 - p) ep[t^p ζ1 ζ2, t^(-2-p) ζ1 ζ3 η1 η2 η3] - ep[t^p ζ1 ζ3, t^(-1-p) ζ2 η2 η3] +
 (1 + p) ep[t^p ζ1 ζ3, t^(-2-p) ζ1 ζ2 η1 η2 η3] + (-1 - p) ep[t^p ζ1 η1, t^(-2-p) ζ1 ζ2 ζ3 η2 η3] +
 ep[t^p ζ1 η2, t^(-1-p) ζ2 ζ3 η3] + (1 + p) ep[t^p ζ1 η2, t^(-2-p) ζ1 ζ2 ζ3 η1 η3] -
 ep[t^p ζ1 η3, t^(-1-p) ζ2 ζ3 η2] + (-1 - p) ep[t^p ζ1 η3, t^(-2-p) ζ1 ζ2 ζ3 η1 η2] +
 ep[t^p ζ2 ζ3, t^(-1-p) ζ1 η2 η3] - ep[t^p ζ2 η2, t^(-1-p) ζ1 ζ3 η3] + ep[t^p ζ2 η3, t^(-1-p) ζ1 ζ3 η2] +
 ep[t^p ζ3 η2, t^(-1-p) ζ1 ζ2 η3] - ep[t^p ζ3 η3, t^(-1-p) ζ1 ζ2 η2] + ep[t^p η2 η3, t^(-1-p) ζ1 ζ2 ζ3] +
 (-1 + p) ep[t^(-2+p) ζ1 ζ2 ζ3 η1, t^-p ζ1 η2 η3] - ep[t^(-2+p) ζ1 ζ2 ζ3 η2, t^(1-p) η3] +
 (1 - p) ep[t^(-2+p) ζ1 ζ2 ζ3 η2, t^-p ζ1 η1 η3] + ep[t^(-2+p) ζ1 ζ2 ζ3 η3, t^(1-p) η2] +
 (-1 + p) ep[t^(-2+p) ζ1 ζ2 ζ3 η3, t^-p ζ1 η1 η2] + (-1 + p) ep[t^(-2+p) ζ1 ζ2 η1 η2, t^-p ζ1 ζ3 η3] +
 (1 - p) ep[t^(-2+p) ζ1 ζ2 η1 η3, t^-p ζ1 ζ3 η2] - ep[t^(-2+p) ζ1 ζ2 η2 η3, t^(1-p) ζ3] +
 (-1 + p) ep[t^(-2+p) ζ1 ζ2 η2 η3, t^-p ζ1 ζ3 η1] + (1 - p) ep[t^(-2+p) ζ1 ζ3 η1 η2, t^-p ζ1 ζ2 η3] +
 (-1 + p) ep[t^(-2+p) ζ1 ζ3 η1 η3, t^-p ζ1 ζ2 η2] + ep[t^(-2+p) ζ1 ζ3 η2 η3, t^(1-p) ζ2] +
 (1 - p) ep[t^(-2+p) ζ1 ζ3 η2 η3, t^-p ζ1 ζ2 η1] + (-1 + p) ep[t^(-2+p) ζ1 η1 η2 η3, t^-p ζ1 ζ2 ζ3] -
 ep[t^(-2+p) ζ2 ζ3 η2 η3, t^(1-p) ζ1] - p ep[t^(-1+p) ζ1 ζ2 ζ3 η1, t^(-1-p) ζ1 η2 η3] +
 ep[t^(-1+p) ζ1 ζ2 ζ3 η2, t^-p η3] + p ep[t^(-1+p) ζ1 ζ2 ζ3 η2, t^(-1-p) ζ1 η1 η3] -
 ep[t^(-1+p) ζ1 ζ2 ζ3 η3, t^-p η2] - p ep[t^(-1+p) ζ1 ζ2 ζ3 η3, t^(-1-p) ζ1 η1 η2] -
 p ep[t^(-1+p) ζ1 ζ2 η1 η2, t^(-1-p) ζ1 ζ3 η3] + p ep[t^(-1+p) ζ1 ζ2 η1 η3, t^(-1-p) ζ1 ζ3 η2] +
 ep[t^(-1+p) ζ1 ζ2 η2 η3, t^-p ζ3] - p ep[t^(-1+p) ζ1 ζ2 η2 η3, t^(-1-p) ζ1 ζ3 η1] +
 p ep[t^(-1+p) ζ1 ζ3 η1 η2, t^(-1-p) ζ1 ζ2 η3] - p ep[t^(-1+p) ζ1 ζ3 η1 η3, t^(-1-p) ζ1 ζ2 η2] -
 ep[t^(-1+p) ζ1 ζ3 η2 η3, t^-p ζ2] + p ep[t^(-1+p) ζ1 ζ3 η2 η3, t^(-1-p) ζ1 ζ2 η1] -
 p ep[t^(-1+p) ζ1 η1 η2 η3, t^(-1-p) ζ1 ζ2 ζ3] + ep[t^(-1+p) ζ2 ζ3 η2 η3, t^-p ζ1] +
 (-2 + p) ep[t^(-3+p) ζ1 ζ2 ζ3 η1 η2 η3, t^(1-p) ζ1] + (1 - p) ep[t^(-2+p) ζ1 ζ2 ζ3 η1 η2 η3, t^-p ζ1]
```

```
splitRes = SplitSum[res, _ep, Expand[Grade[#[[1]]] &]
```

```
{2 p → - ep[t^p, t^(-1-p) ζ1 ζ2 ζ3 η2 η3] - ep[t^(-1+p) ζ1 ζ2, t^-p ζ3 η2 η3] +
    p ep[t^(-1+p) ζ1 ζ2, t^(-1-p) ζ1 ζ3 η1 η2 η3] + ep[t^(-1+p) ζ1 ζ3, t^-p ζ2 η2 η3] -
    p ep[t^(-1+p) ζ1 ζ3, t^(-1-p) ζ1 ζ2 η1 η2 η3] + p ep[t^(-1+p) ζ1 η1, t^(-1-p) ζ1 ζ2 ζ3 η2 η3] -
    ep[t^(-1+p) ζ1 η2, t^-p ζ2 ζ3 η3] - p ep[t^(-1+p) ζ1 η2, t^(-1-p) ζ1 ζ2 ζ3 η1 η3] +
    ep[t^(-1+p) ζ1 η3, t^-p ζ2 ζ3 η2] + p ep[t^(-1+p) ζ1 η3, t^(-1-p) ζ1 ζ2 ζ3 η1 η2] -
    ep[t^(-1+p) ζ2 ζ3, t^-p ζ1 η2 η3] + ep[t^(-1+p) ζ2 η2, t^-p ζ1 ζ3 η3] - ep[t^(-1+p) ζ2 η3, t^-p ζ1 ζ3 η2] -
    ep[t^(-1+p) ζ3 η2, t^-p ζ1 ζ2 η3] + ep[t^(-1+p) ζ3 η3, t^-p ζ1 ζ2 η2] - ep[t^(-1+p) η2 η3, t^-p ζ1 ζ2 ζ3] +
    (-1 + p) ep[t^(-2+p) ζ1 ζ2 ζ3 η1, t^-p ζ1 η2 η3] - ep[t^(-2+p) ζ1 ζ2 ζ3 η2, t^(1-p) η3] +
    (1 - p) ep[t^(-2+p) ζ1 ζ2 ζ3 η2, t^-p ζ1 η1 η3] + ep[t^(-2+p) ζ1 ζ2 ζ3 η3, t^(1-p) η2] +
    (-1 + p) ep[t^(-2+p) ζ1 ζ2 ζ3 η3, t^-p ζ1 η1 η2] + (-1 + p) ep[t^(-2+p) ζ1 ζ2 η1 η2, t^-p ζ1 ζ3 η3] +
    (1 - p) ep[t^(-2+p) ζ1 ζ2 η1 η3, t^-p ζ1 ζ3 η2] - ep[t^(-2+p) ζ1 ζ2 η2 η3, t^(1-p) ζ3] +
    (-1 + p) ep[t^(-2+p) ζ1 ζ2 η2 η3, t^-p ζ1 ζ3 η1] + (1 - p) ep[t^(-2+p) ζ1 ζ3 η1 η2, t^-p ζ1 ζ2 η3] +
    (-1 + p) ep[t^(-2+p) ζ1 ζ3 η1 η3, t^-p ζ1 ζ2 η2] + ep[t^(-2+p) ζ1 ζ3 η2 η3, t^(1-p) ζ2] +
    (1 - p) ep[t^(-2+p) ζ1 ζ3 η2 η3, t^-p ζ1 ζ2 η1] + (-1 + p) ep[t^(-2+p) ζ1 η1 η2 η3, t^-p ζ1 ζ2 ζ3] -
    ep[t^(-2+p) ζ2 ζ3 η2 η3, t^(1-p) ζ1] + (-2 + p) ep[t^(-3+p) ζ1 ζ2 ζ3 η1 η2 η3, t^(1-p) ζ1],
 2 + 2 p → ep[t^(1+p), t^(-2-p) ζ1 ζ2 ζ3 η2 η3] + ep[t^p ζ1 ζ2, t^(-1-p) ζ3 η2 η3] +
    (-1 - p) ep[t^p ζ1 ζ2, t^(-2-p) ζ1 ζ3 η1 η2 η3] - ep[t^p ζ1 ζ3, t^(-1-p) ζ2 η2 η3] +
    (1 + p) ep[t^p ζ1 ζ3, t^(-2-p) ζ1 ζ2 η1 η2 η3] + (-1 - p) ep[t^p ζ1 η1, t^(-2-p) ζ1 ζ2 ζ3 η2 η3] +
    ep[t^p ζ1 η2, t^(-1-p) ζ2 ζ3 η3] + (1 + p) ep[t^p ζ1 η2, t^(-2-p) ζ1 ζ2 ζ3 η1 η3] -
```

```
    ep[t^p ζ1 η3, t^(-1-p) ζ2 ζ3 η2] + (-1 - p) ep[t^p ζ1 η3, t^(-2-p) ζ1 ζ2 ζ3 η1 η2] +
    ep[t^p ζ2 ζ3, t^(-1-p) ζ1 η2 η3] - ep[t^p ζ2 η2, t^(-1-p) ζ1 ζ3 η3] + ep[t^p ζ2 η3, t^(-1-p) ζ1 ζ3 η2] +
    ep[t^p ζ3 η2, t^(-1-p) ζ1 ζ2 η3] - ep[t^p ζ3 η3, t^(-1-p) ζ1 ζ2 η2] + ep[t^p η2 η3, t^(-1-p) ζ1 ζ2 ζ3] -
    p ep[t^(-1+p) ζ1 ζ2 ζ3 η1, t^(-1-p) ζ1 η2 η3] + ep[t^(-1+p) ζ1 ζ2 ζ3 η2, t^-p η3] +
    p ep[t^(-1+p) ζ1 ζ2 ζ3 η2, t^(-1-p) ζ1 η1 η3] - ep[t^(-1+p) ζ1 ζ2 ζ3 η3, t^-p η2] -
    p ep[t^(-1+p) ζ1 ζ2 ζ3 η3, t^(-1-p) ζ1 η1 η2] - p ep[t^(-1+p) ζ1 ζ2 η1 η2, t^(-1-p) ζ1 ζ3 η3] +
    p ep[t^(-1+p) ζ1 ζ2 η1 η3, t^(-1-p) ζ1 ζ3 η2] + ep[t^(-1+p) ζ1 ζ2 η2 η3, t^-p ζ3] -
    p ep[t^(-1+p) ζ1 ζ2 η2 η3, t^(-1-p) ζ1 ζ3 η1] + p ep[t^(-1+p) ζ1 ζ3 η1 η2, t^(-1-p) ζ1 ζ2 η3] -
    p ep[t^(-1+p) ζ1 ζ3 η1 η3, t^(-1-p) ζ1 ζ2 η2] - ep[t^(-1+p) ζ1 ζ3 η2 η3, t^-p ζ2] +
    p ep[t^(-1+p) ζ1 ζ3 η2 η3, t^(-1-p) ζ1 ζ2 η1] - p ep[t^(-1+p) ζ1 η1 η2 η3, t^(-1-p) ζ1 ζ2 ζ3] +
    ep[t^(-1+p) ζ2 ζ3 η2 η3, t^-p ζ1] + (1 - p) ep[t^(-2+p) ζ1 ζ2 ζ3 η1 η2 η3, t^-p ζ1]}
```

```
VNormal[PartSplit[splitRes, 2 p] + (PartSplit[splitRes, 2 + 2 p] /. p -> p - 1)]
```

```
0
```

```
res = Kb[t^-1 ζ1 ζ2 ζ3, pCas] // VNormal;
splitRes = SplitSum[res, _ep, Expand[Grade[#[[1]]] &];
VNormal[PartSplit[splitRes, 2 p] + (PartSplit[splitRes, 2 + 2 p] /. p -> p - 1)]
```

```
0
```

We have checked the components with p far from 0 (so that the action of the generators of $\mathfrak{k}(1|6)$ cannot break the order of operands in ep[x,y]). The situation is different for p=0.

First, not all terms of pCas/.p→0 are present in the Casimir element, but only those with correct order of arguments:

```
cas0 = pCas /. ep[x_, y_] :> 0 /; ! envOrd[x /. p → 0, y /. p → 0] /. p → 0
```

```
ep[t, t^-2 ζ1 ζ2 ζ3 η1 η2 η3] + ep[ζ1 ζ2, t^-1 ζ3 η1 η2 η3] -
 ep[ζ1 ζ3, t^-1 ζ2 η1 η2 η3] + ep[ζ1 η1, t^-1 ζ2 ζ3 η2 η3] -
 ep[ζ1 η2, t^-1 ζ2 ζ3 η1 η3] + ep[ζ1 η3, t^-1 ζ2 ζ3 η1 η2] + ep[ζ2 ζ3, t^-1 ζ1 η1 η2 η3] +
 ep[ζ2 η2, t^-1 ζ1 ζ3 η1 η3] - ep[ζ2 η3, t^-1 ζ1 ζ3 η1 η2] + ep[ζ3 η3, t^-1 ζ1 ζ2 η1 η2] +
 ep[t^-1 ζ1 ζ2 ζ3 η1, η2 η3] - ep[t^-1 ζ1 ζ2 ζ3 η2, η1 η3] + ep[t^-1 ζ1 ζ2 ζ3 η3, η1 η2] -
 ep[t^-1 ζ1 ζ2 η1 η3, ζ3 η2] + ep[t^-1 ζ1 ζ2 η2 η3, ζ3 η1] - ep[t^-1 ζ1 ζ3 η2 η3, ζ2 η1]
```

Next, there are also linear terms:

```
casLin = GeneralSum[a, Select[pComp /. p → 0, Weight[#] == {2, 0, 0, 0} &]]
```

```
a[1] t + a[2] ζ1 η1 + a[3] ζ2 η2 + a[4] ζ3 η3 + a[5] t^-1 ζ1 ζ2 η1 η2 +
 a[6] t^-1 ζ1 ζ3 η1 η3 + a[7] t^-1 ζ2 ζ3 η2 η3 + a[8] t^-2 ζ1 ζ2 ζ3 η1 η2 η3
```

Let us find the coefficients a_i by calculating the linear part of Kb[f, $x \otimes y$]:

```
KbLin[x_, y_] := VNormal[Kb[x, y]] /. ep[_, _] → 0
```

```
casLin1 = GeneralSolve[KbLin[ζ1 η2, cas0 + casLin] == 0, casLin, a]
```

```
a[1] t + a[2] ζ1 η1 + a[2] ζ2 η2 + a[3] ζ3 η3 + a[4] t^-1 ζ1 ζ2 η1 η2 +
  (-2 + a[5]) t^-1 ζ1 ζ3 η1 η3 + a[5] t^-1 ζ2 ζ3 η2 η3 + a[6] t^-2 ζ1 ζ2 ζ3 η1 η2 η3
```

```
casLin2 = GeneralSolve[KbLin[ζ2 η3, cas0 + casLin1] == 0, casLin1, a]
```

```
a[1] t + a[2] ζ1 η1 + a[2] ζ2 η2 + a[2] ζ3 η3 + (-4 + a[3]) t^-1 ζ1 ζ2 η1 η2 +
  (-2 + a[3]) t^-1 ζ1 ζ3 η1 η3 + a[3] t^-1 ζ2 ζ3 η2 η3 + a[4] t^-2 ζ1 ζ2 ζ3 η1 η2 η3
```

```
casLin3 = GeneralSolve[KbLin[ζ2 ζ3, cas0 + casLin2] == 0, casLin2, a]
```

```
a[1] t + 2 t^-1 ζ1 ζ3 η1 η3 + 4 t^-1 ζ2 ζ3 η2 η3 + a[2] t^-2 ζ1 ζ2 ζ3 η1 η2 η3
```

```
casLin4 = GeneralSolve[KbLin[t η1, cas0 + casLin3] == 0, casLin3, a]
```

$$2\,t^{-1}\,\xi_1\,\xi_3\,\eta_1\,\eta_3 + 4\,t^{-1}\,\xi_2\,\xi_3\,\eta_2\,\eta_3 + 4\,t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3$$

To check, we write down 3 components: the linear, p=0, and p=1.

```
cas = casLin4 + cas0 + (pCas /. p → 1)
```

$$\begin{aligned}
&2\,t^{-1}\,\xi_1\,\xi_3\,\eta_1\,\eta_3 + 4\,t^{-1}\,\xi_2\,\xi_3\,\eta_2\,\eta_3 + 4\,t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3 + \\
&\ ep[t, t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3] + ep[\xi_1\,\xi_2, t^{-1}\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - ep[\xi_1\,\xi_3, t^{-1}\,\xi_2\,\eta_1\,\eta_2\,\eta_3] + \\
&\ ep[\xi_1\,\eta_1, t^{-1}\,\xi_2\,\xi_3\,\eta_2\,\eta_3] - ep[\xi_1\,\eta_2, t^{-1}\,\xi_2\,\xi_3\,\eta_1\,\eta_3] + ep[\xi_1\,\eta_3, t^{-1}\,\xi_2\,\xi_3\,\eta_1\,\eta_2] + \\
&\ ep[\xi_2\,\xi_3, t^{-1}\,\xi_1\,\eta_1\,\eta_2\,\eta_3] + ep[\xi_2\,\eta_2, t^{-1}\,\xi_1\,\xi_3\,\eta_1\,\eta_3] - ep[\xi_2\,\eta_3, t^{-1}\,\xi_1\,\xi_3\,\eta_1\,\eta_2] + \\
&\ ep[\xi_3\,\eta_3, t^{-1}\,\xi_1\,\xi_2\,\eta_1\,\eta_2] + ep[t^{-1}\,\xi_1\,\xi_2\,\xi_3\,\eta_1, \eta_2\,\eta_3] - ep[t^{-1}\,\xi_1\,\xi_2\,\xi_3\,\eta_2, \eta_1\,\eta_3] + \\
&\ ep[t^{-1}\,\xi_1\,\xi_2\,\xi_3\,\eta_3, \eta_1\,\eta_2] - ep[t^{-1}\,\xi_1\,\xi_2\,\eta_1\,\eta_3, \xi_3\,\eta_2] + ep[t^{-1}\,\xi_1\,\xi_2\,\eta_2\,\eta_3, \xi_3\,\eta_1] - \\
&\ ep[t^{-1}\,\xi_1\,\xi_3\,\eta_2\,\eta_3, \xi_2\,\eta_1] + ep[t^{2}, t^{-3}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - ep[t\,\xi_1, t^{-2}\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3] + \\
&\ ep[t\,\xi_2, t^{-2}\,\xi_1\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - ep[t\,\xi_3, t^{-2}\,\xi_1\,\xi_2\,\eta_1\,\eta_2\,\eta_3] + \\
&\ ep[t\,\eta_1, t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_2\,\eta_3] - ep[t\,\eta_2, t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_3] + ep[t\,\eta_3, t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2] + \\
&\ ep[t\,\xi_1\,\xi_2, t^{-2}\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - ep[t\,\xi_1\,\xi_3, t^{-2}\,\xi_2\,\eta_1\,\eta_2\,\eta_3] + ep[t\,\xi_1\,\eta_1, t^{-2}\,\xi_2\,\xi_3\,\eta_2\,\eta_3] - \\
&\ ep[t\,\xi_1\,\eta_2, t^{-2}\,\xi_2\,\xi_3\,\eta_1\,\eta_3] + ep[t\,\xi_1\,\eta_3, t^{-2}\,\xi_2\,\xi_3\,\eta_1\,\eta_2] + ep[t\,\xi_2\,\xi_3, t^{-2}\,\xi_1\,\eta_1\,\eta_2\,\eta_3] - \\
&\ ep[t\,\xi_2\,\eta_1, t^{-2}\,\xi_1\,\xi_3\,\eta_2\,\eta_3] + ep[t\,\xi_2\,\eta_2, t^{-2}\,\xi_1\,\xi_3\,\eta_1\,\eta_3] - ep[t\,\xi_2\,\eta_3, t^{-2}\,\xi_1\,\xi_3\,\eta_1\,\eta_2] + \\
&\ ep[t\,\xi_3\,\eta_1, t^{-2}\,\xi_1\,\xi_2\,\eta_2\,\eta_3] - ep[t\,\xi_3\,\eta_2, t^{-2}\,\xi_1\,\xi_2\,\eta_1\,\eta_3] + ep[t\,\xi_3\,\eta_3, t^{-2}\,\xi_1\,\xi_2\,\eta_1\,\eta_2] + \\
&\ ep[t\,\eta_1\,\eta_2, t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_3] - ep[t\,\eta_1\,\eta_3, t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_2] + ep[t\,\eta_2\,\eta_3, t^{-2}\,\xi_1\,\xi_2\,\xi_3\,\eta_1] - \\
&\ ep[\xi_1\,\xi_2\,\xi_3, t^{-1}\,\eta_1\,\eta_2\,\eta_3] + ep[\xi_1\,\xi_2\,\eta_1, t^{-1}\,\xi_3\,\eta_2\,\eta_3] - ep[\xi_1\,\xi_2\,\eta_2, t^{-1}\,\xi_3\,\eta_1\,\eta_3] + \\
&\ ep[\xi_1\,\xi_2\,\eta_3, t^{-1}\,\xi_3\,\eta_1\,\eta_2] - ep[\xi_1\,\xi_3\,\eta_1, t^{-1}\,\xi_2\,\eta_2\,\eta_3] + ep[\xi_1\,\xi_3\,\eta_2, t^{-1}\,\xi_2\,\eta_1\,\eta_3] - \\
&\ ep[\xi_1\,\xi_3\,\eta_3, t^{-1}\,\xi_2\,\eta_1\,\eta_2] - ep[\xi_1\,\eta_1\,\eta_2, t^{-1}\,\xi_2\,\xi_3\,\eta_3] + ep[\xi_1\,\eta_1\,\eta_3, t^{-1}\,\xi_2\,\xi_3\,\eta_2] - \\
&\ ep[\xi_1\,\eta_2\,\eta_3, t^{-1}\,\xi_2\,\xi_3\,\eta_1] + ep[\xi_2\,\xi_3\,\eta_1, t^{-1}\,\xi_1\,\eta_2\,\eta_3] - ep[\xi_2\,\xi_3\,\eta_2, t^{-1}\,\xi_1\,\eta_1\,\eta_3] + \\
&\ ep[\xi_2\,\xi_3\,\eta_3, t^{-1}\,\xi_1\,\eta_1\,\eta_2] + ep[\xi_2\,\eta_1\,\eta_2, t^{-1}\,\xi_1\,\xi_3\,\eta_3] - ep[\xi_2\,\eta_1\,\eta_3, t^{-1}\,\xi_1\,\xi_3\,\eta_2] + \\
&\ ep[\xi_2\,\eta_2\,\eta_3, t^{-1}\,\xi_1\,\xi_3\,\eta_1] - ep[\xi_3\,\eta_1\,\eta_2, t^{-1}\,\xi_1\,\xi_2\,\eta_3] + ep[\xi_3\,\eta_1\,\eta_3, t^{-1}\,\xi_1\,\xi_2\,\eta_2] - \\
&\ ep[\xi_3\,\eta_2\,\eta_3, t^{-1}\,\xi_1\,\xi_2\,\eta_1] + ep[\eta_1\,\eta_2\,\eta_3, t^{-1}\,\xi_1\,\xi_2\,\xi_3] + ep[\xi_1\,\xi_2\,\xi_3\,\eta_1, t^{-1}\,\eta_2\,\eta_3] - \\
&\ ep[\xi_1\,\xi_2\,\xi_3\,\eta_2, t^{-1}\,\eta_1\,\eta_3] + ep[\xi_1\,\xi_2\,\xi_3\,\eta_3, t^{-1}\,\eta_1\,\eta_2] + ep[\xi_1\,\xi_2\,\eta_1\,\eta_2, t^{-1}\,\xi_3\,\eta_3] - \\
&\ ep[\xi_1\,\xi_2\,\eta_1\,\eta_3, t^{-1}\,\xi_3\,\eta_2] + ep[\xi_1\,\xi_2\,\eta_2\,\eta_3, t^{-1}\,\xi_3\,\eta_1] - ep[\xi_1\,\xi_3\,\eta_1\,\eta_2, t^{-1}\,\xi_2\,\eta_3] + \\
&\ ep[\xi_1\,\xi_3\,\eta_1\,\eta_3, t^{-1}\,\xi_2\,\eta_2] - ep[\xi_1\,\xi_3\,\eta_2\,\eta_3, t^{-1}\,\xi_2\,\eta_1] + ep[\xi_1\,\eta_1\,\eta_2\,\eta_3, t^{-1}\,\xi_2\,\xi_3] + \\
&\ ep[\xi_2\,\xi_3\,\eta_1\,\eta_2, t^{-1}\,\xi_1\,\eta_3] - ep[\xi_2\,\xi_3\,\eta_1\,\eta_3, t^{-1}\,\xi_1\,\eta_2] + ep[\xi_2\,\xi_3\,\eta_2\,\eta_3, t^{-1}\,\xi_1\,\eta_1] - \\
&\ ep[\xi_2\,\eta_1\,\eta_2\,\eta_3, t^{-1}\,\xi_1\,\xi_3] + ep[\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{-1}\,\xi_1\,\xi_2] - ep[t^{-1}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2, \eta_3] + \\
&\ ep[t^{-1}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_3, \eta_2] - ep[t^{-1}\,\xi_1\,\xi_2\,\xi_3\,\eta_2\,\eta_3, \eta_1] + ep[t^{-1}\,\xi_1\,\xi_2\,\eta_1\,\eta_2\,\eta_3, \xi_3] - \\
&\ ep[t^{-1}\,\xi_1\,\xi_3\,\eta_1\,\eta_2\,\eta_3, \xi_2] + ep[t^{-1}\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3, \xi_1] + ep[t^{-1}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3, 1]
\end{aligned}$$

```
Kb[ξ1 η2, cas] // VNormal
```

$$0$$

```
Kb[ξ2 η3, cas] // VNormal
```

$$0$$

```
Kb[ξ2 ξ3, cas] // VNormal
```

$$0$$

```
Kb[η1 η2 η3, cas] // VNormal
```

$$\begin{aligned}
&-ep[t\,\xi_1\,\eta_1\,\eta_2, t^{-2}\,\xi_2\,\eta_1\,\eta_2\,\eta_3] - ep[t\,\xi_1\,\eta_1\,\eta_3, t^{-2}\,\xi_3\,\eta_1\,\eta_2\,\eta_3] + \\
&\ ep[t\,\xi_2\,\eta_1\,\eta_2, t^{-2}\,\xi_1\,\eta_1\,\eta_2\,\eta_3] - ep[t\,\xi_2\,\eta_2\,\eta_3, t^{-2}\,\xi_3\,\eta_1\,\eta_2\,\eta_3] + ep[t\,\xi_3\,\eta_1\,\eta_3, t^{-2}\,\xi_1\,\eta_1\,\eta_2\,\eta_3] + \\
&\ ep[t\,\xi_3\,\eta_2\,\eta_3, t^{-2}\,\xi_2\,\eta_1\,\eta_2\,\eta_3] - ep[t\,\eta_1\,\eta_2\,\eta_3, t^{-2}\,\xi_1\,\xi_2\,\eta_1\,\eta_2] - ep[t\,\eta_1\,\eta_2\,\eta_3, t^{-2}\,\xi_1\,\xi_3\,\eta_1\,\eta_3] - \\
&\ ep[t\,\eta_1\,\eta_2\,\eta_3, t^{-2}\,\xi_2\,\xi_3\,\eta_2\,\eta_3] - 2\,ep[t\,\eta_1\,\eta_2\,\eta_3, t^{-3}\,\xi_1\,\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - \\
&\ ep[\xi_1\,\xi_2\,\eta_1\,\eta_2\,\eta_3, t^{-1}\,\eta_1\,\eta_2] - ep[\xi_1\,\xi_2\,\eta_1\,\eta_2\,\eta_3, t^{-2}\,\xi_3\,\eta_1\,\eta_2\,\eta_3] - \\
&\ ep[\xi_1\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{-1}\,\eta_1\,\eta_3] + ep[\xi_1\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{-2}\,\xi_2\,\eta_1\,\eta_2\,\eta_3] - \\
&\ ep[\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{-1}\,\eta_2\,\eta_3] - ep[\xi_2\,\xi_3\,\eta_1\,\eta_2\,\eta_3, t^{-2}\,\xi_1\,\eta_1\,\eta_2\,\eta_3]
\end{aligned}$$

```
VBasis[%] //. x_ep :→ Grade[x[[1]]]
```

```
{5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
```

According to the above, we could have obtained from the term with p=1 the two types of terms: with grade 1+2p=3 and 3+2p=5, of which the first should cancel with the some terms from the previous terms (p−1=0), which takes place, and the one which will definitely cancel with the next term. So we are done, for this generator.

Now, let us pass to other generators.

```
Kb[t η1 , cas] // VNormal
```

```
- ep[t^2 ξ2 , t^-2 ξ3 η1 η2 η3 ] + ep[t^2 ξ3 , t^-2 ξ2 η1 η2 η3 ] - ep[t^2 η1 , t^-2 ξ2 ξ3 η2 η3 ] +
 ep[t^2 η2 , t^-2 ξ2 ξ3 η1 η3 ] - ep[t^2 η3 , t^-2 ξ2 ξ3 η1 η2 ] + ep[t ξ1 ξ2 η1 , t^-2 ξ3 η1 η2 η3 ] -
 ep[t ξ1 ξ3 η1 , t^-2 ξ2 η1 η2 η3 ] + ep[t ξ1 η1 η2 , t^-2 ξ2 ξ3 η1 η3 ] -
 ep[t ξ1 η1 η3 , t^-2 ξ2 ξ3 η1 η2 ] - ep[t ξ2 ξ3 η1 , t^-1 η2 η3 ] + ep[t ξ2 ξ3 η1 , t^-2 ξ1 η1 η2 η3 ] +
 ep[t ξ2 ξ3 η2 , t^-1 η1 η3 ] - ep[t ξ2 ξ3 η3 , t^-1 η1 η2 ] - ep[t ξ2 η1 η2 , t^-1 ξ3 η3 ] -
 ep[t ξ2 η1 η2 , t^-2 ξ1 ξ3 η1 η3 ] + ep[t ξ2 η1 η3 , t^-1 ξ3 η2 ] + ep[t ξ2 η1 η3 , t^-2 ξ1 ξ3 η1 η2 ] -
 ep[t ξ2 η2 η3 , t^-1 ξ3 η1 ] + ep[t ξ3 η1 η2 , t^-1 ξ2 η3 ] + ep[t ξ3 η1 η2 , t^-2 ξ1 ξ2 η1 η3 ] -
 ep[t ξ3 η1 η3 , t^-1 ξ2 η2 ] - ep[t ξ3 η1 η3 , t^-2 ξ1 ξ2 η1 η2 ] + ep[t ξ3 η2 η3 , t^-1 ξ2 η1 ] -
 ep[t η1 η2 η3 , t^-1 ξ2 ξ3 ] + ep[t η1 η2 η3 , t^-2 ξ1 ξ2 ξ3 η1 ] + 2 ep[ξ1 ξ2 ξ3 η1 η2 , t^-1 η1 η3 ] -
 2 ep[ξ1 ξ2 ξ3 η1 η3 , t^-1 η1 η2 ] + 2 ep[ξ1 ξ2 η1 η2 η3 , t^-1 ξ3 η1 ] -
 2 ep[ξ1 ξ3 η1 η2 η3 , t^-1 ξ2 η1 ] - ep[ξ2 ξ3 η1 η2 η3 , 1] + 2 ep[ξ2 ξ3 η1 η2 η3 , t^-1 ξ1 η1 ]
```

```
VBasis[%%] //. x_ep :→ Grade[x[[1]]]
```

```
{5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
```

```
Kb[ξ2 η1 , cas] // VNormal
```

```
0
```

```
Kb[ξ3 η2 , cas] // VNormal
```

```
0
```

```
Kb[η2 η3 , cas] // VNormal
```

```
0
```

```
Kb[ξ1 , cas] // VNormal
```

```
ep[t^2 , t^-3 ξ1 ξ2 ξ3 η2 η3 ] + ep[t ξ1 ξ2 , t^-2 ξ3 η2 η3 ] - 2 ep[t ξ1 ξ2 , t^-3 ξ1 ξ3 η1 η2 η3 ] -
 ep[t ξ1 ξ3 , t^-2 ξ2 η2 η3 ] + 2 ep[t ξ1 ξ3 , t^-3 ξ1 ξ2 η1 η2 η3 ] -
 2 ep[t ξ1 η1 , t^-3 ξ1 ξ2 ξ3 η2 η3 ] + ep[t ξ1 η2 , t^-2 ξ2 ξ3 η3 ] + 2 ep[t ξ1 η2 , t^-3 ξ1 ξ2 ξ3 η1 η3 ] -
 ep[t ξ1 η3 , t^-2 ξ2 ξ3 η2 ] - 2 ep[t ξ1 η3 , t^-3 ξ1 ξ2 ξ3 η1 η2 ] + ep[t ξ2 ξ3 , t^-2 ξ1 η2 η3 ] -
 ep[t ξ2 η2 , t^-2 ξ1 ξ3 η3 ] + ep[t ξ2 η3 , t^-2 ξ1 ξ3 η2 ] + ep[t ξ3 η2 , t^-2 ξ1 ξ2 η3 ] -
 ep[t ξ3 η3 , t^-2 ξ1 ξ2 η2 ] + ep[t η2 η3 , t^-2 ξ1 ξ2 ξ3 ] - ep[ξ1 ξ2 ξ3 η1 , t^-2 ξ1 η2 η3 ] +
 ep[ξ1 ξ2 ξ3 η2 , t^-1 η3 ] + ep[ξ1 ξ2 ξ3 η2 , t^-2 ξ1 η1 η3 ] - ep[ξ1 ξ2 ξ3 η3 , t^-1 η2 ] -
 ep[ξ1 ξ2 ξ3 η3 , t^-2 ξ1 η1 η2 ] - ep[ξ1 ξ2 η1 η2 , t^-2 ξ1 ξ3 η3 ] +
 ep[ξ1 ξ2 η1 η3 , t^-2 ξ1 ξ3 η2 ] + ep[ξ1 ξ2 η2 η3 , t^-1 ξ3 ] - ep[ξ1 ξ2 η2 η3 , t^-2 ξ1 ξ3 η1 ] +
 ep[ξ1 ξ3 η1 η2 , t^-2 ξ1 ξ2 η3 ] - ep[ξ1 ξ3 η1 η3 , t^-2 ξ1 ξ2 η2 ] - ep[ξ1 ξ3 η2 η3 , t^-1 ξ2 ] +
 ep[ξ1 ξ3 η2 η3 , t^-2 ξ1 ξ2 η1 ] - ep[ξ1 η1 η2 η3 , t^-2 ξ1 ξ2 ξ3 ] + ep[ξ2 ξ3 η2 η3 , t^-1 ξ1 ]
```

```
VBasis[%] //. x_ep :→ Grade[x[[1]]]
```

```
{4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}
```

```
Kb[t^-1 ξ1 ξ2 ξ3 , cas] // VNormal
```

```
ep[t ζ1 ζ2, t^-3 ζ1 ζ2 ζ3 η1 η2] + ep[t ζ1 ζ3, t^-3 ζ1 ζ2 ζ3 η1 η3] +
 ep[t ζ2 ζ3, t^-3 ζ1 ζ2 ζ3 η2 η3] + ep[ζ1 ζ2 ζ3 η1, t^-2 ζ1 ζ2 η2] + ep[ζ1 ζ2 ζ3 η1, t^-2 ζ1 ζ3 η3] +
 ep[ζ1 ζ2 ζ3 η1, t^-3 ζ1 ζ2 ζ3 η2 η3] - ep[ζ1 ζ2 ζ3 η2, t^-2 ζ1 ζ2 η1] + ep[ζ1 ζ2 ζ3 η2, t^-2 ζ2 ζ3 η3] -
 ep[ζ1 ζ2 ζ3 η2, t^-3 ζ1 ζ2 ζ3 η1 η3] - ep[ζ1 ζ2 ζ3 η3, t^-2 ζ1 ζ3 η1] - ep[ζ1 ζ2 ζ3 η3, t^-2 ζ2 ζ3 η2] +
 ep[ζ1 ζ2 ζ3 η3, t^-3 ζ1 ζ2 ζ3 η1 η2] + ep[ζ1 ζ2 η1 η2, t^-2 ζ1 ζ2 ζ3] + ep[ζ1 ζ3 η1 η3, t^-2 ζ1 ζ2 ζ3] +
 ep[ζ2 ζ3 η2 η3, t^-2 ζ1 ζ2 ζ3] + 2 ep[t^-1 ζ1 ζ2 ζ3 η1 η2 η3, t^-2 ζ1 ζ2 ζ3]
```

```
VBasis[%] //. x_ep :→ Grade[x[[1]]]
```

```
{4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}
```

So we have tested all the generators and c_2 is indeed invariant.

SuperLie Index