ACE

an Algebraic Combinatorics Environment
for the computer algebra system MAPLE

User's Reference Manual
Version 2.0

Sébastien Veigneau
Université de Marne-la-Vallée
Institut Gaspard Monge

June 2, 1996

Université de
Marne-la-Vallée
Institut Gaspard Monge

ACE

an Algebraic Combinatorics Environment
for the computer algebra system MAPLE

User's Reference Manual

Version 2.0

ACE

Version 2.0

ACE is a joint effort of a number of people at the universities of Caen, Marne-la-Vallée, Paris VII and Rouen. The project was led since 1994 by Sébastien Veigneau (University of Marne-la-Vallée) who is grateful to all people having made up the version 2.0 of `ACE`. All the development has been done under supervision of Alain Lascoux (Paris VII) and Jean-Yves Thibon (Marne-la-Vallée). Many thanks also to Vincent Mulier for his continuing support in the development of the MACINTOSH version of `ACE`, and to Vincent Prosper for providing the necessary TEX knowledge to write this documentation. The following persons have contributed to the `ACE` project:

GÉRARD DUCHAMP[(1)]

Laboratoire d'Informatique de Rouen,
Université de Rouen,
Faculté des Sciences et Techniques,
Place Émile Blondel,
76821 Mont-Saint-Aignan Cedex, France.
`Email: ged@lir.dir.univ-rouen.fr`

RENAUD EPPSTEIN

Institut Gaspard Monge,
Université de Marne-la-Vallée,
2, rue de la Butte Verte,
93166 Noisy-le-Grand Cedex, France.
`Email: eppstein@univ-mlv.fr`

DANIEL KROB[(1)]

LITP,
Université PARIS 7,
2, place Jussieu,
75251 Paris Cedex 05, France.
`Email: dk@litp.ibp.fr`

(1)Supported by the EC network "Algebraic Combinatorics".

Alain Lascoux[2]

LITP,
Université PARIS 7,
2, place Jussieu,
75251 Paris Cedex 05, France.
Email: `al@lascoux.ibp.fr`

Bernard Leclerc[2]

Département de Mathématiques,
Université de Caen,
Esplanade de la Paix, BP5186,
14032 Caen Cedex, France.
Email: `bl@litp.ibp.fr`

Éric Laugerotte

Laboratoire d'Informatique de Rouen,
Université de Rouen,
Faculté des Sciences et Techniques,
Place Émile Blondel,
76821 Mont-Saint-Aignan Cedex, France.
Email: `laugerot@lir.dir.univ-rouen.fr`

Jean-Yves Thibon[2]

Institut Gaspard Monge,
Université de Marne-la-Vallée,
2, rue de la Butte Verte,
93166 Noisy-le-Grand Cedex, France.
Email: `jyt@univ-mlv.fr`

Bun-Chan-Vorac Ung[2]

Institut Gaspard Monge,
Université de Marne-la-Vallée,
2, rue de la Butte Verte,
93166 Noisy-le-Grand Cedex, France.
Email: `ung@univ-mlv.fr`

Sébastien Veigneau[2]

Institut Gaspard Monge,
Université de Marne-la-Vallée,
2, rue de la Butte Verte,
93166 Noisy-le-Grand Cedex, France.
Email: `sv@univ-mlv.fr`

[2] Supported by the EC network "Algebraic Combinatorics".

Copyright notice

Doing business without advertising is
like winking at a girl in the dark.
You know what you are doing,
but nobody else does.

New York Herald Tribune, October 30, 1956.


```
        __/__/ __/_/ __/_/__/
      __/ __/ __/   __/
    __/_/__/ __/   __/_/
  __/   __/ __/   __/
__/    __/ __/_/ __/_/__/
```

Université de
Marne-la-Vallée
Institut Gaspard Monge

New features of release 2.0

> Il ne faut pas dire que la science est utile
> parce qu'elle nous permet de construire des machines.
> Il faut dire : les machines sont utiles parce qu'elles
> nous permettent de faire de la science.
>
> *H. Poincaré.*

Users already familiar with ACE may be interested in the differences between versions 1.0 and 2.0. New users of ACE may skip this part and proceed with the next chapter. In this documentation, new features of release 2.0 of ACE are identified by a ✘ in the margin. Furthermore, each ✘ is referenced – ACE 2.0 (new features) – in the index at the end of this document so that it is quite easy to find out the new possibilities of ACE 2.0. New features of release 2.0 include: ✘

- **New package** BN: this package provides functions to work with the hyperoctahedral group, in particular with its Bruhat order. It gives functions to handle the elements of this group, represented by codes. It also provides functions to make conversions between a code and a signed permutation or a palindromic permutation. ✘

- **New package** BNA: it provides functions to work with several algebras associated to the hyperoctahedral groups B_n, together with their realizations as algebras of operators on the ring of polynomials. It includes the B_n-group algebra together with the B_n-nilCoxeter and B_n-idCoxeter algebra. Bases of these algebras are indexed by codes of B_n. ✘

- **New package** CG: this package gives functions to work with the representation theory of the classical Lie groups, *i.e.* the symplectic and orthogonal groups. It provides functions to compute products of characters, dimension of irreducible representations together with functions to handle new bases of the ring of symmetric polynomials. ✘

- **New package** FM to compute in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ regarded as a free module of rank $n!$ over the ring $\mathbf{Sym}[x_1, \ldots, x_n]$ of symmetric polynomials in the variables $x_1, \ldots, x_n$. We choose the simple Schubert polynomials indexed by permutations in $\mathfrak{S}_n$ as a basis of this module. It essentially includes a new Monk formula [66] that describes the multiplicative structure in this module and allows to express any polynomial in the $x_1, \ldots, x_n$ in terms of simple Schubert polynomials with symmetric polynomials as coefficients. ✘

- **New package** FOCK: this package is devoted to calculations in the Fock space representations of the quantum affine algebras $U_q(\widehat{sl}_n)$. It can in particular compute the canonical bases of the Fock space, whose transition matrices evaluated at $q = 1$ give the decomposition matrices of t-Schur algebras and Hecke algebras of type A at roots of unity. ✘

- **The package `GL` has been replaced by the package** `CLG` which stands for **C**haracters of **L**inear **G**roups.

- ✘ **The package `HEQA` has been replaced by the package** `SHRI` that stands for **S**ymmetric group's **H**ecke algebra **R**epresentations and **I**somorphisms. New functions allow to handle the usual Hecke algebra of type A, the generic Hecke algebra and the Hecke algebra with symmetric parameters.

- ✘ **New functions** have been added in the `ACE` package including functions to handle tables of pre-computed data.

- ✘ **New functions** have been developed in the `NCSF` package. In particular, it is now able to handle quasi-symmetric functions (matrices of change of basis, algebraic computations) and to express any symmetric function in terms of quasi-symmetric functions.

- ✘ **New functions** in the `SGA`, `NCA`, `IDCA` and `HEKA` packages to act on the polynomial ring seen as a free module over symmetric polynomials.

- ✘ **New functions** are now included in the `SYMF` package to handle the monomial symmetric functions. Moreover, special algorithms have been added to compute specific plethysms of symmetric functions.

- ✘ **New functions** have been included in the `SP` package to specialize Schubert polynomials and to give the expansion of any expression in terms of double Schubert polynomials.

- ✘ **New functions** have been added in the `TAB` package including a generator of tableaux, the Schensted correspondence between words and pairs of tableaux, the plactic action of a permutation on a product of columns (rows) and the computation of the left and right keys of a given word.

- **Some bugs** have been fixed and **algorithms have been improved** among which the computation of the character table of the symmetric group (package `SG`).

- ✘ `ACE` 2.0 also includes a new documentation which points out new packages by a ✘ whereas packages with new functions or significant new algorithms are indicated by a ✓.

Preface

ACE is a *Maple* library devoted to Algebraic Combinatorics computations. The project started on April 1994, at the University of Marne-la-Vallée, with the development of three *Maple* packages, dealing respectively with the symmetric group, Schubert polynomials and noncommutative symmetric functions. These packages were available on July 1994. We made up our minds to develop a more general project about algebraic combinatorics on October 1994. The aim was to build an "easy to use" Algebraic Combinatorics Environment. We have chosen *Maple* because this system is widely used by combinatorists and has a convenient user's interface. We had in mind to develop an efficient toolbox of functions for research in algebraic combinatorics. The interface had to be very flexible for first experimentations, whereas it may be more interesting to use some specific programs for large scale computations: for instance, we refer the reader to the `SYMMETRICA` system that is well suited for such computations [3].

The story started in 1994. `ACE` 1.0 was divided into various packages dealing with specific objects such as noncommutative symmetric functions, Schubert polynomials, tableaux or Hecke algebra, each package having the ability to use some functions defined in another package. The first version of `ACE` was available for any MS-DOS or UNIX system, for *Maple V* Release 1, 2 and 3. The new version 2.0 is available since June 1996 for any MS-DOS, UNIX and MACINTOSH system. It provides new packages and new functions in the already existing packages. Several people have been involved in the project in order to develop new programs as for instance to work with the hyperoctahedral group and with several algebras associated to the hyperoctahedral groups B_n, together with their realizations as algebras of operators on the ring of polynomials. Moreover, a package has been developed to work with the representation theory of classical Lie groups (symplectic and orthogonal groups), whereas a new package is devoted to computations in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ regarded as a free module of rank $n!$ over the ring $\mathbf{Sym}[x_1, \ldots, x_n]$ of symmetric polynomials, and another one is devoted to calculations in the Fock space representations of the quantum affine algebras $U_q(\widehat{sl}_n)$.

This documentation describes how this new version of `ACE` can be installed and how the new user can make his first steps towards an efficient use of the environment. It points out the new features of `ACE` version 2.0 and gives a complete documentation about all available functions. New packages will be added in the next version of `ACE`, among which packages about q-calculus, Virasoro algebra, free Lie algebras, quantum linear groups and special symmetric functions.

Sébastien Veigneau, Université de Marne-la-Vallée, June 2, 1996.

Table of contents

1 Overview

The Algebraic Combinatorics Environment is composed of various packages dealing with specific objects. Several packages have been added in the new release of `ACE`. These packages can be identified by a ✘ whereas packages with new functions or significant new algorithms are pointed out by a ✓:

- ➩ `BN`: hyperoctahedral group. ✘
- ➩ `BNA`: algebras related to hyperoctahedral groups. ✘
- ➩ `CG`: classical Lie groups. ✘
- ➩ `CLG`: characters of the linear groups.
- ➩ `FM`: polynomial ring in several variables. ✘
- ➩ `FOCK`: Fock space representations of quantum affine algebras $U_q(\widehat{sl}_n)$. ✘
- ➩ `HEKA`: Hecke algebra.
- ➩ `IDCA`: idCoxeter algebra.
- ➩ `NCA`: nilCoxeter algebra.
- ➩ `NCSF`: noncommutative symmetric functions. ✓
- ➩ `SG`: symmetric group.
- ➩ `SGA`: symmetric group algebra.
- ➩ `SHRI`: symmetric group Hecke algebra representations and isomorphisms. ✓
- ➩ `SP`: Schubert polynomials.
- ➩ `SYMF`: symmetric functions. ✓
- ➩ `TAB`: tableaux. ✓

We give here a brief presentation of these libraries. The background needed to understand all the examples is presented in the next chapter.

1.1 The BN library

✘ This package gives functions to work with the hyperoctahedral group, in particular with its Bruhat order. It provides functions to handle the elements of this group, represented by codes. It also gives functions to make conversions between a code and a signed permutation (`Bar`) or a palindromic permutation (`Palin`). For instance, the `BnInterval` function computes the list of all codes of permutations that are smaller than a given one according to the Bruhat order:

```
ACE> BnInterval([1,1,3]);

      [[1, 1, 3], [1, 1, 2], [1, 2, 1], [1, 1, 1], [1, 0, 3], [1, 0, 2], [1, 2, 0],

       [1, 1, 0], [0, 1, 3], [0, 1, 2], [0, 2, 1], [0, 1, 1], [0, 0, 3], [0, 0, 2],

       [1, 0, 1], [0, 0, 1], [0, 2, 0], [0, 1, 0], [1, 0, 0], [0, 0, 0]]

ACE> map(BnCode2Bar, ");

        [[-3, 2, -1], [3, 2, -1], [-2, 3, -1], [2, 3, -1], [-3, -1, 2], [3, -1, 2],

             [-2, -1, 3], [2, -1, 3], [-3, 2, 1], [3, 2, 1], [-2, 3, 1], [2, 3, 1],

               [-3, 1, 2], [3, 1, 2], [-1, 3, 2], [1, 3, 2], [-2, 1, 3], [2, 1, 3],

               [-1, 2, 3], [1, 2, 3]]
```

in which the `BnCode2Bar` converts a code of B_n into a signed permutation.

1.2 The BNA library

✘ This package provides functions to work with several algebras associated to the hyperoctahedral groups B_n, together with their realizations as algebras of operators on the ring of polynomials. It includes the B_n-group algebra together with the B_n-nilCoxeter and B_n-idCoxeter algebra. Bases of these algebras are indexed by codes of B_n. For example, the `BnNcaOnPol` function realizes the action of an element of the B_n-nilCoxeter algebra on a given expression:

```
ACE> BnNcaOnPol([1,0,1], x1^3*x2^2*x3);

                                       2
                                     x1  x2 x3
```

1.3 The CG library

✘ This package gives functions to work with the representation theory of the classical Lie groups, *i.e.* the symplectic and orthogonal groups. It provides functions to compute products of characters, dimension of irreducible representations together with functions to handle new bases of the ring of symmetric polynomials. For instance, the `LRrSp` function computes the product of two symplectic Schur functions:

```
ACE> LRrSp(Sp[2,1], Sp[2,1,1]);

     Sp[4, 2, 1] + Sp[4, 1, 1, 1] + Sp[3, 3, 1] + 2 Sp[3, 2, 1, 1] + Sp[3, 2, 2]

        + Sp[3, 1, 1, 1, 1] + Sp[2, 2, 2, 1] + Sp[2, 2, 1, 1, 1] + 3 Sp[2, 2, 1]

        + 3 Sp[2, 1, 1, 1] + Sp[1, 1, 1, 1, 1] + 2 Sp[3, 2] + 3 Sp[3, 1, 1]

        + Sp[4, 1] + 2 Sp[1, 1, 1] + 3 Sp[2, 1] + Sp[3] + Sp[1]
```

1.4 The CLG library

This library provides functions to manipulate characters of the linear group $GL(n)$. It essentially amounts to consider symmetric functions in a fixed number of variables. For instance, given the order of the linear group, say 3, the `s2x_n` function converts an expression from the Schur function basis to the basis of monomials (using an algorithm on Schubert polynomials):

```
ACE> CLG_n(3);
                                        3

ACE> expand( s2x_n(s[2,1]) );

           2         2                          2         2      2            2
         x3  x2 + x3  x1 + 2 x3 x2 x1 + x3 x2  + x3 x1  + x2  x1 + x2 x1
```

1.5 The FM library

This package provides functions to compute in the polynomial ring in n variables seen as a free module of rank $n!$ over the ring of symmetric polynomials in the same variables. We choose as basis of this module Schubert polynomials $\mathbf{X}_\mu$ indexed by permutations μ in the symmetric group $\mathfrak{S}_n$ of degree n. For instance, we can express any polynomial in 4 variables in terms of the Schubert polynomials $\mathbf{X}_\mu, \mu \in \mathfrak{S}_4$ with symmetric functions as coefficients expressed in the basis of Schur functions. The `x2XfixScal` realizes this operation by computing scalar products:

✘

```
ACE> FM_n(4);

                                    4
```

```
ACE> x2XfixScal(q^2*x2^3*x4);

         2                            2
      - q  s[1, 1, 1] X[1, 2, 4, 3] + q  s[1, 1, 1] X[2, 1, 3, 4]

            2                         2
         + q  s[1, 1] X[1, 4, 2, 3] + q  s[1, 1] X[1, 3, 4, 2]

            2                         2
         - q  s[1, 1] X[3, 1, 2, 4] - q  s[1, 1] X[2, 3, 1, 4]

            2                      2                  2
         - q  s[1] X[1, 4, 3, 2] + q  X[4, 1, 3, 2] + q  X[3, 4, 1, 2]

            2
         + q  X[2, 4, 3, 1]
```

Furthermore, we also have generalized the Monk formula [66] to describe the multiplicative structure of this module, as illustrated by the following example:

```
ACE> Monkfix(3, X[3,1,4,2]);

     - X[3, 4, 1, 2] - X[4, 1, 3, 2] + s[1, 1, 1, 1] X[1, 2, 3, 4]

         - s[1, 1, 1] X[2, 1, 3, 4] + s[1] X[3, 1, 4, 2]
```

The package is also able to convert any expression given in the basis of Schubert polynomials indexed by permutations in $\mathfrak{S}_\infty$ in terms of Schubert polynomials in $\mathfrak{S}_n$, with coefficients in $\mathbf{Sym}\,[x_1, \ldots, x_n]$. The following example illustrates this computation:

```
ACE> X2Xfix(X[3,2,1] + q*X[3,2,1] + z*X[4,5,1,3,2], collect);

                   (1 + q) X[3, 2, 1, 4] + z s[1, 1, 1] X[3, 4, 1, 2]

                      - z s[1, 1, 1, 1] X[2, 4, 1, 3]
```

1.6 The FOCK library

✘ This package is devoted to calculations in the Fock space representations of the quantum affine algebras $U_q(\widehat{sl}_n)$. It can in particular compute the canonical basis of the Fock space, whose matrices evaluated at $q = 1$ give the decomposition matrices of t-Schur algebras and Hecke algebras at roots of unity. For example, the `CBMat`(n, m) call computes the matrix of the $U_q(\widehat{sl}_n)$-canonical basis of the Fock space in the basis of partitions of weight m, represented by Schur functions:

```
ACE> CBMat(2, 4);

                      [ 1    0  0  0  0 ]
                      [                 ]
                      [ q    1  0  0  0 ]
                      [                 ]
                      [ 0    q  1  0  0 ]
                      [                 ]
                      [      2          ]
                      [ q  q    q  1  0 ]
                      [                 ]
                      [ 2               ]
                      [q     0  0  q  1 ]
```

in which rows and columns are labelled by partitions $[[4], [3,1], [2,2], [2,1,1], [1,1,1,1]]$.

1.7 The HEKA library

This package provides functions to work with the generic Hecke algebra of the symmetric group. This algebra is generated by simple operators τ_i's satisfying braid relations and the Hecke relation: $\tau_i^2 = (q_1 + q_2)\tau_i - q_1 q_2$.

For instance, we have included the **HekaYang** function that computes the Yang-Baxter basis of this algebra (related to the Yang-Baxter equation), seen as a free module with coefficients in the x_i's, q_1, q_2:

```
ACE> HekaYang([3,1,2]);

       (- x3 + x2) [1, 3, 2]                   (x3 - x1) [2, 1, 3]
   - --------------------- + [1, 2, 3] + -------------------
          x2 (q1 + q2)                        x1 (q1 + q2)

                2
         (- x3   + x3 x1 + x3 x2 - x1 x2) [3, 1, 2]
       - ----------------------------------------
                                    2
                       x1 (q1 + q2)  x2
```

1.8 The IDCA library

It provides functions to work in the idCoxeter algebra (or 0-Hecke algebra) associated with the symmetric group. This is the algebra generated by operators π_i satisfying the braid relations:

$$\begin{aligned} \pi_i \pi_j &= \pi_j \pi_i && \text{if } |i-j| > 1, \\ \pi_i \pi_{i+1} \pi_i &= \pi_{i+1} \pi_i \pi_{i+1} && 1 \le i \le n-2, \end{aligned}$$

together with the extra relations: $\pi_i^2 = -\pi_i$. This algebra corresponds to the Hecke algebra for which we set $q = 0$. It is related to *Grothendieck polynomials* and the K-theory of the flag manifold. We can distinguish special elements of this algebra such as the *Demazure* elements: `Demazure`(*perm*) where *perm* is a permutation, is defined as the sum of all permutations below *perm* for the Bruhat order, as illustrated by the following lines:

```
ACE> Demazure([2,3,1]);

                 [2, 3, 1] + [2, 1, 3] + [1, 3, 2] + [1, 2, 3]

ACE> expand( IdcaOnPol(", x1^2*x2) );

                    2                   2        2        2
                  x2  x3 + x3 x2 x1 + x2  x1 + x1  x3 + x1  x2
```

1.9 The `NCA` library

It provides functions to work in the nilCoxeter algebra corresponding to the symmetric group. This algebra is realized as the algebra of divided differences acting on polynomials.

The `NcaOnX` function realizes the action of an element of the nilCoxeter algebra on an expression in the $\mathbf{X}$ Schubert basis. A simple divided difference ∂_i acts on a Schubert polynomial $\mathbf{X}_{perm}$ by sending it to 0 if $perm(i) < perm(i+1)$, or to $\mathbf{X}_\nu$ where ν is obtained by transposing $perm(i)$ and $perm(i+1)$, if $perm(i) > perm(i+1)$.

The following example illustrates this action:

```
ACE> NcaOnX(q^4*[1,3,2] - q^3*[2,1], z*X[3,2,1] - X[2,1,3]);

                      4                3               3
                  z q  X[3, 1, 2] - z q  X[2, 3, 1] + q  X[1, 2]
```

1.10 The `NCSF` library

This package provides functions to work with (integral) noncommutative symmetric functions together with quasi-symmetric functions. For noncommutative symmetric functions, changes of basis, ordinary product, internal product and the transformations of alphabet $\mathrm{A} \to \mathrm{A}/(1-q)$ and $\mathrm{A} \to \mathrm{A}(1-q)$ have been programmed. These transformations are related to interesting idempotents of the symmetric group algebra. For instance:

$$\varphi_n(q) = \frac{1-q^n}{n}\Psi_n\left(\frac{\mathrm{A}}{1-q}\right)$$

corresponds to a Lie idempotent of the descent algebra. This can be checked by the following computation:

```
ACE> SpDirect(Ps[3], q):
ACE> ToR(simplify("*(1-q^3)/3));

                         q R[2, 1]       R[1, 2]
        1/3 R[3] - 1/3 --------- - 1/3 ------- + 1/3 R[1, 1, 1]
                           q + 1         q + 1

ACE> ToR(Internal(", "));

                         q R[2, 1]       R[1, 2]
        1/3 R[3] - 1/3 --------- - 1/3 ------- + 1/3 R[1, 1, 1]
                           q + 1         q + 1
```

The package can also handle quasi-symmetric functions. It provides the usual changes of basis (quasi-ribbon Schur functions, quasi-monomial functions, ...).

Because symmetric functions are quasi-symmetric functions, we also provide functions to give the expression of any symmetric function in terms of quasi-symmetric functions. For instance:

```
ACE> Sf2Qsf(s[2,1]*p3);

    F[5, 1] + F[4, 2] + F[2, 4] + F[1, 5] - F[2, 3, 1] - 2 F[2, 2, 2] - F[1, 3, 2]

       - 2 F[1, 2, 2, 1] - F[2, 1, 2, 1] - F[1, 2, 1, 2] + F[2, 1, 1, 1, 1]

       + F[1, 2, 1, 1, 1] + F[1, 1, 1, 2, 1] + F[1, 1, 1, 1, 2]
```

1.11 The SG library

It provides functions to manipulate permutations. Basic operations on permutations have been programmed as for instance, conversions between the multiple representations (diagram, code, permutation matrix, ...) of the elements of the symmetric group. Furthermore, we have for example the `Bruhat` function that realizes the comparison between two permutations according to the *Bruhat order*, or the `Interval` function that computes the list of the Lehmer codes of all permutations that are smaller than a given one, according to this order.

```
ACE> Interval([3,4,1,2]);

       [[2, 2, 0, 0], [2, 1, 0, 0], [2, 0, 1, 0], [2, 0, 0, 0], [1, 2, 0, 0],

        [1, 1, 0, 0], [1, 0, 1, 0], [1, 0, 0, 0], [0, 2, 1, 0], [0, 2, 0, 0],

                     [0, 1, 1, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]
```

```
ACE> map(Code2Perm, ");

       [[3, 4, 1, 2], [3, 2, 1, 4], [3, 1, 4, 2], [3, 1, 2, 4], [2, 4, 1, 3],

        [2, 3, 1, 4], [2, 1, 4, 3], [2, 1, 3, 4], [1, 4, 3, 2], [1, 4, 2, 3],

                   [1, 3, 4, 2], [1, 3, 2, 4], [1, 2, 4, 3], [1, 2, 3, 4]]
```

One can naturally associate to this interval the generating function of the lengths of permutations. For instance, the following expression:

```
ACE> Betti([3,4,1,2], q);

                          4      3      2
                         q  + 4 q  + 5 q  + 3 q + 1
```

corresponds, in geometric terms, to the Poincaré polynomial

$$\sum q^k \dim \mathrm{H}^{2k}(X_{3412}, \mathbb{Q})$$

of the Schubert variety X_{3412} of the manifold $\mathcal{F}_4$ of complete flags of $\mathbb{C}^4$.

1.12 The SGA library

It provides functions to work in the symmetric group algebra $\mathbb{K}[\mathfrak{S}_n]$ where $\mathbb{K}$ is a ring of coefficients. For instance, we have the `SgaOnPol` function that realizes the action of an element of this algebra on a polynomial, or `SgaMult` which multiplies two elements of the algebra:

```
ACE> K3:=[1,2,3]+q*([2,1,3]+[3,1,2])+q^2*([1,3,2]+[2,3,1])+q^3*[3,2,1]:
ACE> G3:=[1,2,3]-q*([2,1,3]+[3,1,2])+q^2*[3,2,1]:
ACE> map(factor, SgaMult(expand(K3), expand(G3)));

                              2             2
                 (q + 1) (q  + q + 1) (q - 1)  [1, 2, 3]
```

1.13 The SHRI library

✘ This package provides functions to work with the usual Hecke algebra of type A and Hecke algebras of type A with other spectral parameters, essentially the generic Hecke algebra and the Hecke algebras with symmetric parameters. The usual Hecke algebra is generated by simple operators τ_i's satisfying braid relations and the Hecke relation:

$$\tau_i^2 = (q-1)\tau_i + q \quad .$$

For the generic Hecke algebra one takes the relation:

$$\tau_i^2 = (q_1 + q_2)\tau_i - q_1 q_2 \quad .$$

A third case is given, with symmetric parameters:

$$\tau_i^2 = (v^{-1} - v)\tau_i + 1 \quad .$$

For instance, the `ShriRepT` function returns the representation matrix of a given element of the Hecke algebra in the irreducible representation indexed by a partition:

```
ACE> ShriRepT(a*[2,3,1] + b*[3,1,2], [2,1,1]);

                         [                3           4       3       3     ]
                         [ a + b     a q         a q   - a q   - b q        ]
                         [                                                  ]
                         [              2              3       2       2 ]
                         [   0      - a q    - a q - a q   + a q   + b q ]
                         [                                                  ]
                         [                        2                         ]
                         [   0     a q - b        a q   - a q - b q         ]
```

One gets the corresponding matrix for the generic Hecke algebra by adding the extra parameter `G`:

```
ACE> ShriRepT(a*[2,3,1] + b*[3,1,2],[2,1,1], G);

       [                           3                3                             ]
       [     2       2          a q2              q2  (a q2 + a q1 + b q1)        ]
       [ a q1  + b q1         - -----             ------------------------        ]
       [                           q1                           2                 ]
       [                                                      q1                  ]
       [                                                                          ]
       [                                             2       2                    ]
       [                              2         q2 (a q1  + a q2  + a q1 q2 + b q2 q1) ]
       [       0              - a q2         -------------------------------------- ]
       [                                                      q1                  ]
       [                                                                          ]
       [                                   2                                      ]
       [       0         - a q1 q2 - b q1               q2 (a q2 + a q1 + b q1)       ]
```

1.14 The SP library

It provides functions to compute in the linear basis of Schubert polynomials. The *Monk* formula describing the multiplicative structure is used in many algorithms, for instance to express a Schubert polynomial in the basis of monomials: this is done by the `X2x` function which uses the so-called *transition* algorithm. We can compute the complete table of all Schubert polynomials of the symmetric group using the `TableX` function as shown in the following example for $\mathfrak{S}_3$:

```
ACE> TableX(3);

                                   table([
                                       [1, 2, 3] = 1
                                       [2, 1, 3] = x1
                                       [1, 3, 2] = x2 + x1
                                       [2, 3, 1] = x2 x1
                                                     2
                                       [3, 1, 2] = x1
                                                        2
                                       [3, 2, 1] = x2 x1
                                   ])
```

The `TableY` function realizes the same calculation but works with codes using another algorithm. The same combinatorics may be used to define Schubert polynomials in two sets of variables (*double Schubert* polynomials). The product of a double Schubert polynomial and a single variable is realized by the `Monk` function (that extends the formula known in geometry for the intersection of a Schubert variety with a hyperplane):

```
ACE> Monk(4, XX[4,1,2,5,3,7,8,6]);

                  - XX[4, 1, 5, 2, 3, 7, 8, 6] - XX[5, 1, 2, 4, 3, 7, 8, 6]

                  + XX[4, 1, 2, 7, 3, 5, 8, 6] + XX[4, 1, 2, 6, 3, 7, 8, 5]

                                          + y5 XX[4, 1, 2, 5, 3, 7, 8, 6]
```

Double Schubert polynomials are polynomials in two sets of variables $\{x_i\}, \{y_j\}$, which are the universal coefficients in the Newton interpolation of functions in several variables. For many applications, one needs to specialize the second set of variables to a permutation of the first one. The following example computes the table of double Schubert polynomials specialized to $y_1 = x_3, y_2 = x_1, y_3 = x_2$:

```
ACE> TableXX(3, [3,1,2]);
                             table([
                                 [1, 2, 3] = 1
                                 [2, 1, 3] = x1 - x3
                                 [1, 3, 2] = x2 - x3
                                 [2, 3, 1] = (x1 - x3) (x2 - x3)
                                 [3, 1, 2] = 0
                                 [3, 2, 1] = 0
                              ])
```

The cohomology ring of the flag manifold $\mathcal{F}_n$ is isomorphic to $\mathbb{Z}[x_1, \dots, x_n]/I^+$ where I^+ is the ideal generated by the symmetric polynomials in $x_1, \dots, x_n$ without constant term. A geometrical linear basis of the cohomology ring is constituted by Schubert cycles, which are the images of Schubert polynomials in $\mathbb{Z}[x_1, \ldots, x_n]/I^+$. Thus Schubert polynomials allow an efficient "Schubert calculus" on flag manifolds.

1.15 The SYMF library

It provides functions to handle symmetric functions. Several modes are available. In particular, we have reprogrammed almost all functions of the first version of the SF package of J.Stembridge using specific algorithms for the changes of basis to accelerate certain computations.

Newton, *Wronski*, *Muir*, *Pieri* formulae and the *Littlewood-Richardson* rule constitute the heart of all algorithms for the classical changes of basis. For instance, iterating the *Pieri* formula on the Schur function corresponding to the empty partition, gives the expression of a product of elementary symmetric functions as a linear combination of Schur functions indexed by partitions:

```
ACE> PieriSf(h3, s[3,2,2,1]);

         s[3, 3, 2, 2, 1] + s[4, 2, 2, 2, 1] + s[4, 3, 2, 1, 1] + s[4, 3, 2, 2]

             + s[5, 2, 2, 1, 1] + s[5, 2, 2, 2] + s[5, 3, 2, 1] + s[6, 2, 2, 1]
```

The product of a Schur function and a monomial symmetric function is given by the *Muir* formula:

```
ACE> MuirSf(m[2,2,1], s[4,2,2,1]);
                                            s[5, 3, 3, 1, 1, 1] + s[6, 4, 3, 1]

        + s[5, 4, 4, 1] - s[6, 3, 3, 2]  +  s[6, 4, 2, 2] + s[4, 2, 2, 2, 2, 2]

        - s[6, 3, 3, 1, 1] + s[6, 4, 2, 1, 1]  + 3 s[4, 2, 2, 1, 1, 1, 1, 1, 1]

        + 2 s[4, 3, 3, 1, 1, 1, 1]  -  2 s[4, 4, 2, 1, 1, 1, 1] + s[4, 4, 4, 2]

        - 2 s[6, 2, 2, 1, 1, 1, 1]  +  s[4, 4, 4, 1, 1]  -  s[6, 2, 2, 2, 1, 1]

        + s[6, 2, 2, 2, 2] -  s[4, 4, 2, 2, 1, 1]  +  s[4, 2, 2, 2, 1, 1, 1, 1]

        + s[4, 4, 2, 2, 2]  + s[5, 2, 2, 1, 1, 1, 1, 1]  -  s[4, 4, 3, 1, 1, 1]

        - s[4, 3, 3, 2, 2]  + s[4, 4, 3, 3] -  s[4, 3, 3, 3, 1] - s[5, 3, 3, 3]

        - s[6, 3, 2, 1, 1, 1] + s[4, 3, 2, 1, 1, 1, 1, 1] - s[5, 4, 2, 1, 1, 1]

        - s[4, 2, 2, 2, 2, 1, 1] + s[4, 3, 3, 2, 1, 1]
```

The general changes of basis divide calculations into elementary operations easily computed using these formulae. The package provides also some functions to compute specific plethysms such as the function `Pletee` that gives in the basis of Schur functions the plethysm of two elementary symmetric functions:

```
ACE> Pletee(e4, e4);

     s[3, 2, 2, 2, 2, 2, 1, 1, 1] +  s[4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

   + s[4, 4, 4, 1, 1, 1, 1] + s[4, 2, 2, 2, 2, 2, 2] + s[3, 3, 3, 2, 2, 2, 1]

   + s[4, 3, 3, 2, 1, 1, 1, 1] + s[3, 3, 2, 2, 2, 2, 1, 1] + s[3, 2, 2, 2, 2, 2, 2, 1]

   + s[4, 3, 3, 1, 1, 1, 1, 1, 1] + s[4, 2, 2, 2, 2, 1, 1, 1, 1]

   + s[4, 2, 2, 2, 1, 1, 1, 1, 1, 1] + s[4, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1]

   + s[3, 3, 3, 2, 1, 1, 1, 1, 1] + s[3, 3, 3, 1, 1, 1, 1, 1, 1, 1]

   + s[3, 3, 2, 2, 1, 1, 1, 1, 1, 1] + s[3, 2, 2, 2, 2, 1, 1, 1, 1, 1]

   + s[2, 2, 2, 2, 2, 2, 1, 1, 1, 1] + s[3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1]

   + s[3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1] + s[4, 4, 3, 2, 2, 1] + s[4, 3, 3, 3, 3]

   + s[4, 3, 3, 2, 2, 1, 1] + s[3, 3, 3, 3, 2, 1, 1] + s[3, 3, 2, 2, 2, 1, 1, 1, 1]

   + s[4, 3, 2, 2, 2, 1, 1, 1] + s[3, 3, 3, 2, 2, 1, 1, 1]
```

1.16 The `TAB` library

Its purpose is to manipulate tableaux, contretableaux and more generally words, that are transformed into tableaux by an algorithm due to Schensted.

This library also gives, in the algebra of the symmetric group, the orthogonal idempotents corresponding to standard tableaux:

```
ACE> Word2Tab([3,1,2]);

                                  [[3], [1, 2]]

ACE> StdTab2Idemp(");

              - 1/6 [3, 2, 1] - 1/6 [1, 3, 2] + 1/3 [1, 2, 3]

              - 1/6 [3, 1, 2] - 1/6 [2, 3, 1] + 1/3 [2, 1, 3]
```

We have for instance, the `ListStdTab` function that computes the list of all standard tableaux filled with n letters, or the `ListTab` function which returns the list of all tableaux of a given shape and a given weight:

```
ACE> map(Tab2Mat, ListTab([3,2,2], [1,0,0,1,2,1,0,1,1]));

        [ 8  9    ]  [ 8  9    ]  [ 8  9    ]  [ 6  9    ]  [ 6  9    ]  [ 5  9    ]
        [         ]  [         ]  [         ]  [         ]  [         ]  [         ]
       [[ 5  6    ], [ 4  6    ], [ 5  5    ], [ 5  8    ], [ 4  8    ], [ 4  8    ],
        [         ]  [         ]  [         ]  [         ]  [         ]  [         ]
        [ 1  4  5 ]  [ 1  5  5 ]  [ 1  4  6 ]  [ 1  4  5 ]  [ 1  5  5 ]  [ 1  5  6 ]

        [ 6  9    ]  [ 5  9    ]  [ 6  8    ]  [ 5  8    ]
        [         ]  [         ]  [         ]  [         ]
        [ 5  5    ], [ 4  6    ], [ 5  5    ], [ 4  6    ]]
        [         ]  [         ]  [         ]  [         ]
        [ 1  4  8 ]  [ 1  5  8 ]  [ 1  4  9 ]  [ 1  5  9 ]
```

The `PermOnWord` function realizes the action of a permutation on a word (and in particular, on a tableau). This action is compatible with the plactic congruence and Schensted's algorithm. The simple transposition s_i exchanges in a non-trivial manner the number of i and $i+1$ in the word:

```
ACE> PermOnWord([1,3,2], Tab2Word([[2,2,3], [1,1,2,2,3]])):
ACE> Word2Tab(");

                              [[2, 3, 3], [1, 1, 2, 3, 3]]
```

1.17 The TEX library

This package provides an environment of functions that produce output which is suitable for a TEX processor. It knows how to format almost all the types defined in *Maple*, while it knows how to traduce all standard internal *Maple* functions or objects such as integrals, limits, sums, products, and even arrays and tables. A special format devoted to the objects manipulated in `ACE` is now included by default. For instance:

```
ACE> TeX(s[4,2,2,1,1,1]=JtMatrix([4,2,2,1,1,1]), 'TEX_MODE'='MATHS');
$s_{4, 2, 2, 1, 1, 1}=\pmatrix{S_{4}& S_{5}& S_{6}& S_{7}& S_{8}& S_{9}\cr\cr S_{1} &
S_{2}& S_{3}& S_{4}& S_{5}& S_{6}\cr\cr 1& S_{1}& S_{2}& S_{3}& S_{4}& S_{5}\cr\cr 0&
0& 1& S_{1}& S_{2}& S_{3}\cr\cr 0& 0& 0& 1& S_{1}& S_{2}\cr\cr 0& 0& 0& 0& 1 & S_{1}}$
```

which can be included in a TEX file to produce the following output:

$$s_{4,2,2,1,1,1} = \begin{pmatrix} S_4 & S_5 & S_6 & S_7 & S_8 & S_9 \\ S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\ 1 & S_1 & S_2 & S_3 & S_4 & S_5 \\ 0 & 0 & 1 & S_1 & S_2 & S_3 \\ 0 & 0 & 0 & 1 & S_1 & S_2 \\ 0 & 0 & 0 & 0 & 1 & S_1 \end{pmatrix}$$

2

Background

Das Schönste, was wir erleben können,
ist das Geheimnisvolle.

Mein Weltbild, Albert Einstein.

This chapter sketches a few aspects of the underlying theories of the ACE environment. It is not our purpose to develop the complete theory of the symmetric group, tableaux, Schubert polynomials or non commutative symmetric functions but we want to introduce some notions, set up notations and point out some relations between the mathematical objects manipulated by ACE. We first deal with the symmetric group and its algebra together with deformations of this algebra (nilCoxeter algebra, idCoxeter algebra, Hecke algebra). The next section is devoted to representations of the Hecke algebra, just followed by a section about Fock space representations of the quantum affine algebras $U_q(\widehat{sl}_n)$. Then a section deals with characters of classical Lie groups. The next section is devoted to symmetric functions, whereas the following one introduces noncommutative symmetric functions. The next one details the hyperoctahedral groups and some algebras related to these groups. The last two section give respectively a brief introduction about Schubert polynomials and an application of Schubert polynomials to compute in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ seen as a free module over symmetric polynomials in $x_1, \ldots, x_n$.

2.1 Symmetric group

The symmetric group (`SG`) $\mathfrak{S}_n$ of degree n is the group of the $n!$ permutations (`ListPerm`) of the set $[1, n] := \{1, 2, \ldots, n\}$. It is defined for any $n > 0$ and given a $m > n$, we have $\mathfrak{S}_n \hookrightarrow \mathfrak{S}_m$ since we can identify $\mathfrak{S}_n$ with the subgroup of $\mathfrak{S}_m$ by adding to any permutation $w \in \mathfrak{S}_n$ the fixed points $n+1, n+2, \ldots, m$. Using this embedding, we define the group $\mathfrak{S}_\infty$ of all permutations of the set of positive integers that fix all but a finite number of them:

$$\mathfrak{S}_\infty := \bigcup_{i>0} \mathfrak{S}_i \quad .$$

We usually denote a permutation $w \in \mathfrak{S}_n$ by the list of the images of $\{1, 2, \ldots, n\}$ under w, seen as a mapping of $[1, n]$ onto itself: $w := [w(1), w(2), \ldots, w(n)]$.

A basic statistic on permutations is the notion of inversion: given a permutation $w \in \mathfrak{S}_n$, an inversion is defined to be a couple (i, j) of indices such that $1 \leq i < j \leq n$ and $w(i) > w(j)$. We will refer to an inversion on consecutive positions i and $i+1$ as a descent (`Perm2Desc`) of the permutation and conversely, a rise of w is a position i such that $w(i) < w(i+1)$.
The length $l(w)$ (`Perm2Length`) of w is the number of its inversions, *i.e.* the cardinal of the set $I(w)$ (`Inversions`):

$$I(w) := \{(i, j), 1 \leq i < j \leq n \text{ and } w(i) > w(j)\}.$$

The element $[n, n-1, \ldots, 1]$ of $\mathfrak{S}_n$ is called the maximal permutation (`LastPerm`) of $\mathfrak{S}_n$ because it has the maximal number of inversions for the symmetric group of degree n, *i.e.* the binomial $\binom{n}{2}$.

The notion of inversion leads to another coding of the elements of the symmetric group. Given a permutation $w \in \mathfrak{S}_n$, we define the code $c(w)$ (`Perm2Code`) to be the vector of non-negative integers whose i-th component is the number of positions $j > i$ such that $w(j) < w(i)$. One can check that the length of the permutation w is the sum of all components of the code $c(w)$. The code $c(w)$ of the permutation $w := [2, 6, 4, 3, 1, 7, 5]$ is $c(w) := [1, 4, 2, 1, 0, 1, 0]$.

```
ACE> map(Perm2Code, ListPerm(4));

          [[0, 0, 0, 0], [0, 2, 0, 0], [0, 0, 1, 0], [3, 0, 0, 0], [0, 1, 0, 0],

           [0, 2, 1, 0], [0, 1, 1, 0], [3, 0, 1, 0], [2, 0, 0, 0], [2, 2, 0, 0],

           [2, 0, 1, 0], [3, 2, 0, 0], [1, 0, 0, 0], [1, 2, 0, 0], [1, 0, 1, 0],

           [3, 1, 0, 0], [1, 1, 0, 0], [1, 2, 1, 0], [1, 1, 1, 0], [3, 1, 1, 0],

           [2, 1, 0, 0], [2, 2, 1, 0], [2, 1, 1, 0], [3, 2, 1, 0]]
```

A permutation $w \in \mathfrak{S}_n$ can be represented by a $n \times n$ matrix of 0's and 1's. In row r, $r = 1, \ldots, n$ there is a 1 in column $w(r)$ and 0 elsewhere. For instance, the following matrix represents the permutation $w := [2, 6, 4, 3, 1, 7, 5]$:

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

In quite a similar manner, we introduce the diagram $D(w)$ of a permutation $w \in \mathfrak{S}_n$. This combinatorial object is defined to be a $n \times n$ square matrix filled as follows: we put a circle $\circ$ at position (r, c) where r (respectively c) is a row index (resp. column index) whenever there is an inversion $(r, w^{-1}(c))$ in which w^{-1} denotes the inverse permutation of w. The diagram $D(w)$ of the previous permutation is the following:

$$\begin{bmatrix} \circ & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \circ & \cdot & \circ & \circ & \circ & \cdot & \cdot \\ \circ & \cdot & \circ & \cdot & \cdot & \cdot & \cdot \\ \circ & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \circ & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

since there is an inversion between the first position ($w(1) = 2$) and the value 1, between the second position ($w(2) = 6$) and the values $1, 3, 4$ and 5 (second row of the previous diagram), and so on. The code of a permutation w can also be defined as the sequence of numbers of circles in the successive rows of the diagram $D(w)$.

Let $s_i, i = 1, \ldots, n-1$ be the elementary transposition that interchanges i and $i+1$, and fixes all other values. Elementary transpositions satisfy the following relations:

$$s_i s_j = s_j s_i \qquad \text{if } |i-j| > 1 \quad , \tag{2.1}$$

$$s_i s_{i+1} s_i = s_{i+1} s_i s_{i+1} \qquad 1 \le i \le n-2 \quad , \tag{2.2}$$

known as braid relations, together with the extra relation:

$$s_i^2 = 1 \quad , \tag{2.3}$$

this last relation being changed into more general quadratic relations to get the different Hecke algebras (`NCA, IDCA, HEKA`).

We constantly use the decomposition of a permutation as a product of simple transpositions. A product of minimal length is called a *reduced decomposition* (`Perm2Rd`).

```
ACE> Perm2Rd([6,5,4,3,2,1]);

            [5, 4, 3, 2, 1, 5, 4, 3, 2, 5, 4, 3, 5, 4, 5]
```

This decomposition is far from being unique as show by the function `Perm2ListRd` which enumerates all reduced decompositions of a given permutation:

```
ACE> Perm2ListRd([3,1,5,4,2]);

    [[2, 3, 4, 3, 1], [4, 2, 3, 4, 1], [2, 4, 3, 4, 1], [2, 3, 4, 1, 3],

     [2, 3, 1, 4, 3], [2, 1, 3, 4, 3], [4, 2, 3, 1, 4], [2, 4, 3, 1, 4],

     [4, 2, 1, 3, 4], [2, 4, 1, 3, 4], [2, 1, 4, 3, 4]]
```

The number of reduced decompositions is given by the function `Perm2NbRd`:

```
ACE> Perm2NbRd([6,5,4,3,2,1]);

                                 292864
```

We have the following conventions to multiply permutations. Given w and an elementary transposition s_i:

$$w\, s_i := [w(1), \ldots, w(i+1), w(i), \ldots, w(n)] \quad .$$

One can easily compute a reduced decomposition of a permutation w from its diagram. Reading the diagram $D(w)$ from left to right and top to bottom, we replace each circle by a positive integer i starting from the row number, and increasing the value of i by 1 at each step. One moreover puts a `x` in place of the 1's of the matrix representing w. It gives a canonical reduced decomposition of the corresponding permutation by reading this Rothe diagram from right to left and downwards, each i being interpreted as the elementary transposition s_i exchanging i and $i+1$. Given a permutation $w := [2, 6, 4, 3, 1, 7, 5]$, its Rothe diagram is:

```
ACE> Perm2Rothe([2, 6, 4, 3, 1, 7, 5]);

                         [ 1  x  .  .  .  .  . ]
                         [                     ]
                         [ 2  .  3  4  5  x  . ]
                         [                     ]
                         [ 3  .  4  x  .  .  . ]
                         [                     ]
                         [ 4  .  x  .  .  .  . ]
                         [                     ]
                         [ x  .  .  .  .  .  . ]
                         [                     ]
                         [ .  .  .  .  6  .  x ]
                         [                     ]
                         [ .  .  .  .  x  .  . ]
```

from which we deduce a reduced decomposition of w, that is $w := s_1\ s_5 s_4 s_3 s_2\ s_4 s_3\ s_4\ s_6$.

The set of permutations ranked by length, with edges (w, ws_i) is called the *permutohedron*, the order being the *weak order* (which is the natural order for divided differences (`NCA`)). To the permutohedron, one can add edges between permutations which differ by a transposition and with lengths differing by 1. The order is now called the *Bruhat order* (or strong order), and is the natural order for the idCoxeter algebra (`IDCA`).

The interval of permutations below a given one for the Bruhat order is obtained with the function `Interval`:

```
ACE> Interval([3,4,2,1]);

        [[2, 2, 1, 0], [2, 2, 0, 0], [2, 1, 1, 0], [2, 1, 0, 0], [2, 0, 1, 0],

         [2, 0, 0, 0], [1, 2, 1, 0], [1, 2, 0, 0], [1, 1, 1, 0], [1, 1, 0, 0],

         [1, 0, 1, 0], [1, 0, 0, 0], [0, 2, 1, 0], [0, 2, 0, 0], [0, 1, 1, 0],

         [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]
```

whereas the function `Betti` gives the number of permutations in the interval corresponding to a given permutation according to their lengths:

```
ACE> Betti([3,4,2,1], 'q');

                                  5      4      3      2
                                 q  + 3 q  + 5 q  + 5 q  + 3 q + 1
```

2.2 Symmetric group algebra

The symmetric group algebra $\mathbb{K}[\mathfrak{S}_n]$ (`SGA`) where $\mathbb{K}$ stands for any ring of coefficients is defined as the algebra generated by the elementary transpositions $s_i, i = 1, \ldots, n-1$. Thus it has a natural linear basis consisting of all permutations in $\mathfrak{S}_n$.

We give special elements, which are called *Jucis-Murphy* elements:

$$\mathtt{Jucis}(k) := \sum_{i \leq k-1} t_{ik} \quad ,$$

the sum being on all transpositions exchanging i and k, $i = 1, \ldots, k-1$:

```
ACE> Jucis(4,6);

               [4, 2, 3, 1, 5, 6] + [1, 4, 3, 2, 5, 6] + [1, 2, 4, 3, 5, 6]
```

The algebra generated by the Jucis-Murphy elements is a maximal commutative subalgebra of the symmetric group algebra. The symmetric functions in these elements span the center of the group algebra $\mathbb{K}[\mathfrak{S}_n]$.

Using these elements, one can build central idempotents corresponding to standard tableaux, which are mutually orthogonal (`StdTab2Idemp`):

```
ACE> StdTab2Idemp([[3], [1,2,4]]);

  - 1/48 [1, 4, 2, 3] + 1/24 [1, 2, 4, 3] + 5/48 [1, 4, 3, 2] - 1/48 [1, 3, 4, 2]

  - 1/12 [3, 4, 1, 2] - 1/48 [3, 1, 4, 2] - 1/48 [2, 4, 1, 3] + 1/24 [2, 1, 4, 3]

  + 5/48 [2, 4, 3, 1] - 1/48 [2, 3, 4, 1] - 1/12 [3, 4, 2, 1] - 1/48 [3, 2, 4, 1]

  + 1/8 [1, 2, 3, 4]  - 1/48 [4, 1, 2, 3] - 1/16 [1, 3, 2, 4] + 5/48 [4, 1, 3, 2]

  - 1/16 [3, 1, 2, 4] - 1/12 [4, 3, 1, 2] + 1/8 [2, 1, 3, 4]  - 1/48 [4, 2, 1, 3]

  - 1/16 [2, 3, 1, 4] + 5/48 [4, 2, 3, 1] - 1/16 [3, 2, 1, 4] - 1/12 [4, 3, 2, 1]
```

Both functions `Carre` and `Nabla`) give other distinguished linear bases of $\mathbb{K}[\mathfrak{S}_n]$. One can obtain idempotents from them: `Carre`(`LastPerm`(n)) is the sum of all permutations in $\mathfrak{S}_n$ and `Nabla`(`LastPerm`(n)) is the alternating sum of all permutations in $\mathfrak{S}_n$. More generally, corresponding to a given shape, e.g. $[2,3]$, one will have the idempotent `Carre([2,4,1,3,5])` $\times$[1] `Nabla([2,5,1,4,3])` that one gets from reading the following special tableaux and contretableaux:

$$\begin{array}{|c|c|c}\cline{1-2} 2 & 4 & \\ \hline 1 & 3 & \multicolumn{1}{c|}{5} \\ \hline \end{array} \qquad\qquad \begin{array}{|c|c|}\hline 2 & 5 \\ \hline 1 & 4 \\ \hline \multicolumn{1}{c|}{} & 3 \\ \cline{2-2} \end{array}$$

2.3 NilCoxeter algebra

Consider a function f in the variables $\ldots, x_i, x_{i+1}, \ldots$ The divided difference of the function f with respect to the variables x_i and x_{i+1} is:

$$\partial_{x_i,x_{i+1}}(f) := \frac{f(\ldots,x_i,x_{i+1},\ldots) - f(\ldots,x_{i+1},x_i,\ldots)}{x_i - x_{i+1}} \quad . \tag{2.4}$$

This operator sends polynomials to polynomials and decreases degrees by 1. The result is a polynomial which is symmetric in x_i and x_{i+1}.

Consider the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates. For each $i \geq 1$, one can define

$$\partial_i \quad := \quad \partial_{x_i,x_{i+1}} \quad . \tag{2.5}$$

This operator was introduced by Newton and is called Newton's divided difference. In 1973, Bernstein, Gelfand & Gelfand and Demazure established that divided differences satisfy the braid relations:

$$\partial_i\partial_j \quad = \quad \partial_j\partial_i \qquad\qquad \text{if } |i-j| > 1 \quad , \tag{2.6}$$

$$\partial_i\partial_{i+1}\partial_i \quad = \quad \partial_{i+1}\partial_i\partial_{i+1} \qquad 1 \leq i \leq n-2 \quad , \tag{2.7}$$

equation (2.3) being replaced by:

$$\partial_i^2 \quad = \quad 0 \quad . \tag{2.8}$$

[1] The $\times$ operation being realized by the `&!*` *Maple* operator.

Relations (2.6) and (2.7) imply that for any permutation $w \in \mathfrak{S}_n$, there exists a divided difference ∂_w (Newton's case corresponding to elementary transpositions). If $w := s_{i_1} s_{i_2} \cdots s_{i_k}$ is a reduced decomposition of w, one sets:

$$\partial_w := \partial_{i_1} \partial_{i_2} \cdots \partial_{i_k} \quad . \tag{2.9}$$

Considering a polynomial $f \in \mathbb{K}[x_1]$ of degree less than n in the unique variable x_1 and $\{y_1, \ldots, y_{n-1}\}$ denoting any set of elements of $\mathbb{K}$, the Newton interpolation formula may be written:

$$f(x_1) = f(y_1) + \sum_{i=1}^{n-1} \partial_i \cdots \partial_1(f)(x_1 - y_1) \cdots (x_1 - y_i) \quad , \tag{2.10}$$

in which divided differences act on the alphabet $\{y_1, \ldots, y_{n-1}\}$. The `SP` package provides a generalization of the Newton interpolation formula to polynomials in several variables.

The algebra generated by the ∂_i's is called the nilCoxeter algebra (`NCA`), or the algebra of divided differences. It has a linear basis consisting of the ∂_w.

The user has two types of computations at his disposal: formal computations involving only the relations (2.6), (2.7), (2.8), or concrete action on polynomials (`NcaOnPol`).

Finally, another basis (`NcaYang`) is given (*Yang-Baxter basis*), for which the braid relations have been replaced by Yang-Baxter equations:

$$Y_i(u) Y_{i+1}(u+v) Y_i(v) = Y_{i+1}(v) Y_i(u+v) Y_{i+1}(u) \quad ,$$

involving extra parameters.

2.4 IdCoxeter algebra

Consider a function f in the variables $\ldots, x_i, x_{i+1}, \ldots$ The isobaric divided difference of the function f with respect to the variables x_i and x_{i+1} is:

$$\pi_{x_i, x_{i+1}}(f) := \frac{f(\ldots, x_i, x_{i+1}, \ldots)\, x_{i+1} - f(\ldots, x_{i+1}, x_i, \ldots)\, x_i}{x_i - x_{i+1}} \quad . \tag{2.11}$$

This operator sends polynomials to polynomials and preserves the degree.

Consider the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates. For each $i \geq 1$, one can define

$$\pi_i \quad := \quad \pi_{x_i, x_{i+1}} \quad . \tag{2.12}$$

This operator still satisfies the braid relations but now

$$\pi_i^2 \quad = \quad -\pi_i \quad . \tag{2.13}$$

By product of simple operators, one has operators π_w indexed by permutations, which constitute a linear basis of the idCoxeter algebra (`IDCA`), *i.e.* the algebra generated by the π_i's. The action on polynomials is given by the `IdcaOnPol` function:

```
ACE> IdcaOnPol(q^2*[3,1,2], x1^2*x2*y1^2*y2^3);

                    2                3    2
                   q  (x2 + x3) y2  y1   x1 x3
```

For this algebra, one has also a Yang-Baxter linear basis (`IdcaYang`):

```
ACE> IdcaYang([3,2,1]);

         (x1 - x3) [2, 1, 3]   (x1 - x2) (x2 - x3) (x1 - x3) [3, 2, 1]
         ------------------- + ---------------------------------------
                 x1                               2
                                                x1  x2

              (x1 - x3) [1, 3, 2]   (x1 - x3) (x1 - x2) [2, 3, 1]
            + ------------------- + -----------------------------
                      x1                          2
                                                x1

              (x1 - x3) (x2 - x3) [3, 1, 2]
            + ----------------------------- + [1, 2, 3]
                         x1 x2
```

2.5 Generic Hecke algebra

The simple operators now act as follows:

$$\tau_i(f) := (q_1 x_2 + q_2 x_1)\, \partial_i(f) + q_1\, f \quad . \tag{2.14}$$

They still satisfy braid relations but now the square of a simple operator involves two fixed parameters:

$$\tau_i^2 \quad = \quad \tau_i\,(q_1 + q_2) - q_1 q_2 \quad . \tag{2.15}$$

Usually, one takes $q_1 = q$, $q_2 = -1$.

Taking products, one has operators τ_w indexed by permutations, which constitute a linear basis of the generic Hecke algebra (`HEKA`), *i.e.* the algebra generated by the τ_i's. Another basis is Yang-Baxter basis (`HekaYang`):

```
ACE> HekaYang([2,1]);

                                  (x1 - x2) [2, 1]
                         [1, 2] - ----------------
                                    x1 (q1 + q2)
```

Finally, the action of an arbitrary element h of the Hecke algebra on a polynomial in the x_i's is given by the `HekaOnPol` function:

```
ACE> HekaOnPol([2,1], x1);

                              x2 q1 + x1 q2 + q1 x1
```

2.6 Symmetric group Hecke algebra representations and isomorphisms

2.6.1 Usual Hecke algebras

These *Hecke* algebras come from geometry over finite fields [90]. Let K be a ring of coefficients and $q \in K$. The usual Hecke algebra of type A, denoted $\mathcal{H}_n$ is a deformation of the algebra of the symmetric group $\mathfrak{S}_n$. With $(T_i)_{1\leq i<n}$ as set of generators, its presentation is:

$$\begin{cases} [T_i, T_j] = 0 & i < i+1 < j < n \quad , \\ T_iT_{i+1}T_i = T_{i+1}T_iT_{i+1} & i+1 < n \quad , \\ (T_i + 1)(T_i - q) = 0 & i < n \quad . \end{cases}$$

As the braid relations appear in the presentation, the product:

$$T_\sigma := T_{i_1}T_{i_2}\dots T_{i_k} \quad ,$$

is independent of a reduced decomposition $\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}$ of the permutation σ. It is well known that $(T_\sigma)_{\sigma\in\mathfrak{S}_n}$ is a linear basis of the free module $\mathcal{H}_n$. Hence, for $n < m$, we get a natural embedding $\mathcal{H}_n \hookrightarrow \mathcal{H}_m$, and define:

$$\mathcal{H}_\infty = \mathcal{H} := \bigcup_{i>0} \mathcal{H}_i \quad .$$

The typical element h of $\mathcal{H}$ then is a sum:

$$h = \sum_{\sigma\in\mathfrak{S}_\infty} \lambda_\sigma T_\sigma \quad .$$

There is a unique scalar product on $\mathcal{H}_n$ satisfying:

$$\begin{cases} < 1|1 >= 1 \quad , \\ < 1|T_\sigma >= 0,\ \sigma \in \mathfrak{S}_n \quad , \\ < T_ih_1|h_2 >=< h_1|T_ih_2 > \quad , \end{cases}$$

for $h_1, h_2 \in \mathcal{H}_n$. It will be called *invariant* and, if K is a field and q not a root of unity, it makes $\mathcal{H}_n$ into a symmetric algebra. More explicitly:

$$< T_\sigma|T_\mu >:= q^{l(\sigma)}\delta_{\sigma\mu} \quad ,$$

with $\sigma, \mu \in \mathfrak{S}_n$ and $l(\sigma)$ is the length of the permutation σ.

2.6.2 Other Hecke algebras

The usual Hecke algebra $\mathcal{H}$ seen previously is a particular case of the generic Hecke algebra $\mathcal{H}_{q_1,q_2}$, $q_1, q_2 \in K$, for $q_1 = -1$ and $q_2 = q$. Its presentation is:

$$\begin{cases} [T_i, T_j] = 0 & i < i+1 < j < n \quad , \\ T_iT_{i+1}T_i = T_{i+1}T_iT_{i+1} & i+1 < n \quad , \\ (T_i - q_1)(T_i - q_2) = 0 & i < n \quad , \end{cases}$$

for $q_1, q_2 \in K$. Again $(T_\sigma)_{\sigma\in\mathfrak{S}_n}$ (constructed as formerly) is a linear basis of $\mathcal{H}_{n,q_1,q_2}$ [89] and we still have an embedding $\mathcal{H}_{n,q_1,q_2} \hookrightarrow \mathcal{H}_{m,q_1,q_2}$ for $n < m$. In the same way, we define:

$$\mathcal{H}_{\infty,q_1,q_2} = \mathcal{H}_{q_1,q_2} := \bigcup_{i>0} \mathcal{H}_{i,q_1,q_2} \quad .$$

Moreover, one has the Hecke algebra $\mathcal{H}_{-v^{-1},v}$ often considered in the theory of quantum groups. It uses symmetric spectral parameters namely $-v^{-1}, v$. Semilinear isomorphisms (`ShriConvert`) between these algebras are described in the tables 2.1, 2.2, 2.3 and examples are given:

Spectra	$-1, q$	$-v^{-1}, v$
$-1, q$	$\bullet$	$\begin{pmatrix} q \\ T_\sigma \end{pmatrix} \mapsto \begin{pmatrix} v^2 \\ vT_\sigma \end{pmatrix}$
$-v^{-1}, v$	$\begin{pmatrix} v \\ T_\sigma \end{pmatrix} \mapsto \begin{pmatrix} \sqrt{q} \\ \frac{1}{\sqrt{q}}T_\sigma \end{pmatrix}$	$\bullet$

Table 2.1: The isomorphism between $\mathcal{H}$ and $\mathcal{H}_{-v^{-1},v}$.

Spectra	$-1, q$	q_1, q_2
$-1, q$	$\bullet$	$\begin{pmatrix} q \\ T_\sigma \end{pmatrix} \mapsto \begin{pmatrix} \frac{-q_2}{q_1} \\ -q_1T_\sigma \end{pmatrix}$
q_1, q_2	$\begin{pmatrix} q_2 \\ T_\sigma \end{pmatrix} \mapsto \begin{pmatrix} -qq_1 \\ -q_1T_\sigma \end{pmatrix}$	$\bullet$

Table 2.2: The isomorphism between $\mathcal{H}$ and $\mathcal{H}_{q_1,q_2}$.

Spectra	$-v^{-1}, v$	q_1, q_2
$-v^{-1}, v$	$\bullet$	$\begin{pmatrix} v \\ T_\sigma \end{pmatrix} \mapsto \begin{pmatrix} \sqrt{\frac{-q_2}{q_1}} \\ \frac{-1}{q_1}\sqrt{\frac{-q_1}{q_2}}T_\sigma \end{pmatrix}$
q_1, q_2	$\begin{pmatrix} q_2 \\ T_\sigma \end{pmatrix} \mapsto \begin{pmatrix} -v^2q_1 \\ -vq_1T_\sigma \end{pmatrix}$	$\bullet$

Table 2.3: The isomorphism between $\mathcal{H}_{-v^{-1},v}$ and $\mathcal{H}_{q_1,q_2}$.

```
ACE> ShriConvert(1-q*[2,3,1,4]+[2,1],Q,S);

                                 4
                     [1] - v   [2, 3, 1] + v [2, 1]


ACE> ShriConvert(",S,G);

                                 /    q2 \1/2 /    q1 \1/2
                                 |- ----|     |- ----|     [2, 1]
              q2 [2, 3, 1]       \    q1 /     \    q2 /
         [1] + ------------ - -----------------------------
                     3                      q1
                   q1

ACE> ShriConvert(",G,Q);

                                      1/2        1/2
                 [1] - q [2, 3, 1] + q      (1/q)      [2, 1]
```

2.6.3 Irreducible representations

We can construct a complete family of irreducible representations of $\mathcal{H}$ as orbits of q-deformation of the Vandermonde determinant. For an integral vector $I \in \mathbb{N}^r$, let $j_0 = 0$, $j_1 = i_1$, $j_2 = i_1 + i_2$, …, $j_r = i_1 + \cdots + i_r$. The *q-Vandermonde* of index I is by definition:

$$\Delta_I := \prod_{m=0}^{r-1} \prod_{j_m < k < \ell \leq j_{m+1}} (q a_\ell - a_k) \quad .$$

The deformation of the usual *Specht* modules is then provided by the orbit of a projection of Δ_I under an action on polynomials which is the quotient of the faithful representation of $\mathcal{H}_n$ by divided difference operators [1] that is, for $h < k$:

$$\begin{cases} T_i.(\ldots x_i^h x_{i+1}^k \ldots) & := & (\ldots x_i^k x_{i+1}^h \ldots) \quad , \\ T_i.(\ldots x_i^h x_{i+1}^h \ldots) & := & (\ldots x_i^h x_{i+1}^h \ldots) \quad , \\ T_i.(\ldots x_i^k x_{i+1}^h \ldots) & := & (q-1)(\ldots x_i^h x_{i+1}^k \ldots) + q(\ldots x_i^k x_{i+1}^h \ldots) \quad . \end{cases}$$

When I is increasing (resp. decreasing) a natural basis of the Specht module is indexed by contretableaux (resp. tableaux) (`ShriRepCT`, `ShriRepT`) (see figure 2.1).

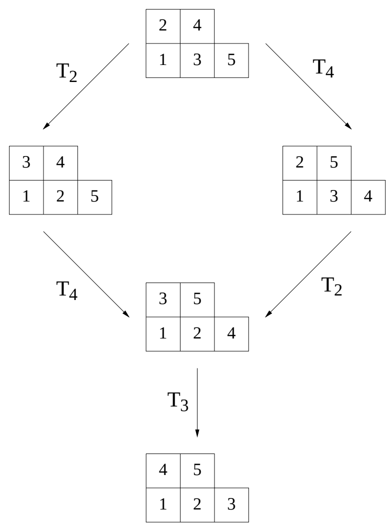


Figure 2.1: A Specht module indexed by tableaux.

```
ACE> ShriRepT([2,1]-[3,1,2],[2,2]);

                          [              2 ]
                          [ -1    - 2 q    ]
                          [                ]
                          [  1      2 q    ]

ACE> ShriRepCT([2,1,4,3]+q,[1,1,2]);

                         [                 3 ]
                         [  q    - q    - q  ]
                         [                   ]
                         [                2  ]
                         [ -1     1      q   ]
                         [                   ]
                         [  0     0      0   ]
```

In the semi-simple case, inverses (`ShriInv`) in $\mathcal{H}_n$ can be computed by means of the minimal polynomials (`ShriCpCT`, `ShriCpT`, `ShriMpCT` or `ShriMpT`). For $e \in \mathcal{H}_n$, I an integral vector (a shape) and x an indeterminate, let $P(e, I, x)$ denote the minimal polynomial of the matrix of e in a representation of shape I. The minimal polynomial Q of e in $\mathcal{H}_n$ is then:

$$Q = lcm_{|I|=n} P(e, I, x) \quad ,$$

which can be written $Q = xR + a_0$. If $a_0 \neq 0$, we get $e^{-1} = \dfrac{1}{a_0} R(e)$.

```
ACE> ShriInv(1-[2,1]);

                         (q - 2) [1]       [2, 1]
                     1/2 ----------- - 1/2 ------
                            q - 1           q - 1
```

If an element is not invertible the function of the package returns 0.

```
ACE> ShriInv([3,1,2]+[2,1]);

                                    0
```

Via isomorphisms, the representations and its operations are available in Hecke algebras with other spectral parameters, the letters G (resp. S) corresponding to the generic (resp. symmetric) algebra.

```
ACE> ShriRepCT([2,1,4,3]+[2,1], [1,1,2], G);

              [                                         3    ]
              [                                       q2     ]
              [      0           q1 q2 + q2           ---    ]
              [                                        q1    ]
              [                                              ]
              [     2                                   2    ]
              [ - q1   - q1  (q1 + q2) (q1 + 1)       q2     ]
              [                                              ]
              [      0                0           q1 q2 + q1 ]

ACE> ShriInv(v+[2,1], S);

                       2
                  (2 v   - 1) [1]            v [2, 1]
              1/2 ---------------- - 1/2 ----------------
                      2        2 1/2        2 1/2    2
                    (v   - 1) (v )         (v )    (v   - 1)
```

The coefficients of the characters tables (`ShriCharTable`) of the generic Hecke algebra can be quickly computed by a bivariate deformation of the Murnaghan - Nakayama rule, due to J. Désarménien. The character of an element is then obtained by conjugation and linearity:

```
ACE> ShriCharTable(4);

             [     3          2                2                      ]
             [   q2         q2               q2               q2     1 ]
             [                                                         ]
             [      2                                                  ]
             [ q1 q2   q2 (q1 + q2)  q2 q1 + q2 (q1 + q2)  2 q2 + q1  3 ]
             [                                                         ]
             [                              2     2                    ]
             [    0        q2 q1          q2   + q1        q1 + q2   2 ]
             [                                                         ]
             [   2                                                     ]
             [ q1   q2  q1 (q1 + q2)  q1 (q1 + q2) + q2 q1  2 q1 + q2  3 ]
             [                                                         ]
             [     3          2                2                      ]
             [   q1         q1               q1               q1     1 ]

ACE> ShriCharTable(2*[2,1]-[4,1,5,3,2], [3,1,1]);

                                            3   3
                       6 q2 + 6 q1 + q1   q2
```

2.6.4 Orthogonal minimal idempotents

Remarkable self-adjoint elements are the *Jucis-Murphy* ones. They are the q -deformation of the elements previously defined for the symmetric group. The i-th ($i \leq n$) Jucis-Murphy element (`ShriJucis`) is equal to:

$$\sum_{k=1}^{i-1} q^{k-i} T_{(k,i)} \quad ,$$

where (k, i) stands for the transposition of k and i.

```
ACE> ShriJucis(4);

                [4, 2, 3, 1]   [1, 4, 3, 2]   [1, 2, 4, 3]
                ------------ + ------------ + ------------
                      3              2              q
                     q              q
```

In case $\mathcal{H}_n$ is semi-simple, they generate a (self-adjoint) maximal commutative subalgebra. Minimal idempotents are polynomials in these elements indexed by Young tableaux coded by Yamanouchi words by right reading (`ShriIdemp`). For instance the tableau of figure 2.2 corresponds to the Yamanouchi word 1 3 1 2 1 2 1.

6			
2	4		
1	3	5	7

Figure 2.2: A Young tableau.

```
ACE> ShriIdemp([1,1,1]);

       [2, 3, 1]              [2, 1]               [3, 1, 2]                [1]
  -------------------- + -------------------- + -------------------- + --------------------
    2                      2                      2                      2
  (q  + q + 1) (q + 1)   (q  + q + 1) (q + 1)   (q  + q + 1) (q + 1)   (q  + q + 1) (q + 1)

                       [1, 3, 2]              [3, 2, 1]
                 + -------------------- + --------------------
                     2                      2
                   (q  + q + 1) (q + 1)   (q  + q + 1) (q + 1)
```

Let I be a shape and e_t a minimal idempotent as above with t a Young tableau of shape I. The sum:

$$C_I = \sum_{shape(t)=I} e_t \quad ,$$

is a central idempotent (**`ShriCentralIdemp`**). It is equal to the projection of $1_{\mathcal{H}}$ onto the block of shape I. Indeed the idempotents C_I are computed by Kilmoyer's formula [86]:

$$C_I = \frac{\dim V_I}{\chi_I(d_I)} d_I \quad ,$$

with

$$d_I = \sum_{\sigma \in \mathfrak{S}_n} q^{-l(\sigma)} \chi_I(T_\sigma) T_\sigma \quad ,$$

V_I being the Specht module of I.

```
ACE> ShriCentralIdemp([2,1]);

                   [2, 3, 1]   (q - 1) [2, 1]    [3, 1, 2]
                 - ---------- + -------------- - ----------
                     2              2              2
                    q  + q + 1     q  + q + 1     q  + q + 1

                           q [1]     (q - 1) [1, 3, 2]
                   + 2 ---------- + -----------------
                         2                2
                        q  + q + 1       q  + q + 1
```

2.7 Fock space representation of $U_q(\widehat{sl}_n)$

The package FOCK provides an environment for computing in the Fock representation of the quantum affine algebra $U_q(\widehat{sl}_n)$. Following [37], we identify the (q-deformed) Fock space with the vector space $\mathcal{F}$ spanned by Young diagrams λ. We also find convenient to regard $\mathcal{F}$ as the space of symmetric functions, by identifying λ with the Schur function s_λ. The main routines compute:

① The action of the generators $e_i,\ f_i$ of $U_q(\widehat{sl}_n)$ on linear combinations of Young diagrams.

② The crystal graph of $\mathcal{F}$ and of the basic $U_q(\widehat{sl}_n)$-module $M(\Lambda_0)$ embedded in it.

③ The global (lower and upper) crystal bases of $M(\Lambda_0)$.

④ The Gram matrix of the Shapovalov form of $M(\Lambda_0)$ in the lower crystal basis.

One motivation for implementing these routines was their conjectured application to the representation theory of Hecke algebras at roots of unity [34]. Indeed, as explained in Section 2.7.7 below, they yield the first efficient general algorithm for computing their decomposition matrices (over a field of characteristic 0).

In [34], canonical bases of the whole Fock space $\mathcal{F}$ were introduced, extending the global crystal bases of $M(\Lambda_0)$. The main idea is to extend to $\mathcal{F}$ the canonical involution of $M(\Lambda_0)$. Routines computing this involution and the associated canonical bases are included in the package (see Section 2.7.6).

It is conjectured that the entries of the transition matrix from the natural basis of $\mathcal{F}$ to its lower canonical basis are q-analogues of the decomposition numbers of the v-Schur algebras for $v = \sqrt[n]{1}$. (It is actually proved for an infinite number of them.)

2.7.1 The quantum affine algebra $U_q(\widehat{sl}_n)$

We recall the definition of $U_q(\widehat{sl}_n)$. Let $\mathfrak{h}$ be a $(n+1)$-dimensional vector space over $\mathbb{Q}$ with basis $\{h_0, h_1, \ldots, h_{n-1}, D\}$. We denote by $\{\Lambda_0, \Lambda_1, \ldots, \Lambda_{n-1}, \delta\}$ the dual basis of $\mathfrak{h}^*$, that is,

$$\langle \Lambda_i, h_j\rangle = \delta_{ij}, \quad \langle \Lambda_i, D\rangle = 0, \quad \langle \delta, h_i\rangle = 0, \quad \langle \delta, D\rangle = 1 \quad ,$$

and we set $\alpha_i = 2\Lambda_i - \Lambda_{i-1} - \Lambda_{i+1} + \delta_{i0}\,\delta$ for $i = 0, 1, \ldots, n-1$. In these formulas it is understood that $\Lambda_n = \Lambda_0$ and $\Lambda_{-1} = \Lambda_{n-1}$. The $n \times n$ matrix $[\langle \alpha_i, h_j\rangle]$ is the generalized Cartan matrix associated to $\widehat{sl}_n$. The weight lattice is $P = (\bigoplus_{i=0}^{n-1} \mathbb{Z}\Lambda_i) \bigoplus \mathbb{Z}\delta$, its dual is $P^\vee = (\bigoplus_{i=0}^{n-1} \mathbb{Z}\, h_i) \bigoplus \mathbb{Z}\, D$, and the root lattice is $Q = \bigoplus_{i=0}^{n-1} \mathbb{Z}\alpha_i$. One defines $U_q(\widehat{sl}_n)$ as the associative algebra with 1 over $\mathbb{Q}(q)$ generated by the symbols $e_i,\ f_i,\ 0 \le i \le n-1$, and $q^h,\ h \in P^\vee$, subject to the relations:

$$q^h\, q^{h'} = q^{h+h'}, \quad q^0 = 1 \quad , \quad q^h e_j q^{-h} = q^{\langle \alpha_j, h\rangle} e_j \quad ,$$

$$q^h f_j q^{-h} = q^{-\langle \alpha_j, h\rangle} f_j \quad ,$$

$$[e_i, f_j] = \delta_{ij} \frac{q^{h_i} - q^{-h_i}}{q - q^{-1}} \quad ,$$

$$\sum_{k=0}^{1-\langle \alpha_i, h_j\rangle} (-1)^k \begin{bmatrix} 1 - \langle \alpha_i, h_j\rangle \\ k \end{bmatrix} e_i^{1-\langle \alpha_i, h_j\rangle - k} e_j e_i^k = 0 \quad (i \neq j) \quad ,$$

$$\sum_{k=0}^{1-\langle \alpha_i, h_j\rangle} (-1)^k \begin{bmatrix} 1 - \langle \alpha_i, h_j\rangle \\ k \end{bmatrix} f_i^{1-\langle \alpha_i, h_j\rangle - k} f_j f_i^k = 0 \quad (i \neq j) \quad .$$

Here we follow the usual notation for q-integers, q-factorials and q-binomial coefficients:

$$[k] = \frac{q^k - q^{-k}}{q - q^{-1}} \quad , [k]! = [k]\,[k-1]\,\cdots[1], \qquad \begin{bmatrix} m \\ k \end{bmatrix} = \frac{[m]!}{[m-k]!\,[k]!} \quad .$$

2.7.2 Fock space representation of $U_q(\widehat{sl}_n)$

An important representation of $U_q(\widehat{sl}_n)$ can be obtained by letting it act on an infinite dimensional vector space with distinguished basis the set $\mathcal{P}$ of all Young diagrams (or partitions):

$$\mathcal{F} = \bigoplus_{\lambda \in \mathcal{P}} \mathbb{Q}(q)\,\lambda \quad .$$

$\mathcal{F}$ is known as the *Fock space*. To describe this action we need to introduce some notation.

Let λ be a partition represented by its Young diagram. Let N be the node (or box) at the intersection of row number i and column number j. The *content* of N is $c = j - i$, and its *n-residue* is $r = c \bmod n$. N is called a *r-node*. For example, the nodes of $\lambda = (3\,2\,1^2)$ have the following 3-residues:

$$\begin{array}{|c|cc}\hline 0 & & \\ \hline 1 & & \\ \hline 2 & \multicolumn{1}{|c|}{0} & \\ \hline 0 & \multicolumn{1}{|c|}{1} & \multicolumn{1}{c|}{2} \\ \hline \end{array}$$

A *removable node* is a node of the boundary of λ which can be removed, leaving a predecessor ν of λ in Young's lattice. An *indent node* is a concave corner on the rim of λ where a node can be added, giving a child of λ in Young's lattice. For instance, $\lambda = (3\,2\,1^2)$ has three removable nodes (two have residue 0, one has residue 2), and four indent nodes (one has residue 0, one has residue 1, and two have residue 2).

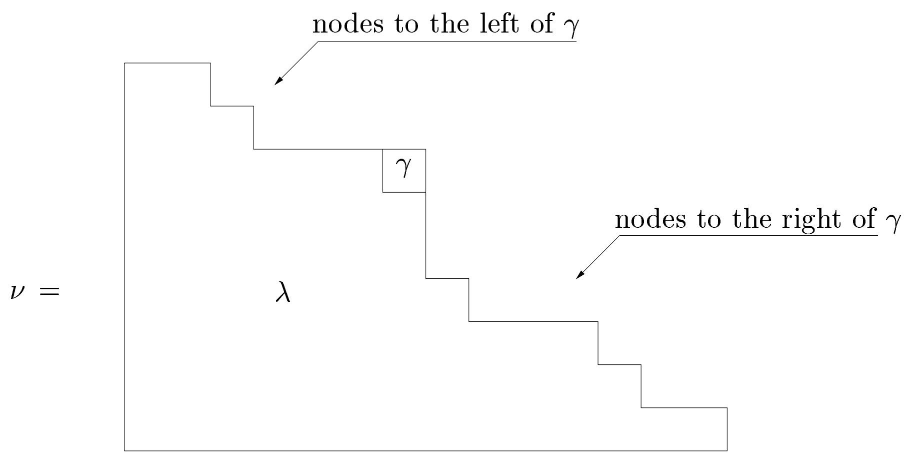
Let $i \in \{0, 1, \ldots, n-1\}$ and let λ, ν be two partitions such that ν is obtained from λ by filling an indent i-node γ. We set:

$N_i(\lambda) = \sharp\{$ indent i-nodes of λ $\} - \sharp\{$ removable i-nodes of λ $\}$,

$N_i^l(\lambda, \nu) = \sharp\{$ indent i-nodes of λ situated to the *left* of γ (not counting γ) $\}$ $-\sharp\{$ removable i-nodes of λ situated to the *left* of γ $\}$,

$N_i^r(\lambda, \nu) = \sharp\{$ indent i-nodes of λ situated to the *right* of γ (not counting γ) $\}$$-\sharp\{$ removable i-nodes of λ situated to the *right* of γ $\}$,

$N^0(\lambda) = \sharp\{$ 0-nodes of λ $\}$.



The algebra $U_q(\widehat{sl}_n)$ acts on $\mathcal{F}$ by [27] [37]:

$q^{h_i}\,\lambda = q^{N_i(\lambda)}\,\lambda\,,$

$q^D\,\lambda = q^{-N^0(\lambda)}\,\lambda\,,$

$f_i\lambda = \sum_\nu q^{N_i^r(\lambda,\nu)}\,\nu$, sum over all partitions ν such that ν/λ is a i-node,

$e_i\nu = \sum_\lambda q^{-N_i^l(\lambda,\nu)}\,\lambda$, sum over all partitions λ such that ν/λ is a i-node.

The action of the raising and lowering operators e_i and f_i has been implemented in `FOCK`. The following example shows the Maple instructions performed to compute the action of these operators on $\lambda = (3,2,1,1)$ in the case $n = 3$:

```
ACE> EE(s[3,2,1,1],3,0);

                     q s[3, 1, 1, 1] + s[3, 2, 1]

ACE> EE(s[3,2,1,1],3,1);

                                  0

ACE> EE(s[3,2,1,1],3,2);

                           s[2, 2, 1, 1]
                           -------------
                                 2
                                q
```

```
ACE> FF(s[3,2,1,1],3,0);

                                s[4, 2, 1, 1]

ACE> FF(s[3,2,1,1],3,1);

                                s[3, 3, 1, 1]

ACE> FF(s[3,2,1,1],3,2);

                        s[3, 2, 2, 1]
                        ------------- + s[3, 2, 1, 1, 1]
                              q
```

It is easy to see that $\mathcal{F}$ is not an irreducible $U_q(\widehat{sl}_n)$-module. Obviously, the empty partition $\emptyset$ is a highest weight vector of weight Λ_0, *i.e.* it satisfies:

$$e_i\emptyset = 0, \quad q^{h_i}\,\emptyset = q^{\delta_{i,0}}\,\emptyset \quad .$$

The irreducible submodule $U_q(\widehat{sl}_n)\,\emptyset$ is the so-called *basic representation* of $U_q(\widehat{sl}_n)$. It will be denoted by $M(\Lambda_0)$.

There is a natural grading of $\mathcal{F}$ given by:

$$\deg\lambda = m \quad \Longleftrightarrow \quad \lambda \vdash m \quad .$$

This is the principal gradation of $\mathcal{F}$. Set:

$$\mathcal{F}_m = \bigoplus_{\deg\lambda=m} \mathbb{Q}(q)\,\lambda \quad , \quad M(\Lambda_0)_m = M(\Lambda_0) \cap \mathcal{F}_m \quad .$$

Then, $\dim\mathcal{F}_m = p(m)$, the number of partitions of m, and $\dim M(\Lambda_0)_m = p^*(m)$, the number of n-regular partitions of m.

The list of partitions of 6 and of 2-regular partitions of 6 can be obtained by the following instructions:

```
ACE> ListPart(6);

 [[6], [5, 1], [4, 2], [4, 1, 1], [3, 3], [3, 2, 1], [3, 1, 1, 1], [2, 2, 2],

     [2, 2, 1, 1], [2, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]

ACE> ListRegPart(2,6);

                        [[6], [5, 1], [4, 2], [3, 2, 1]]
```

Transition matrices between bases of $\mathcal{F}$ or $M(\Lambda_0)$ are indexed by partitions taken in that order (*i.e.* reverse lexicographic order).

2.7.3 Crystal graph of the Fock space

This is an infinite graph with vertices the partitions $\lambda \in \mathcal{P}$ and whose edges record the action of Kashiwara's renormalized lowering operators $\tilde{f}_i$ on these partitions. To describe these edges we introduce the notion of *good node* [37, 33]. Fix a residue $r \in \{0, 1, \ldots, n-1\}$ and consider the sequence of removable and indent r-nodes obtained by reading the boundary of λ from left to right. Thus, for $\lambda = (3\,2\,1^2)$ and $n = 3$ one has:

- residue $r = 0$: $R\,R\,I$
- residue $r = 1$: I
- residue $r = 2$: $I\,I\,R$

Given such a sequence of letters R, I, one can perform the following algorithm:

- read the sequence from left to right and put between brackets all pairs of consecutive letters $R\,I$,
- repeat the procedure ignoring the letters between brackets, until no pattern $R\,I$ is left.

The first R outside the brackets (if any) then indicates the good node of λ of residue r (if any).

For example, this algorithm applied to the sequence:

$$I\,R\,R\,R\,I\,R\,I\,R\,I\,I\,I\,I\,R\,I\,R\,R$$

goes through the following steps:

$$I\,R\,R\,(R\,I)\,(R\,I)\,(R\,I)\,I\,I\,I\,(R\,I)\,R\,R$$

$$I\,R\,(R\,(R\,I)\,(R\,I)\,(R\,I)\,I)\,I\,I\,(R\,I)\,R\,R$$

$$I\,(R\,(R\,(R\,I)\,(R\,I)\,(R\,I)\,I)\,I)\,I\,(R\,I)\,R\,R$$

and the penultimate R corresponds to a good node.

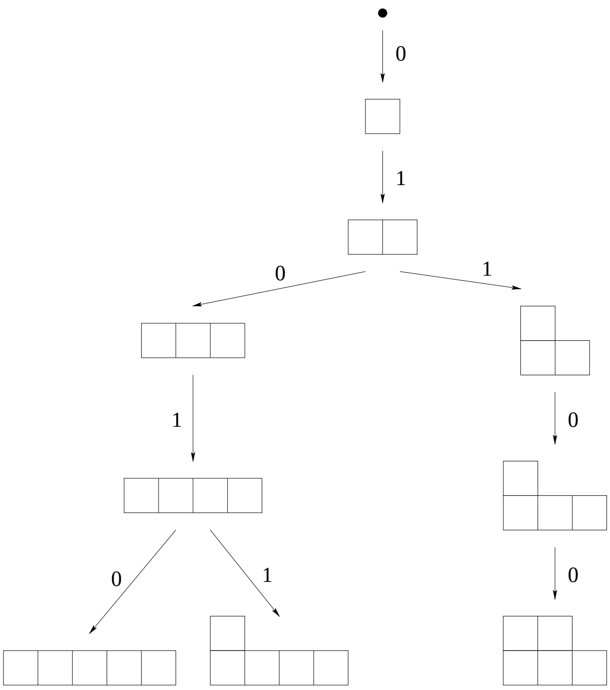
With this rule, we see that $\lambda = (3\,2\,1^2)$ has one good 2-node, no good 1-node, and one good 0-node (the leftmost removable 0-node).

The *crystal graph* of $\mathcal{F}$ is the infinite graph whose vertices are the partitions λ and whose arrows are given by

$$\lambda \xrightarrow{i} \mu \quad \Longleftrightarrow \quad \tilde{f}_i\,\lambda = \mu \quad \Longleftrightarrow \quad \lambda \text{ is obtained from } \mu \text{ by removing a good } i\text{-node} \quad .$$

The decomposition of the crystal graph into connected components reflects the decomposition of $\mathcal{F}$ into irreducible components. In particular, the crystal graph of the basic representation $M(\Lambda_0)$ is the connected component of the empty partition in the crystal graph of $\mathcal{F}$.

The crystal graph of the basic representation of $U_q(\widehat{sl}_2)$ up to partitions of 5 is shown below:



The following example shows how to obtain the images of $\lambda = (3,2,2,1)$ under the operators $\tilde{f}_i, \tilde{e}_i$ for $n = 3$:

```
ACE> Ftilda(3,0,[3,2,1,1]);

                                  0

ACE> Ftilda(3,1,[3,2,1,1]);

                              [3, 3, 1, 1]

ACE> Ftilda(3,2,[3,2,1,1]);

                              [3, 2, 2, 1]

ACE> Etilda(3,0,[3,2,1,1]);

                               [3, 2, 1]

ACE> Etilda(3,1,[3,2,1,1]);

                                  0

ACE> Etilda(3,2,[3,2,1,1]);

                              [2, 2, 1, 1]
```

The set of vertices of the crystal graph of a $U_q(\mathfrak{g})$-module is endowed with a natural action of the Weyl group of the Lie algebra $\mathfrak{g}$ [31]. In the case $\mathfrak{g} = \widehat{sl}_n$, the Weyl group is the affine symmetric group. The action of a simple reflection s_i of W on λ is defined by:

$$s_i\lambda = \begin{cases} \tilde{f_i}^{\langle \mathrm{wt}\,(\lambda), h_i\rangle}\lambda & \text{if } \langle \mathrm{wt}\,(\lambda), h_i\rangle \geq 0 \quad , \\ \tilde{e_i}^{-\langle \mathrm{wt}\,(\lambda), h_i\rangle}\lambda & \text{if } \langle \mathrm{wt}\,(\lambda), h_i\rangle \leq 0 \quad , \end{cases}$$

where wt (λ) is the weight of λ as a vector of the $U_q(\widehat{sl}_n)$-module $\mathcal{F}$. In other words, $s_i\lambda$ is the mirror image of λ in the reflection through the middle of the string of colour i of the crystal graph going through λ.

The following example shows how to obtain the images of $\lambda = (3, 2, 1, 1)$ under the simple reflections s_i ($n = 3$):

```
ACE> Reflect(3,0,[3,2,1,1]);

                              [3, 2, 1]

ACE> Reflect(3,1,[3,2,1,1]);

                              [3, 3, 1, 1]

ACE> Reflect(3,2,[3,2,1,1]);

                              [3, 2, 2, 1]
```

The crystal graph of $M(\Lambda_0)$ has a natural symmetry of order 2 coming from the automorphism of $U_q(\widehat{sl}_n)$ defined by:

$$e_i^{\#} = e_{-i}, \quad f_i^{\#} = f_{-i}, \quad (q^{h_i})^{\#} = q^{h_{-i}}, \quad (q^D)^{\#} = q^D \quad ,$$

where the indices i are understood modulo n. The corresponding involution on the set of partitions is called the Mullineux-Kleshchev involution because of its interpretation in the modular representation theory of the symmetric groups [33].

The image of $\lambda = (3, 2, 2, 1)$ under the Mullineux-Kleshchev map for $n = 3$ can be obtained by the following instruction:

```
ACE> MK(3,[3,2,1,1]);

                              [5, 2]
```

2.7.4 Canonical basis of the basic representation

The *global lower crystal basis* of $M(\Lambda_0)$ (or simply the canonical basis) is defined in terms of a lattice and an involution.

Let $A \subset \mathbb{Q}(q)$ denote the subring of rational functions without pole at $q = 0$. The *lower crystal lattice* L of $\mathcal{F}$ is:

$$L = \bigoplus_{\lambda \in \mathcal{P}} A\,\lambda \quad .$$

To introduce an involution $v \longrightarrow \overline{v}$ of $M(\Lambda_0)$, one starts from the involution $P \longrightarrow \overline{P}$ of $U_q(\widehat{sl}_n)$ defined as the ring automorphism satisfying:

$$\overline{q} = q^{-1}, \quad \overline{q^h} = q^{-h}, \qquad (h \in \mathfrak{h}) \quad ,$$

$$\overline{e_i} = e_i, \quad \overline{f_i} = f_i, \quad i = 0, 1, \ldots, n-1 \quad .$$

Then, for $v = P\,\emptyset \in M(\Lambda_0)$, one sets $\overline{v} = \overline{P}\,\emptyset$. Finally, one denotes by $U_{\mathbb{Q}}^-$ the sub-$\mathbb{Q}[q,q^{-1}]$-algebra of $U_q(\widehat{sl}_n)$ generated by $f_i^{(k)} := f_i^k/[k]!$, and one sets $M(\Lambda_0)_{\mathbb{Q}} = U_{\mathbb{Q}}^-\,\emptyset$.

By [30], there exists a unique $\mathbb{Q}[q,q^{-1}]$-basis $\{G(\mu), \mu \in \mathcal{P}_n\}$ of $M(\Lambda_0)_{\mathbb{Q}}$ such that:

(G1) $\quad G(\mu) \equiv \mu \mod qL$,

(G2) $\quad \overline{G(\mu)} = G(\mu)$.

The basis $\{G(\mu)\}$ is called the *lower global crystal basis* of $M(\Lambda_0)_{\mathbb{Q}}$ (or $M(\Lambda_0)$).

A simple and efficient algorithm to compute $\{G(\mu)\}$ has been proposed in [34]. This algorithm is directly inspired by the techniques described in [29] for computing the decomposition matrices of symmetric groups.

The vectors $G(\mu)$ for all n-regular partitions of m, expanded on the natural basis λ of $\mathcal{F}$ are returned by `FOCK` in a *Maple* table named `TableG(n,m)`. Thus, the canonical basis in degree 6 for $n = 2$ can be obtained by the following instruction:

```
ACE> TableG(2,6);

table([
    (3, 2, 1) = s[3, 2, 1]
                                             2
    (5, 1) = s[5, 1] + q s[4, 2] + q  s[4, 1, 1] + q s[3, 1, 1, 1]

               2                   3
         + q  s[2, 2, 1, 1] + q  s[2, 1, 1, 1, 1]
                                                    2
    6 = s[6] + q s[5, 1] + q s[4, 1, 1] + q  s[3, 1, 1, 1]

               2                        3
         + q  s[2, 1, 1, 1, 1] + q  s[1, 1, 1, 1, 1, 1]
                 2                3                                2
    (4, 2) = q  s[2, 2, 2] + q  s[2, 2, 1, 1] + q s[4, 1, 1] + q  s[3, 1, 1, 1]

         + s[4, 2] + q s[3, 3]
])
```

2.7.5 Kashiwara-Miwa-Stern algebra and ribbon tableaux

For $q = 1$, the operators:

$$a_k = nk\frac{\partial}{\partial p_{nk}} \quad , \qquad b_k = p_{nk} \quad ,$$

commute with the action of $\widehat{sl}_n$ on the Fock space. The quantum analogues A_k, B_k of these operators have been found by Kashiwara-Miwa-Stern [32]. In [35], other generators U_k, V_k of the associative algebra generated by the A_k, B_k are described in terms of ribbon tableaux. V_k is a q-analogue of multiplication by the plethysm $p_n[h_k]$, and U_k corresponds to $D_{p_n[h_k]}$. More generally, for a partition λ, one defines $V_\lambda = V_{\lambda_1}V_{\lambda_2}\ldots V_{\lambda_r}$, and similarly U_λ. The vectors $v_\lambda = V_\lambda\emptyset$ form a basis of $U_q(\widehat{sl}_n)$-highest weight vectors in the Fock space, which can be conveniently computed by enumerating n-ribbon tableaux of weight λ and recording their spins. A recursive formula for the spin polynomials avoids the full enumeration.

The examples below illustrate respectively the computation of $A_2(4,3,2,1)$ for $U_q(\widehat{sl}_3)$, $B_2(2,1)$ for $U_q(\widehat{sl}_2)$, $U_{(2,1)}(4,2,1,1)$ and $V_{(1,1)}(1)$ for $U_q(\widehat{sl}_2)$:

```
ACE> AOp(s[4,3,2,1], 2, 3):
ACE> Tos(", collect);

                             2                          2
                   (- 1 + q ) s[1, 1, 1, 1]   (- 1 + q ) s[4]
                   ------------------------ + ---------------
                                3                      3
                               q                      q


ACE> BOp(s[2,1], 2, 2):
ACE> Tos(", collect);

            s[2, 2, 2, 1]             s[2, 1, 1, 1, 1, 1]
            ------------- + s[6, 1] - ------------------- - s[4, 3]
                   2                           2
                  q                           q

ACE> UOp(s[4,2,1,1], [2,1], 2):
ACE> Tos(", collect);

                                  s[2]     s[1, 1]
                                - ---- + 2 -------
                                   q          2
                                             q

ACE> VOp(s[1], [1,1], 2):
ACE> Tos(", collect);

         s[1, 1, 1, 1, 1]   s[2, 2, 1]     s[3, 1, 1]
         ---------------- + ---------- - 2 ---------- + s[3, 2] + s[5]
                 2               2             q
                q               q
```

2.7.6 Canonical basis of the Fock space representation

Since $\mathcal{F}$ is not an irreducible representation of $U_q(\widehat{sl}_n)$, it is not obvious to apply Kashiwara's or Lusztig's method to define a canonical basis of the whole Fock space $\mathcal{F}$. It is known that the Fock space representation of the affine Lie algebra $\widehat{gl}_n$ is irreducible. Using a q-deformation of the action of $U(\widehat{gl}_n)$ on $\mathcal{F}$ described in [32], one can define a natural semi-linear involution $v \mapsto \overline{v}$ of $\mathcal{F}$ extending the involution of Section 2.7.4 [36].

A routine giving the action of this involution on a linear combination of partitions of a given degree m has been implemented. One can also obtain the matrix $\mathbf{A}_m(q) = [a_{\lambda\,\mu}(q)]_{\lambda,\,\mu\vdash m}$ defined by:

$$\overline{\mu} = \sum_{\lambda\vdash m} a_{\lambda\mu}(q)\lambda \quad .$$

as illustrated by the examples below:

```
ACE> Invol(3,7,s[3,2,1,1]);
                          2
                        (q  - 1) s[2, 2, 2, 1]      2
       s[3, 2, 1, 1] + ---------------------- + (q  - 1) s[2, 2, 1, 1, 1]
                                 q




ACE> InvolMat(2,5);

                        [     1        0        0        0    0  0  0 ]
                        [                                              ]
                        [     0        1        0        0    0  0  0 ]
                        [                                              ]
                        [     0        0        1        0    0  0  0 ]
                        [                                              ]
                        [   2                 2                        ]
                        [  q  - 1            q  - 1                    ]
                        [  ------      0     ------      1    0  0  0 ]
                        [     q                 q                      ]
                        [                                              ]
                        [    2                         2               ]
                        [   q  - 1            2       q  - 1           ]
                        [ - ------     0     q  - 1   ------  1  0  0 ]
                        [      2                        q              ]
                        [     q                                        ]
                        [                                              ]
                        [             2                                ]
                        [            q  - 1                            ]
                        [     0      ------     0        0    0  1  0 ]
                        [               q                              ]
                        [                                              ]
                        [                      2        2              ]
                        [   2                 q  - 1   q  - 1          ]
                        [  q  - 1      0    - ------   ------  0  0  1 ]
                        [                        2        q            ]
                        [                       q                      ]
```

Using this extended involution and imitating Section 2.7.4 one obtains a canonical basis $\{G(\lambda)\}$ of $\mathcal{F}$. The subset $\{G(\mu)\}$ labelled by n-regular partitions μ coincides with the canonical basis of $M(\Lambda_0)$.

The following example shows how to compute a given vector of the canonical basis, as well as the transition matrix from $\{\lambda\}$ to $\{G(\lambda)\}$:

```
ACE> CB(2,8,[3,2,2,1]);

                                                        2
               s[3, 2, 2, 1] + q s[2, 2, 2, 2] + q  s[2, 2, 2, 1, 1]

ACE> CBmat(2,6);

                    [ 1   0   0   0   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [ q   1   0   0   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [ 0   q   1   0   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [      2                                ]
                    [ q  q    q   1   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [ 0   0   q   0   1  0   0  0  0  0  0 ]
                    [                                       ]
                    [ 0   0   0   0   0  1   0  0  0  0  0 ]
                    [                                       ]
                    [  2       2                            ]
                    [ q   q  q    q   q  0   1  0  0  0  0 ]
                    [                                       ]
                    [          2                            ]
                    [ 0   0  q    q   q  0   0  1  0  0  0 ]
                    [                                       ]
                    [      2   3   2   2                    ]
                    [ 0  q   q   q   q   0  q  q  1  0  0 ]
                    [                                       ]
                    [  2   3                  2             ]
                    [ q  q    0  q    0  0  q   0  q  1  0 ]
                    [                                       ]
                    [  3           2                        ]
                    [ q   0   0  q    0  0   0  0  0  q  1 ]
```

2.7.7 Application: decomposition matrices of Hecke and q-Schur algebras

It was conjectured in [34] and proved independently by Ariki [25] and Grojnowski (building on results announced in [26]) that the matrices expressing the global lower crystal basis of $M(\Lambda_0)$ on the natural basis of the Fock space, are q-analogues of the decomposition matrices of the Hecke algebras $H_m(v)$, for v a primitive n-th root of unity.

Similarly, it was conjectured in [36] that the matrices expressing the canonical basis of $\mathcal{F}$ on the natural basis of the Fock space, are q-analogues of the decomposition matrices of the v-Schur algebras $\mathcal{S}_m(v)$ at $v = \sqrt[n]{1}$.

Thus, to obtain the decomposition matrix of $\mathcal{S}_6(-1)$, the following instructions are performed:

```
ACE> CBmat(2,6);
                          [ 1   0   0   0   0  0   0  0  0  0  0 ]
                          [                                       ]
                          [ q   1   0   0   0  0   0  0  0  0  0 ]
                          [                                       ]
                          [ 0   q   1   0   0  0   0  0  0  0  0 ]
                          [                                       ]
                          [     2                                 ]
                          [ q  q    q   1   0  0   0  0  0  0  0 ]
                          [                                       ]
                          [ 0   0   q   0   1  0   0  0  0  0  0 ]
                          [                                       ]
                          [ 0   0   0   0   0  1   0  0  0  0  0 ]
                          [                                       ]
                          [ 2       2                             ]
                          [q    q  q    q   q  0   1  0  0  0  0 ]
                          [                                       ]
                          [         2                             ]
                          [ 0   0  q    q   q  0   0  1  0  0  0 ]
                          [                                       ]
                          [     2   3   2   2                     ]
                          [ 0  q   q   q   q   0   q  q  1  0  0 ]
                          [                                       ]
                          [ 2   3                  2              ]
                          [q   q    0   q   0  0  q   0  q  1  0 ]
                          [                                       ]
                          [ 3           2                         ]
                          [q    0   0  q    0  0   0  0  0  q  1 ]

ACE> subs(q=1,");
                             [ 1  0  0  0  0  0  0  0  0  0  0 ]
                             [                                 ]
                             [ 1  1  0  0  0  0  0  0  0  0  0 ]
                             [                                 ]
                             [ 0  1  1  0  0  0  0  0  0  0  0 ]
                             [                                 ]
                             [ 1  1  1  1  0  0  0  0  0  0  0 ]
                             [                                 ]
                             [ 0  0  1  0  1  0  0  0  0  0  0 ]
                             [                                 ]
                             [ 0  0  0  0  0  1  0  0  0  0  0 ]
                             [                                 ]
                             [ 1  1  1  1  1  0  1  0  0  0  0 ]
                             [                                 ]
                             [ 0  0  1  1  1  0  0  1  0  0  0 ]
                             [                                 ]
                             [ 0  1  1  1  1  0  1  1  1  0  0 ]
                             [                                 ]
                             [ 1  1  0  1  0  0  1  0  1  1  0 ]
                             [                                 ]
                             [ 1  0  0  1  0  0  0  0  0  1  1 ]
```

2.8 Characters of classical Lie groups

Let G be one of the following groups: $GL_n(\mathbb{C}), Sp_{2n}(\mathbb{C}), O_n(\mathbb{C})$. The formal character of the irreducible representation V_λ with highest weight λ is:

$$\mathrm{Char}(V_\lambda) = \sum_\mu \mathrm{dim} V_\lambda(\mu) e^\mu \quad , \tag{2.16}$$

where $V_\lambda(\mu)$ is the subspace of vectors of weight μ in V_λ.

Let $R \subset \mathbb{R}^d$ be the root system of the considered group, and $v_1, \ldots, v_d$ the canonical basis of $\mathbb{R}^d$. If one sets $e^{v_i} = x_i$, then $\mathrm{Char}(V_\lambda)$ is equal to a specialization of a certain symmetric function, called the *universal character* of V_λ.

- For $GL_n(\mathbb{C})$,
$$\mathrm{Char}(V_\lambda) = s_\lambda(x_1, \ldots, x_n) \quad , \tag{2.17}$$
where s_λ is a Schur function.

- For $Sp_{2n}(\mathbb{C})$ we have:
$$\mathrm{Char}(V_\lambda) = Sp_\lambda(x_1, \ldots, x_n, x_1^{-1}, \ldots, x_n^{-1}) \quad , \tag{2.18}$$
where Sp_λ is a symplectic Schur function.

- For $O_{2r}(\mathbb{C})$:
$$\mathrm{Char}(V_\lambda) = O_\lambda(x_1, \ldots, x_r, x_1^{-1}, \ldots, x_r^{-1}) \quad , \tag{2.19}$$
where O_λ is an orthogonal Schur function.

- For $O_{2r+1}(\mathbb{C})$:
$$\mathrm{Char}(V_\lambda) = O_\lambda(x_1, \ldots, x_r, x_1^{-1}, \ldots, x_r^{-1}, 1) \quad . \tag{2.20}$$

Symplectic and orthogonal Schur functions are defined by

$$Sp_\mu = D_{\lambda_{-1}[e_2]} s_\mu \quad , \tag{2.21}$$
$$O_\mu = D_{\lambda_{-1}[h_2]} s_\mu \quad , \tag{2.22}$$

where

$$\sigma_z(X) = \sum_{n \geq 0} z^n h_n(X) = \prod_{i \geq 1} (1 - z x_i)^{-1} \quad ,$$

and

$$\lambda_z(X) = \sum_{n \geq 0} z^n e_n(X) = \prod_{i \geq 1} (1 + z x_i) \quad ,$$

are the generating series of complete and elementary symmetric functions, and where $f[g]$ denotes the plethysm of g by f.

The expression of the two plethysms $\lambda_{-1}[e_2]$ and $\lambda_{-1}[h_2]$ are:

$$\lambda_{-1}[e_2] = \prod_{i<j} (1 - x_i x_j) \quad , \tag{2.23}$$

and

$$\lambda_{-1}[h_2] = \prod_{i \geq j} (1 - x_i x_j) \quad . \tag{2.24}$$

The expansions of $\lambda_{-1}[e_2]$ and $\lambda_{-1}[h_2]$ on the Schur basis have been given by Littlewood:

$$\begin{aligned}\lambda_{-1}[e_2] &= 1 - s_{11} + s_{211} - s_{3111} - s_{222} + s_{3221} - s_{3322} + \cdots \quad ,\\ \lambda_{-1}[h_2] &= 1 - s_2 + s_{31} - s_{411} - s_{33} + s_{431} + s_{5111} - s_{61111} + \cdots \quad .\end{aligned}$$

The `Sseries` function computes the truncature to a given rank of these series. For example $\lambda_{-1}[h_2]$ up to partitions of weight 10 will be given by:

```
ACE> Sseries(C,10);

            1 - s[2] + s[3, 1] - s[4, 1, 1] + s[5, 1, 1, 1] - s[4, 4, 2]

              - s[6, 1, 1, 1, 1] - s[3, 3] + s[4, 3, 1]  - s[5, 3, 1, 1]
```

To describe the right-hand side, we need the Frobenius notation for partitions, which is defined as follows. Suppose that the main diagonal of a diagram of λ consists of r cells. If there are a_i cells to the right of the main diagonal in the ith row and b_i above the main diagonal[(2)] in the ith column then the partition is denoted by:

$$\begin{pmatrix} a_1 & a_2 & \dots & a_r \\ b_1 & b_2 & \dots & b_r \end{pmatrix} \quad .$$

r is the *rank* of the partition.

With these notations, $\lambda_{-1}[e_2]$ is the $S-$function series where partitions of Frobenius rank r take the form

$$\begin{pmatrix} a_1 & a_2 & \dots & a_r \\ a_1+1 & a_2+1 & \dots & a_r+1 \end{pmatrix} \quad ,$$

and $\lambda_{-1}[h_2]$ is conjugate to $\lambda_{-1}[e_2]$. In both cases, the sign is given by $(-1)^n$ where n is the number of cells above the main diagonal.

The `Part2Frob` and `Frob2Part` functions allow the use of both notations:

```
ACE> Part2Frob([7,3,2,1]);

                          [[6, 1], [3, 1]]

ACE> Frob2Part([6,3,0],[5,1,0]);

                           [7, 1, 1, 1]
```

Symplectic and orthogonal Schur functions can be expressed in terms of complete symmetric functions

(2) In French notation.

by the following formulas:

$$Sp_\mu = \begin{vmatrix} h_{\mu_1} & h_{\mu_1+1} + h_{\mu_1-1} & \dots & h_{\mu_1-1+n} + h_{\mu_1+1-n} \\ h_{\mu_2-1} & h_{\mu_2} + h_{\mu_2-2} & \dots & h_{\mu_2-2+n} + h_{\mu_2-n} \\ \vdots & & & \vdots \\ h_{\mu_n-n+1} & h_{\mu_n-n+2} + h_{\mu_n-n} & \dots & h_{\mu_n} + h_{\mu_n-2n+2} \end{vmatrix} \tag{2.25}$$

and

$$O_\mu = \begin{vmatrix} H_{\mu_1} & H_{\mu_1+1} + H_{\mu_1-1} & \dots & H_{\mu_1-1+n} + H_{\mu_1-n+1} \\ H_{\mu_2-1} & H_{\mu_2} + h_{\mu_2-2} & \dots & H_{\mu_2-2+n} + H_{\mu_2-n} \\ \vdots & & & \vdots \\ H_{\mu_n-n+1} & H_{\mu_n-n+2} + H_{\mu_n-n} & \dots & H_{\mu_n} + H_{\mu_n-2n+2} \end{vmatrix} \tag{2.26}$$

where $H_i = h_i - h_{i-2}$.

The `SpJtMatrix` and `Sp2s` functions use these identities to compute the expression of symplectic Schur functions on the Schur basis:

```
ACE> SpJtMatrix(Sp[3,2,1,1]);

                           [ h3  h4 + h2  h5 + h1  h6 + 1 ]
                           [                               ]
                           [ h1   h2 + 1     h3       h4   ]
                           [                               ]
                           [  0      1       h1       h2   ]
                           [                               ]
                           [  0      0        1       h1   ]

ACE> Sp2s(Sp[3,2,1,1]);

           s[3] - s[3,2] + s[3,2,1,1] - s[3,1,1] - s[1] + 2 s[2,1]

                - s[2,1,1,1] - s[2,2,1] + s[1,1,1]
```

The `OJtMatrix` and `O2s` functions play the same role for computing the expression of orthogonal Schur functions on the Schur basis.

The inverse series are also known:

$$(\lambda_{-1}[e_2])^{-1} = \sigma_1[e_2] = \prod_{i<j} (1 - x_i x_j)^{-1} \quad , \tag{2.27}$$

and

$$(\lambda_{-1}[h_2])^{-1} = \sigma_1[h_2] = \prod_{i \geq j} (1 - x_i x_j)^{-1} \quad , \tag{2.28}$$

and their expansion on Schur basis is:

$$\begin{aligned} \sigma_1[e_2] &= 1 + s_{11} + s_{22} + s_{1111} + s_{2211} + s_{33} + s_{111111} + s_{2222} + \cdots \quad , \\ \sigma_1[h_2] &= 1 + s_2 + s_{22} + s_{42} + s_{222} + s_6 + s_{42} + s_{422} + \cdots \quad . \end{aligned}$$

$\sigma_1[e_2]$ is the Schur function series where all partitions have an even number of parts and $\sigma_1[h_2]$ is conjugate to $\sigma_1[e_2]$.

Using these series we can compute the $GL_{2n}(\mathbb{C}) \downarrow Sp_{2n}(\mathbb{C})$ and $GL_n(\mathbb{C}) \downarrow O_n(\mathbb{C})$ branching rules:

$$s_\mu = \sum_\nu a_{\mu\nu} Sp_\nu \quad , \tag{2.29}$$

where the coefficients $a_{\mu\nu}$ are defined by:

$$D_{\sigma_1[e_2]} s_\mu = \sum_\nu a_{\mu\nu} s_\nu \quad .$$

Similarly:

$$s_\mu = \sum_\nu b_{\mu\nu} O_\nu \quad , \tag{2.30}$$

where the $b_{\mu\nu}$ are defined by:

$$D_{\sigma_1[h_2]} s_\mu = \sum_\nu b_{\mu\nu} s_\nu \quad .$$

These branching rules are computed by the functions `s2Sp` and `s2O`:

```
ACE> s2Sp(s[2,2,1]);

                    Sp[2,1,1] + Sp[2,2] + Sp[1,1,1] + Sp[2,1] + Sp[1]

ACE> s2O(s[5,3]);

              O[5,3] + O[3,3] + O[4,2] + O[5,1] + 2 O[3,1] + O[4] + O[1,1] + O[2]
```

If the `Sprank` (respectively `Orank`) variable is assigned, the result is expressed in $Sp_{\texttt{Sprank}}$ (respectively $O_{\texttt{Orank}}$).

```
ACE> Sprank:=4:
ACE> s2Sp(s[2,2,1]);

                          Sp[2,2] + Sp[2,1] + Sp[1,0]

ACE> Sprank:=6;
ACE> s2Sp(s[2,2,1]);

              Sp[2,1,1] + Sp[2,2,0] + Sp[1,1,1] + Sp[2,1,0] + Sp[1,0,0]
```

2.8.1 Dimension of representations

For the symplectic group, the dimension of V_λ is given by [7]:

$$\dim(V_\lambda) = \prod_{i=1}^{n} \frac{(\lambda_i + n - i + 1)}{(n - i + 1)} \prod_{k>i}^{n} \frac{(\lambda_i - \lambda_k + k - i)(\lambda_i + \lambda_k + 2n - i - k + 2)}{(k - i)(2n + 2 - i - k)} \quad . \tag{2.31}$$

The `DimRepSp` function has been implemented. The rank of Sp depends on the number of parts of the partition:

```
ACE> DimRepSp([7,4,2]);

                              39312
```

For computing dimensions of irreducible representations of O_n we must distinguish two cases:

- When n is odd ($n = 2k+1$) the dimension of an irreducible representation of O_n can be obtained by the formula:

$$\dim(V_\lambda) = \prod_{r<s}^{k} (\lambda_r - \lambda_s - r + s)(\lambda_r + \lambda_s + n - r - s) \times \frac{\prod_{r=1}^{k} (2\lambda_r + n - 2r)}{(n-2)!(n-4)!\dots(1)!} \quad . \tag{2.32}$$

- When n is even ($n = 2k$), the formula becomes:

$$\dim(V_\lambda) = 2^k \prod_{r<s}^{k} \frac{(\lambda_r - \lambda_s - r + s)(\lambda_r + \lambda_s + n - r - s)}{(n-2)!(n-4)!\dots(2)!} \quad , \tag{2.33}$$

  if $\lambda_k \neq 0$ and half of this if $\lambda_k = 0$.

The `DimRepO` function calculates the dimension of irreducible representations of O_n:

```
ACE> DimRepO(4,[3,2]);

                              24

ACE> DimRepO(5,[3,2]);

                              105
```

2.8.2 Product of characters

A simple formula for product of characters of $Sp_{2n}(\mathbb{C})$ and $O_n(\mathbb{C})$ has been given by Newell (1951) and Littlewood (1958) who have shown that:

$$O_\lambda O_\mu = \sum_\nu c^\nu_{\lambda\mu} O_\nu = D_{\lambda_{-1}[h2]} \sum_\nu c^\nu_{\lambda\mu} s_\nu = D_{\lambda_{-1}[h2]} \sum_\zeta s_{\lambda/\zeta} s_{\mu/\zeta} \quad . \tag{2.34}$$

The case $Sp_\lambda Sp_\mu$ is completely similar:

$$Sp_\lambda Sp_\mu = D_{\lambda_{-1}[e2]} \sum_\nu c^\nu_{\lambda\mu} s_\nu = D_{\lambda_{-1}[e2]} \sum_\zeta s_{\lambda/\zeta} s_{\mu/\zeta} \quad . \tag{2.35}$$

Various proofs of this identity can be found (see e.g. [19] [11] [9]).

Using the `LRrSp` function to compute this identity one can obtain for example:

```
ACE> LRrSp(Sp[3,2],Sp[2,1]);

        Sp[3,2,2,1] + Sp[3,3,1,1] + Sp[3,3,2] + Sp[4,2,1,1] + Sp[4,2,2]

      + 2 Sp[4,3,1] + Sp[4,4] + Sp[5,2,1] + Sp[5,3] + Sp[2,2,1,1] + 4 Sp[3,2,1]

      + 2 Sp[3,3] + Sp[2,2,2] + 3 Sp[4,2] + Sp[3,1,1,1] + 2 Sp[4,1,1] + Sp[5,1]

      + 2 Sp[2,1,1] + 2 Sp[2,2] + 3 Sp[3,1] + Sp[4] + Sp[1, 1] + Sp[2]
```

2.8.3 Freudenthal's multiplicity formula

Freudenthal's formula gives a straightforward way of computing the multiplicity of the weight μ in V_λ, that is, the dimension of the weight space $V_\lambda(\mu)$, by working down recursively from λ. If $d_\mu = \dim V_\lambda(\mu)$, d_μ is explicitly given in terms of multiplicities of weights that are higher than μ:

$$c(\mu)d_\mu = 2\sum_{\alpha\in R^+}\sum_{k\geq 1}(\mu+k\alpha,\alpha)d_{\mu+k\alpha} \quad , \tag{2.36}$$

where $(,)$ is the usual scalar product and R^+ denote the set of positive roots of the considered group.

$$c(\mu) = (\lambda+\rho)^2 - (\mu+\rho)^2 \quad .$$

The `Freudenthal` function uses this formula to compute the multiplicity of weight μ in V_λ:

```
ACE> Freudenthal(C,[5,3,2,0],[4,2,1,1]);

                                  22
```

`PositiveRoots` and `Rho` functions provide extra tools. They are respectively computing the set of positive roots of the Weyl group associated to the group A, B, C or D, and half the sum of the positive roots:

```
PositiveRoots(D,4);

  [[1, -1, 0, 0], [1, 1, 0, 0], [1, 0, -1, 0], [1, 0, 1, 0], [1, 0, 0, -1], [1, 0, 0, 1],

   [0, 1, -1, 0], [0, 1, 1, 0], [0, 1, 0, -1], [0, 1, 0, 1], [0, 0, 1, -1], [0, 0, 1, 1]]

ACE> Rho(D,4);

                              [0, 1, 2, 3]
```

where $\frac{1}{2}e_1 = 3, \ldots, \frac{1}{2}e_4 = 0$.

2.9 Symmetric functions

The basic reference for this section is Macdonald's book [82]. Let $X = \{x_1,\ x_2,\ \ldots\}$ an infinite set of commutative indeterminates, which will be called an *alphabet*. The *elementary symmetric functions* $e_k(X)$ are defined by the generating series:

$$\lambda_t(X) := \sum_{k\geq 0} t^k\, e_k(X) = \prod_{i\geq 1}(1+x_i t) \quad . \tag{2.37}$$

The *complete homogeneous symmetric functions* $h_k(X)$ are defined by:

$$\sigma_t(X) := \sum_{k\geq 0} t^k\, h_k(X) = \prod_{i\geq 1}(1-x_i t)^{-1} \quad . \tag{2.38}$$

The *power sums symmetric functions* $p_k(X)$ are defined by:

$$\psi_t(X) := \sum_{k\geq 1} t^{k-1}\, p_k(X) = \sum_{i\geq 1} x_i\,(1-x_i t)^{-1} \quad . \tag{2.39}$$

These generating series satisfy the following relations:

$$\psi_t(X) = \frac{d}{dt}\log\,\sigma_t(X) = -\frac{d}{dt}\log\,\lambda_{-t}(X) \quad , \tag{2.40}$$

$$\frac{d}{dt}\,\sigma_t(X) = \sigma_t(X)\ \psi_t(X) \quad , \tag{2.41}$$

$$-\frac{d}{dt}\,\lambda_{-t}(X) = \psi_t(X)\ \lambda_{-t}(X) \quad . \tag{2.42}$$

Formula (2.42) is known as Newton's formula. The so-called fundamental theorem of the theory of symmetric functions states that the $e_k(X)$ are algebraically independent. Therefore, any formal power series $f(t) = 1 + \sum_{k\geq 1} a_k\, t^k$ may be considered as the specialization of the series $\lambda_t(X)$ to a virtual set of arguments A.

We denote by *Sym* the algebra of symmetric functions, *i.e.* the algebra generated over $\mathbb{Q}$ by the elementary functions. It is a graded algebra for the weight function $w(e_k) = k$, and the dimension of its homogeneous component of weight n, denoted by Sym_n, is equal to $p(n)$, the number of partitions of n. A *partition* is a finite non-increasing sequence of positive integers, $\lambda = (\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r)$. We shall also write $\lambda = (1^{\alpha_1}\, 2^{\alpha_2}\, \ldots)$, α_m being the number of parts λ_k which are equal to m. The *weight* of λ is $|\lambda| = \sum_k \lambda_k$ and its *length* is the number of its (nonzero) parts $\ell(\lambda) = r$.

For a partition λ, one sets:

$$p_\lambda = p_1^{\alpha_1}\, p_2^{\alpha_2}\, \ldots \quad , \quad e_\lambda = e_1^{\alpha_1}\, e_2^{\alpha_2}\, \ldots \quad , \quad h_\lambda = h_1^{\alpha_1}\, h_2^{\alpha_2}\, \ldots \quad .$$

These are linear bases of *Sym* (`Top`, `Toe`, `Toh`):

```
ACE> Toe(h4*p2^2);

                 2             2        4             2          3             5
        - 4 e4 e2  + 4 e4 e2 e1  - e4 e1  + 8 e3 e1 e2  - 8 e3 e1  e2 + 2 e3 e1

                 4        3   2        2   4          6     8
           + 4 e2  - 16 e2  e1  + 17 e2  e1  - 7 e2 e1  + e1
```

For $\lambda \in \mathbb{Z}^r$, not necessarily a partition, the *Schur function* s_λ is defined by:

$$s_\lambda = \det\left(h_{\lambda_i - i + j}\right)_{1\leq i,j\leq r} \quad , \tag{2.43}$$

where $h_j = 0$ for $j < 0$. The Schur functions indexed by partitions form a $\mathbb{Z}$-basis of *Sym* (`Tos`), and one usually endows *Sym* with a scalar product $\langle\cdot,\cdot\rangle$ for which this basis is orthonormal. The p_λ form then an orthogonal $\mathbb{Q}$-basis of *Sym*, with $z_\lambda := \langle p_\lambda, p_\lambda\rangle = 1^{\alpha_1}\,\alpha_1!\,2^{\alpha_2}\,\alpha_2!\,\ldots$ (`Zee`, `ScalarSf`):

```
ACE> Tos(s[4,3,2,2,1]*h3);

          s[4, 3, 3, 2, 2, 1] + s[4, 4, 2, 2, 2, 1] + s[4, 4, 3, 2, 1, 1]

        + s[4, 4, 3, 2, 2] + s[5, 3, 2, 2, 2, 1] + s[5, 3, 3, 2, 1, 1]

        + s[5, 3, 3, 2, 2] + s[5, 4, 2, 2, 1, 1] + s[5, 4, 2, 2, 2]

        + s[5, 4, 3, 2, 1] + s[6, 3, 2, 2, 1, 1] + s[6, 3, 2, 2, 2]

        + s[6, 3, 3, 2, 1] + s[6, 4, 2, 2, 1] + s[7, 3, 2, 2, 1]

ACE> ScalarSf(p3^2*p1, p3^2*p1, 'p', 'p');

                              18
ACE> Zee([3,3,1]);

                              18
```

Thus, for a partition λ of weigth n, $n!/\langle p_\lambda, p_\lambda\rangle$ is the cardinality of the conjugacy class of $\mathfrak{S}_n$ whose elements have α_k cycles of length k for all $k \in [1,n]$. A permutation σ in this class will be said *of type* λ (`Perm2Cycle`, `Perm2CycleType`), and we shall write $T(\sigma) = \lambda$.

These definitions are motivated by the following classical results of Frobenius. Let $CF(\mathfrak{S}_n)$ be the ring of central functions of the symmetric group $\mathfrak{S}_n$. The *Frobenius characteristic map* $\mathcal{F}$: $CF(\mathfrak{S}_n) \longrightarrow Sym_n$ associates (`Sf2Char`) with any central function ξ the symmetric function:

$$\mathcal{F}(\xi) = \frac{1}{n!}\sum_{\sigma\in\mathfrak{S}_n} \xi(\sigma)\; p_{T(\sigma)} = \sum_{|\lambda|=n} \xi(\lambda)\;\frac{p_\lambda}{z_\lambda} \quad ,$$

where $\xi(\lambda)$ is the common value of the $\xi(\sigma)$ for all σ such that $T(\sigma) = \lambda$:

```
ACE> Sf2Char(p3*s[3] - e4);

          - cl[2, 2] + 6 cl[3, 3] + 3 cl[3, 1, 1, 1] + cl[2, 1, 1] + cl[4]

             - cl[1, 1, 1, 1] + 3 cl[3, 2, 1] - cl[3, 1]
```

in which `cl`[*part*], where *part* is a partition, is the class function on the symmetric group that is evaluated to 1 for permutations with cycle-type *part*, and 0 otherwise.

We can also consider $\mathcal{F}$ as a map from the representation ring $R(\mathfrak{S}_n)$ to Sym_n by setting $\mathcal{F}([\rho]) = \mathcal{F}(\chi_\rho)$, where $[\rho]$ denotes the equivalence class of a representation ρ. Glueing these maps together, one has a linear map:

$$\mathcal{F} \,:\, R := \bigoplus_{n\geq 0} R(\mathfrak{S}_n) \longrightarrow Sym \quad ,$$

which turns out to be an isomorphism of graded rings. We denote by $[\lambda]$ the class of the irreducible representation of $\mathfrak{S}_n$ associated with the partition λ, and by χ^λ its character. We have then $\mathcal{F}(\chi^\lambda) = s_\lambda$.

The product $*$, defined on the homogeneous component Sym_n by:

$$\mathcal{F}([\rho] \otimes [\eta]) = \mathcal{F}(\chi_\rho \chi_\eta) = \mathcal{F}([\rho]) * \mathcal{F}([\eta]) \quad , \tag{2.44}$$

and extended to Sym by defining the product of two homogeneous functions of different weights to be zero, is called the *internal product* (`ITensor`):

```
ACE> ITensor(p3*p3*p2*p1*p1, p3*p3*p2*p1*p1);

                                   2       2
                               72 p3   p2 p1
```

One can identify the tensor product $Sym \otimes Sym$ with the algebra $Sym(X, Y)$ of polynomials which are separately symmetric in two infinite disjoint sets of indeterminates X and Y, the correspondence being given by $F \otimes G \longmapsto F(X)\, G(Y)$. Denoting by $X + Y$ the disjoint union of X and Y, one can then define a comultiplication Δ on Sym by setting $\Delta(F) = F(X + Y)$. This comultiplication endows Sym with the structure of a self-dual Hopf algebra, which is very useful for calculations involving characters of symmetric groups. The basic formulas are:

$$(F_1 F_2 \dots F_r) * G = \mu_r[(F_1 \otimes F_2 \otimes \cdots \otimes F_r) * \Delta^r G] \quad , \tag{2.45}$$

where μ_r denotes the r-fold ordinary multiplication and Δ^r the iterated coproduct, and:

$$\Delta^r(F * G) = \Delta^r(F) * \Delta^r(G) \quad . \tag{2.46}$$

The symmetric functions of the virtual alphabet $(-X)$ are defined by the generating series:

$$\lambda_t(-X) := \sigma_{-t}(X) = [\lambda_t(X)]^{-1} \quad , \tag{2.47}$$

and one can more generally consider *differences* of alphabets. The symmetric functions of $X - Y$ are given by the generating series:

$$\lambda_t(X - Y) := \lambda_t(X)\ \lambda_t(-Y) = \lambda_t(X)\ \sigma_{-t}(Y) \quad . \tag{2.48}$$

In particular, $p_k(X - Y) = p_k(X) - p_k(Y)$.

There is another coproduct δ on Sym, which is obtained by considering products instead of sums, that is:

$$\delta(F) = F(XY) \quad . \tag{2.49}$$

One can check that its adjoint is the internal product:

$$\langle \delta F\, ,\, P \otimes Q \rangle = \langle F\, ,\, P * Q \rangle \quad . \tag{2.50}$$

This equation can be used to give an intrinsic definition of the internal product, *i.e.* without any reference to characters of the symmetric group. Also, one can see that two bases $(U_\lambda), (V_\mu)$ of Sym are adjoint to each other if and only if

$$\sigma_1(XY) = \sum_I U_\lambda(X)\, V_\lambda(Y) \quad . \tag{2.51}$$

For example, writing $\sigma_1(XY) = \prod_i \sigma_{x_i}(Y)$ and expanding the product, one obtains that the adjoint basis of h_λ is formed by the *monomial functions* m_λ (`Tom`) .

2.10 Noncommutative symmetric functions

Starting from the quasi-determinants of Gelfand and Retakh [45, 46], a noncommutative theory of symmetric functions [44] has been developed in such a way that most of the classical applications can be lifted to the noncommutative case. In particular, the character theory of the symmetric group has a natural noncommutative analogue, in which the role of the character ring is played by Solomon's descent algebra [58]. The main applications of this correspondence are connected with Lie series and free Lie algebras (*cf.* [43, 56]). Also, the algebra of noncommutative symmetric functions, together with its dual Hopf algebra, which is Gessel's algebra of quasi-symmetric functions [47, 54] encode the representation theory of the 0-Hecke algebra of the symmetric group [42].

The *algebra of formal noncommutative symmetric functions* can be defined as the free associative algebra $\mathbf{Sym}=\mathbb{C}\langle\Lambda_1,\Lambda_2,\ldots\rangle$ generated by an infinite sequence of noncommuting indeterminates Λ_k, called *elementary* functions. It is convenient to set $\Lambda_0 := 1$. Let t be another indeterminate, commuting with the Λ_k. One introduces the generating series:

$$\lambda(t) = \sum_{k\geq 0} \Lambda_k\, t^k \quad , \tag{2.52}$$

$$\sigma(t) = \sum_{k\geq 0} S_k\, t^k = \lambda(-t)^{-1} \quad , \tag{2.53}$$

$$\Phi(t) = \sum_{k\geq 1} \frac{1}{k}\Phi_k\, t^k = \log\sigma(t) \quad . \tag{2.54}$$

The S_k are called *complete symmetric* functions, and the Φ_k are the *power sums of the second kind.* The *power sums of the first kind*, denoted by Ψ_k are the coefficients of the formal series:

$$\psi(t) = \sum_{k\geq 1} \Psi_k\, t^{k-1} \quad , \tag{2.55}$$

defined by the equation:

$$\psi(t) = \lambda(-t)\frac{d}{dt}\sigma(t) \quad . \tag{2.56}$$

The identities (1)-(2.56), between formal power series imply relations between their coefficients. We have:

$$\sum_{k=0}^{n}(-1)^k S_{n-k}\Lambda_k = \sum_{k=0}^{n}(-1)^k \Lambda_{n-k}S_k = 0 \quad ,$$
$$\sum_{k=0}^{n-1} S_k\Psi_{n-k} = nS_n \quad ,$$
$$\sum_{k=1}^{n}(-1)^{k-1}\Psi_k\Lambda_{n-k} = n\Lambda_n \quad .$$

In the commutative case, those formulas are known as Wronski and Newton formulas.

The algebra $\mathbf{Sym}$ is graded by the weight function w defined by $w(\Lambda_k) = k$, and its homogeneous component of weight n is denoted by $\mathbf{Sym}_n$. A composition of n is a vector $I = (i_1, \ldots, i_k)$ of positive integers such that $i_1 + \cdots + i_k = n$.

If (F_n) is a sequence of noncommutative symmetric functions such that $F_n \in \mathbf{Sym}_n$, we set for any composition $I = (i_1, \ldots, i_k)$ of n:

$$F^I := F_{i_1}F_{i_2}\ldots F_{i_k} \quad . \tag{2.57}$$

Then, (Λ^I), (S^I), (Φ^I) and (Ψ^I) are homogeneous bases of **Sym**.

We recall general results on composition. A composition of n naturally encodes a *ribbon* (or *skew-hook*) with n boxes. A ribbon is a connected skew Young diagram containing no 2×2 block of boxes. For instance, the skew diagram λ/μ with $\lambda = (4,4,3,3)$ and $\mu = (3,2,2)$ is a ribbon, that corresponds to the composition $(3,1,2,\,1)$, *i.e.* the number of boxes per row (reading from top to bottom).

The set of all compositions of a given integer n can be computed by the function The length $\ell(I)$ of the composition I is the number k of its parts and the weight of I is the sum $|I| = \sum i_j$ of its parts. We denote by $\bar{I}$ the mirror image of the composition I, *i.e.* the composition obtained by reading I from right to left. We can also define the *conjugate* composition $I^{\sim}$ of I as the composition whose diagram is the conjugate (in the sense of skew diagrams) of the diagram of I. For instance, the conjugate composition of $(3,1,2,1)$ is $(2,3,1,1)$.

We consider two orderings on the set of all compositions of an integer n. The first one is the *reverse refinement* order, denoted $\preceq$, and defined by $I \preceq J$ if and only if J is *finer* than I. For example, the compositions J of 5 such that $J \preceq (2,1,2)$ are $(2,1,2)$, $(3,2)$, $(2,3)$ and (5). The second one is the *reverse lexicographic* ordering, denoted by $\leq$. For instance, $(6) \leq (5,1) \leq (4,2) \leq (4,1,1)$.

One also introduces the basis, (R_I) of *ribbon Schur* functions, originally defined in terms of quasi-determinants in [44]. Ribbon Schur functions can also be defined by one of the following two equivalent systems of equations:

$$S^I = \sum_{J \preceq I} R_J, R_I = \sum_{J \preceq I} (-1)^{\ell(I)-\ell(J)} S^J \quad . \tag{2.58}$$

Multiplication of ribbon Schur functions is given by the following formula whose commutative version is due to MacMahon [52]:

$$R_I R_J = R_{I \triangleright J} + R_{I.J} \quad ,$$

with $I = (i_1,\ldots,i_r)$, $J = (j_1,\ldots,j_s)$ and where $I \triangleright J = (i_1,\ldots,i_{r-1},i_r + j_1, j_2,\ldots,j_s)$ and $I.J = (i_1,\ldots,i_r,j_1,\,\ldots,j_s)$.

2.10.1 Change of basis

Transition matrices between the various bases of **Sym**, as well as functions `To`X (X={`S`, `L`, `R`, `Ph`, `Ps`}) giving the expansion of any element of **Sym** in one of these bases have been implemented. We use algorithms described in [44].

One defines $M(F,G)_n$ to be the transition matrix from the basis (F_I) to the basis (G_J), I and J being compositions of n. This matrix is indexed by compositions of n in reverse lexicographic order. Our convention is that the *row* indexed by I contains the components of F_I in the basis (G_J). Consider the simple example of the matrices $M(S,\Lambda)$ which are easily obtained by means of the relation:

$$S^I = \sum_{I \preceq J} (-1)^{l(J)-|I|} \Lambda^J \quad .$$

Computations for all compositions of 3 give:

$$\begin{aligned} S^3 &= \Lambda^3 - \Lambda^{21} - \Lambda^{12} + \Lambda^{111} \quad , \qquad S^{21} = -\Lambda^{21} + \Lambda^{111} \quad , \\ S^{12} &= -\Lambda^{12} + \Lambda^{111} \quad , \qquad S^{111} = \Lambda^{111} \quad . \end{aligned}$$

Therefore, $M(S,\Lambda)_3$ can be obtained by the following *Maple* instruction:

```
ACE> L2SMat(3);

                              [ 1  -1  -1  1 ]
                              [              ]
                              [ 0  -1   0  1 ]
                              [              ]
                              [ 0   0  -1  1 ]
                              [              ]
                              [ 0   0   0  1 ]
```

`To`X functions can handle complicated arguments. For instance, one can do:

```
ACE> ToR((1-q)^4*S[2,1]+t*L[3],noexpand);

                     4              4
              (1 - q)  R[3] + (1 - q)  R[2, 1] + t R[1, 1, 1]
```

The standard product of **Sym** (viewed as a free associative algebra) convolution product on graded endomorphisms corresponds to the convolution product on graded endomorphisms of a tensor algebra [56]. In *Maple*, the multiplication operator `*` is commutative, hence one has to represent this product by means of the infix operator `&*`.

2.10.2 Descent algebras

We denote by $\mathrm{Des}(\sigma)$ the descent set of a permutation σ. For each subset $A \subseteq \{1, \ldots, n-1\}$ we consider the element:

$$D_{=A} = \sum_{\mathrm{Des}\,(\sigma)=\mathrm{A}} \sigma \qquad \in \mathbb{Q}[S_n] \quad .$$

The elements $D_{=A}$ are linearly independent. Solomon shown that the subspace of $\mathbb{Q}[S_n]$ generated by the elements $D_{=A}$ is a subalgebra of $\mathbb{Q}[S_n]$ ([43],[58]). In fact, there exist nonnegative integers d_{AB}^C such that:

$$D_{=A} D_{=B} = \sum_C d_{AB}^C D_{=C} \quad ,$$

for A, $B \subseteq \{1, \ldots, n-1\}$. This subalgebra is called the *descent algebra* of S_n and is denoted by $\sum_n$. We will write $\sum = \bigoplus_{n\geq 0} \sum_n$ (where $\sum_0 := \mathbb{Q}$) the direct sum of all descent algebras. A subset of $\{1, \ldots, n-1\}$ can be encoded by a composition of n. Let $A = \{a_1 < a_2 < \cdots < a_k\}$ of be a subset of$\{1, \ldots, n-1\}$, then we define:

$$\begin{aligned} c(A) &= I = (i_1, \ldots, i_{k+1}) \\ &= (a_1, a_2 - a_1, \ldots, a_k - a_{k-1}, n - a_k) \ . \end{aligned}$$

We can associate a ribbon to a permutation. For example, starting from the permutation (52341), with descent set $\{1, 4\}$, we build the ribbon:

$$\begin{array}{ccc} \cline{1-1} \multicolumn{1}{|c|}{5} & & \\ \hline \multicolumn{1}{|c|}{2} & \multicolumn{1}{c|}{3} & \multicolumn{1}{c|}{4} \\ \hline & & \multicolumn{1}{|c|}{1} \\ \cline{3-3} \end{array}$$

which corresponds to the composition $I = (1, 3, 1)$. Then the permutation has descent set $c^{-1}(I)$. In this way, one defines a linear isomorphism $\alpha : \sum_n \longrightarrow \mathbf{Sym}_n$ by:

$$\alpha(D_{=A}) = R_{c(A)} \quad .$$

Thus, by analogy with the product on $\sum_n$, one defines another product on $\mathbf{Sym}_n$, denoted $*$, by:

$$F * G = \alpha(\alpha^{-1}(G)\alpha^{-1}(F)) \quad ,$$

for F, $G \in \mathbf{Sym}_n$. One extends this product to $\mathbf{Sym}$ by setting $F * G = 0$ if F and G are homogeneous of different weights. This product $*$ is called the *internal product* of $\mathbf{Sym}$. It is a noncommutative analog of the usual internal product, which corresponds to tensor products of representations of the symmetric group.

In `NCSF`, the internal product is computed by the function `Internal`, similar to Stembridge's `itensor`. The evaluation is based on the following property: given two compositions $I = (i_1, i_2, \ldots, i_p)$ and $J = (j_1, j_2, \ldots, j_q)$, one has:

$$S^I * S^J = \sum_{M \in \mathrm{Mat}(I,J)} S^M \quad ,$$

where $\mathrm{Mat}(I,J)$ denotes the set of matrices of nonnegative integers $M = (m_{ij})$ of size $p \times q$ such that $\sum_s m_{rs} = i_r$ and $\sum_r m_{rs} = j_s$ for $r \in [1, p]$ and $s \in [1, q]$. Also, the product S^M is obtained by reading M from left to right and top to bottom. From this relation, we can see that if $|I| \neq |J|$ then $S^I * S^J = 0$. Like the ToX functions, `Internal` accepts as argument quite complicated expressions. For instance:

```
ACE> Internal(q*L[2,2],Ph[4]+t*L[2,2]);

                          2 q t S[2, 2] + q t S[1, 1, 1, 1]
```

2.10.3 Lie idempotents

A *Lie idempotent* is an idempotent of the algebra of the symmetric group which projects a homogeneous component of the free associative algebra onto the free Lie algebra (see *e.g.* [56]). It turns out that most interesting Lie idempotents belong to Solomon's algebra, which allows one to interpret them as noncommutative symmetric functions. For example, it has been shown by Klyachko [48] that for a primitive n-th root of unity ε, the element:

$$\kappa_n = \frac{1}{n} \sum_{\sigma \in S_n} \varepsilon^{\mathrm{maj}(\sigma)} \sigma \quad , \qquad (2.59)$$

is a Lie idempotent, where maj(σ) denotes the *major index* of a permutation. By definition, for $\sigma \in S_n$ with Des$(\sigma) = \{d_1, \ldots, d_r\}$, maj$(\sigma) = d_1 + \cdots + d_r$. To prove the previous assertion, one can rewrite (2.59) in terms of descent sets. Hence:

$$\begin{aligned}\kappa_n &= \frac{1}{n} \sum_{A \subseteq \{1,\ldots,n-1\}} \varepsilon^{\mathrm{maj}(A)} \Big(\sum_{\mathrm{Des}\,(\sigma)=A} \sigma \Big) \\ &= \frac{1}{n} \sum_{A \subseteq \{1,\ldots,n-1\}} \varepsilon^{\mathrm{maj}(A)} D_{=A} \quad .\end{aligned}$$

It follows that:

$$\alpha(\kappa_n) = \frac{1}{n} \sum_{|I|=n} \varepsilon^{\mathrm{maj}(I)} R_I \quad .$$

Let us check these properties for κ_4. The corresponding symmetric function is obtained by the following *Maple* sequence:

```
ACE> C:=ListCompo(4):
ACE> K := 0:
ACE> for x in C do K:=K+q^MajorIndex(x)*R[op(x)] od:
ACE> K:=subs(q=I, K)/4;

          K := 1/4 R[4] - 1/4 I R[3, 1] - 1/4 R[2, 2] + 1/4 I R[2, 1, 1] + 1/4 I R[1, 3]

             + 1/4 R[1, 2, 1] - 1/4 I R[1, 1, 2] - 1/4 R[1, 1, 1, 1]
```

We check that it is actually an idempotent:

```
ACE> Internal(K,K) - ToS(K);

                                   0
```

We have to compare with `ToS(K)` because the output of `Internal` is given in the basis (S^I). To see that κ_4 projects onto the free Lie algebra, one may compute its left and right products with a known Lie idempotent, for example Dynkin's element $\theta_4/4 := \frac{1}{4}[[[1,2],3],4]$. We recall that θ_n is defined by $\theta_n = [\theta_{n-1}, n]$, which can be rewritten in terms of descent sets by:

$$\theta_n = \sum_{k=0}^{n-1} (-1)^k D_{=\{1,2,\ldots,k\}} \quad .$$

Then applying α to θ_n, we find

$$\alpha(\theta_n) = \sum_{k=0}^{n-1} (-1)^k R_{1^k,n-k} \quad .$$

We verify $\theta_4/4$ is an idempotent.

```
ACE> [seq((-1)^k*R[1$k,4-k], k=0..3)]:
ACE> T:=convert(", `+`)/4;

                 T := 1/4 R[4] - 1/4 R[1, 3] + 1/4 R[1, 1, 2] - 1/4 R[1, 1, 1, 1]

ACE> Internal(T,T)-ToS(T);

                                              0
```

We compute the left and right products of κ_4 and θ_4:

```
ACE> Internal(K,T)-ToS(T);

                                              0
ACE> Internal(T,K)-ToS(K);

                                              0
```

The symmetric functions of the virtual alphabet $A/(1-q)$ are defined by the equation:

$$S_n\left(\frac{A}{1-q}\right) = \frac{K_n(q)}{(q)_n} \quad ,$$

where A can itself be a virtual alphabet, $(q)_n = (1-q)\dots(1-q^n)$ and $K_n(q)$ is given by:

$$K_n(q) = \sum_{|I|=n} q^{\mathrm{maj}\,(I)} R_I \quad .$$

In the package, this transformation is performed by the function `SpDirect`:

```
ACE> SpDirect(S[3], q):
ACE> ToR((1-q)*(1-q^2)*(1-q^3)*", collect):
ACE> map(simplify,");

                                     3                    2
                         R[1, 1, 1] q  + R[1, 2] q + R[2, 1] q  + R[3]
```

The inverse transformation is given by the function `SpInverse`. For example, $\Theta_n(q) := S_n((1-q)A)/(1-q)$ corresponds in the descent algebra to the q-bracket operator. For $n = 4$:

```
ACE> SpInverse(S[4]/(1-q), q, collect);

                                    2                3
                   R[4] - q R[1, 3] + q  R[1, 1, 2] - q  R[1, 1, 1, 1]
```

One can show that the element:

$$\varphi_n(q) = \frac{1-q^n}{n}\Psi_n\left(\frac{A}{1-q}\right) \quad ,$$

is a Lie idempotent [41, 49], interpolating between several known ones:

```
ACE> SpDirect(Ps[4],q,collect):
ACE> ToR((1-q^4)*"/4,noexpand,collect):
ACE> P:=map(simplify,"):
ACE> subs(q=1, P):
ACE> ToPh(");

                                  1/4 Ph[4]

ACE> subs(q=I, P);

  1/4 R[4] - 1/4 I R[3, 1] - 1/4 R[2, 2] + 1/4 I R[2, 1, 1] + 1/4 I R[1, 3]

        + 1/4 R[1, 2, 1] - 1/4 I R[1, 1, 2] - 1/4 R[1, 1, 1, 1]
```

Thus, for $q = 1$ we recover Dynkin's idempotent, and for $q = i$, one obtains Klyachko's element.

Following a recent work by Malvenuto and Reutenauer [54] the dual bialgebra (in the graded sense) of **Sym**, denoted by **Sym***, can be identified with another interesting generalization of symmetric functions, that is, the algebra **Qsym** of *quasi-symmetric* functions, whose definition is due to Gessel [47].

2.10.4 Quasi-symmetric functions

Let $X = \{x_1 < x_2 < x_3 < \cdots\}$ be an infinite totally ordered set of commuting variables. We consider the ring $\mathbb{Q}[[X]]$ of formal power series in these variables. A formal series $F = F(x)$ of $\mathbb{Q}[[X]]$ is said to be quasi-symmetric if for any sequences $y_1 < \cdots < y_k$ and $z_1 < \cdots < z_k$ of X, and any positive integers $a_1, a_2, \ldots, a_k$, the monomials $y_1^{a_1} \ldots y_k^{a_k}$ and $z_1^{a_1} \ldots z_k^{a_k}$ have the same coefficient in F. Gessel defines a natural basis of **Qsym**, the *quasi-monomial functions* (M_I) by:

$$M_I = \sum_{y_1 < y_2 < \cdots < y_k} y_1^{i_1} y_2^{i_2} \ldots y_k^{i_k} \quad ,$$

where $I = (i_1, i_2, \ldots, i_k)$ is a composition of n. Another interesting basis, also introduced by Gessel, is constituted by the functions (F_I) defined by

$$F_I = \sum_{I \preceq J} M_J \quad ,$$

which will be called the *quasi-ribbon* functions.

2.10.5 Pairing between Sym and Qsym

The pairing between **Qsym** and **Sym** is defined by:

$$\langle M_I, S^I \rangle = \delta_{IJ} \quad . \tag{2.60}$$

In the sequel, the normalized dual basis of *power sums of the first kind* is denoted by ${}^Q\Psi$, of *power sums of the second kind* by ${}^Q\Phi$, of *elementary* functions by E. To obtain simple coefficients, we set:

$$\left.\begin{array}{r} \langle {}^Q\Phi_I, \dfrac{\Phi^J}{\Pi(J)} \rangle \\ \langle {}^Q\Psi_I, \dfrac{\Psi^J}{\Pi_u(J)} \rangle \\ \langle F_I, R_J \rangle \\ \langle E_I, \Lambda^J \rangle \end{array}\right\} = \delta_{IJ} \quad , \tag{2.61}$$

where $J = (j_1, \ldots, j_k)$, $\Pi(J) = j_1 \ldots j_k$ and $\Pi_u(J) = j_1(j_1 + j_2) \ldots (j_1 + j_2 + \cdots + j_k)$.

As for **Sym**, we define $M(X, Y)$ to be the transition matrix from basis X to basis Y with the same conventions as previously. For instance, the matrix $M(F, M)_3$ can be computed by the following *Maple* instruction:

```
ACE> F2MMat(3);

                         [ 1  1  1  1 ]
                         [            ]
                         [ 0  1  0  1 ]
                         [            ]
                         [ 0  0  1  1 ]
                         [            ]
                         [ 0  0  0  1 ]

ACE> M2FMat(3);

                      [ 1  -1  -1   1 ]
                      [               ]
                      [ 0   1   0  -1 ]
                      [               ]
                      [ 0   0   1  -1 ]
                      [               ]
                      [ 0   0   0   1 ]
```

On **Qsym**, the usual product can be computed as follows (cf. [53]): given $|I| = m$ and $|J| = n$, one considers $u \in S_{1..m}$ and $v \in S_{m+1..m+n}$ with $c(u) = I$ and $c(v) = J$, let:

$$u \sqcup\!\sqcup\, v = \sum_{w \in S_{1..m+n}} c_w \, w \quad ,$$

where $\sqcup\!\sqcup$ denotes the *shuffle*. Then:

$$F_I \cdot F_J = \sum_{w \in S_{1..m+n}} c_w \, F_{c(w)} \quad .$$

The package provides a set of functions *ToX* to compute on **Qsym**. For instance, one has:

```
ACE> ToM(F[2,1] &* E[2]);

            - M[2, 3] - M[1, 1, 3] - 2 M[2, 2, 1] - M[1, 1, 2, 1] - M[2, 1, 2]

            - M[1, 1, 1, 2] - M[2, 1, 1, 1] - M[4, 1] - M[3, 1, 1] - M[1, 3, 1]

            - M[1, 2, 1, 1]

ACE> ToF(F[3] &* F[1,2,1]);

            F[1, 1, 4, 1] + F[4, 2, 1] + F[3, 3, 1] + F[3, 1, 2, 1] + F[3, 2, 2]

          + F[3, 2, 1, 1] + F[2, 4, 1] + F[2, 3, 2] + F[2, 3, 1, 1] + F[2, 1, 3, 1]

          + F[2, 1, 2, 2] + F[2, 1, 2, 1, 1] + F[2, 2, 3] + 2 F[2, 2, 2, 1]

          + F[2, 2, 1, 2] + F[1, 5, 1] + F[1, 4, 2] + F[1, 4, 1, 1] + F[1, 2, 2, 1, 1]

          + F[1, 3, 3] + 2 F[1, 3, 2, 1] + F[1, 3, 1, 2] + F[1, 1, 3, 2]

          + F[1, 1, 3, 1, 1] + F[1, 1, 2, 3] + F[1, 1, 2, 2, 1] + F[1, 1, 2, 1, 2]

          + F[1, 2, 4] + 2 F[1, 2, 2, 2] + F[1, 2, 1, 3] + 2 F[1, 2, 3, 1]
```

From the definition of quasi-symmetric functions, it follows that a symmetric function is in particular a quasi-symmetric function. According to Gessel [47], if f is a symmetric function, then:

$$f = \sum_I \langle f, r_I \rangle F_I \quad ,$$

where r_I is the commutative image of the ribbon Schur function R_I, and $\langle\ ,\ \rangle$ is the scalar product of the algebra of symmetric functions. This scalar product is defined by:

$$\langle s_\lambda, s_\mu \rangle = \delta_{\lambda\mu} \quad ,$$

in which $r_I = s_{\lambda/\mu}$. For example, one can compute the development of s_{32} on the (F_I) basis by the following instruction:

```
ACE> Sf2Qsf(s[3,2]);

                 F[3, 2] + F[2, 3] + F[2, 2, 1] + F[1, 3, 1] + F[1, 2, 2]
```

The $p(n) \times 2^{n-1}$ matrix whose rows give the coefficients of the Schur functions s_λ on the basis (F_I) can be interpreted as the decomposition matrix of the 0-Hecke algebra $H_n(0)$ (cf. [39] and [42]).

2.11 Hyperoctahedral groups

B_n (BN)) is the Coxeter group generated by $s_0, \ldots s_{n-1}$. The generators $s_1, \ldots, s_{n-1}$ are those of the symmetric group $\mathfrak{S}_n$ whereas s_0 satisfies the following relations:

$$\begin{cases} s_0^2 = 1 & , \\ s_0 s_1 s_0 s_1 = s_1 s_0 s_1 s_0 & , \\ s_0 s_i = s_i s_0 \ , & i > 1 \quad . \end{cases}$$

A code (for B_n) is a vector of length n, componentwise smaller than the vector $[1, 3, 5, \ldots, 2n-1]$. The maximal element of B_n (`BnLastCode`) corresponds to the code $[1, 3, 5, \ldots, 2n-1]$. It has a reduced decomposition:

$$(0)(101)(21012)\ldots(n-1\ldots 0\ldots n-1) \quad ,$$

factorized into canonical factors of the type

$$(k\ldots 101\ldots k) \quad ,$$

in which i stands for the generator s_i.

Any element w of B_n has a canonical reduced decomposition which is a product of left factors of the above canonical factors, and characterized by their lengths $c_1, \ldots, c_n$. This canonical reduced decomposition (`BnCode2Rd`) is the lexicographically minimum reduced decomposition of w. For instance, the reduced decomposition:

$$(0)(10)()(32101) \quad ,$$

corresponds to the code $[1, 2, 0, 5]$. The length of an element w (`BnCode2Length`) is exactly the number of generators needed to express this element as a product of minimal length of generators. This is also the sum of all entries of its code. For example, the element whose code is $[1, 2, 0, 5]$ has a length equal to 8 and this can be checked by counting the number of generators in one of its reduced decomposition, that is $s_0 s_1 s_0 s_3 s_2 s_1 s_0 s_1$.

The decomposition of an element w as a product of simple generators is not unique. The list of all reduced decompositions of an element given by its code is obtainable by the function `BnCode2ListRd` whereas the number of such expressions is given by the function `BnCode2NbRd`:

```
ACE> BnCode2ListRd([1,3,3]);

        [[1, 0, 1, 2, 0, 1, 0], [1, 0, 1, 0, 2, 1, 0], [0, 1, 0, 1, 2, 1, 0],

         [0, 1, 2, 0, 1, 2, 0], [0, 1, 0, 2, 1, 2, 0], [0, 1, 2, 0, 1, 0, 2],

         [0, 1, 0, 2, 1, 0, 2]]

ACE> BnCode2NbRd([1,3,5,7]);

                                   24024
```

The function `BnListCode` returns the $2^n n!$ codes of the elements of B_n. Moreover, several functions are available to generate step by step all codes of B_n: `BnFirstCode`(n) returns the code $[0, 0, \ldots, 0]$ that is a vector of length n, `BnLastCode`(n) returns the code $[1, 3, \ldots, 2n-1]$, `BnPrevCode` and `BnNextCode` respectively return the previous and the next code of a given code, in the lexicographic order.

Given a code c, the command `BnInterval`(c) returns the list of all codes which are below c with respect to the Bruhat order:

```
ACE> BnInterval([1,3,3]);

        [[1, 2, 4], [1, 1, 4], [1, 2, 3], [1, 2, 2], [1, 1, 3], [1, 1, 2], [1, 2, 1],

         [1, 1, 1], [0, 2, 4], [0, 1, 4], [0, 2, 3], [0, 2, 2], [0, 1, 3], [0, 1, 2],

         [0, 2, 1], [0, 1, 1], [1, 0, 4], [0, 0, 4], [1, 0, 3], [1, 0, 2], [0, 0, 3],

         [0, 0, 2], [1, 0, 1], [0, 0, 1], [1, 3, 0], [0, 3, 0], [1, 2, 0], [1, 1, 0],

         [0, 2, 0], [0, 1, 0], [1, 0, 0], [0, 0, 0]]
```

Group operations are defined on the elements given by their codes. The function `BnMultCode` is the multiplication of 2 elements of B_n whereas `BnInvCode` inverts a given element.

There are other descriptions of B_n. We give two of them. One can embed B_n into the symmetric group $\mathfrak{S}_{2n}$. It amounts to interpret the generators `BnTranspoPalin` as follows:

$$\begin{cases} s_0 & = & t_n \quad , \\ s_1 & = & t_{n-1}t_{n+1} \quad , \\ \vdots & \vdots & \vdots \quad , \\ s_{n-1} & = & t_1 t_{2n-1} \quad , \end{cases}$$

where the t_i are the simple transpositions generating $\mathfrak{S}_{2n}$. The image of B_n consists into palindromic permutations $v := [v_1, \ldots, v_{2n}]$ that we call `Palin`, *i.e.* such that

$$v_1 + v_{2n} = v_2 + v_{2n-1} = \cdots = v_n + v_{n+1} = 2n + 1 \quad .$$

We provide the different decodings `BnCode2Palin` and `BnPalin2Code`. Of course, we can use the functions in the `SG` package to handle palindromic permutations.

Another description of B_n is as the set of signed permutations of $\{1, \ldots, n\}$, that we call `Bar`. Let $u := [u_1, \ldots, u_n]$, then u is a signed permutation when $[|u_1|, \ldots, |u_n|]$ is a permutation of $\{1, \ldots, n\}$.

The right action of $s_i, i > 0$ on a signed permutation $u := [u_1, \ldots, u_n]$ consists into transposing u_i and u_{i+1}:

$$[u_1, \ldots, u_i, u_{i+1}, \ldots, u_n]\, s_i \;=\; [u_1, \ldots, u_{i+1}, u_i, \ldots, u_n] \;, \quad i > 0 \quad .$$

The generator s_0 transforms u_1 into $-u_1$:

$$[u_1, \ldots, u_n]\, s_0 \;=\; [-u_1, \ldots, u_n] \quad .$$

The left action of $s_i, i > 0$ is to exchange values i and $i+1$ leaving signs as they were in the initial permutation. The generator s_0 acts on u by changing the value 1 into -1, or the value -1 to 1.

We provide the different conversions `BnCode2Bar` and `BnBar2Code` as well as the direct multiplication between signed permutations (`BnMultBar`).

`Bar` permutations can also be represented by permutation matrices where some entries 1 can be replaced by -1. For that purpose, we give the two functions `BnBar2Mat` and `BnMat2Bar`.

2.12 Algebras related to hyperoctahedral groups

As in the case of the symmetric group, we provide different algebras associated to B_n, as well as their actions on the ring of polynomials:

Function name prefix	Algebra
`BnGa`	is the group algebra of B_n;
`BnNca`	is the nilCoxeter algebra of B_n;
`BnIdca`	is the idCoxeter algebra of B_n.

Elements of these algebras are sums of elements of B_n (given as codes) with any type of coefficients. Addition (`BnaAdd`, `BnaMinus`) is provided once for all these algebras.

For all of these algebras, we have a different multiplication:

Function	Object
`BnGaMult`	uses the multiplication in B_n;
`BnNcaMult`	uses the multiplication where the relations $s_i^2 = 1, i \geq 0$ have been changed into $\partial_i^B \partial_i^B = 0$;
`BnIdcaMult`	uses the multiplication where the relations $s_i^2 = 1, i \geq 0$ have been changed into $\pi_i^B \pi_i^B = -\pi_i^B, i > 0$ and $\pi_0^B \pi_0^B = \pi_0^B$.

The group algebra (`BnGa`) acts (`BnGaOnPol`) on the ring of polynomials in $x_1, \ldots, x_n$ (by default). The $\{s_i, i = 1, \ldots, n-1\}$ are the usual generators of $\mathfrak{S}_n$, *i.e.* s_i transposes (x_i, x_{i+1}), and s_0 changes x_1 into $-x_1$.

The nilCoxeter algebra (`BnNca`) acts (`BnNcaOnPol`) on the ring of polynomials in $x_1, \ldots, x_n$ (by default). The $\{\partial_i^B, i = 1, \ldots, n-1\}$ are the usual divided differences, *i.e.* ∂_i^B acts by:

$$p \quad \to \quad \frac{p - s_i(p)}{x_i - x_{i+1}} \quad .$$

The extra divided difference ∂_0^B is defined by:

$$p \quad \to \quad \frac{p - s_0(p)}{2x_i} \quad .$$

The idCoxeter algebra (`BnIdca`) acts (`BnIdcaOnPol`) on the ring of (Laurent) polynomials in $x_1, \ldots, x_n$ (by default). The $\{\pi_i^B, i = 1, \ldots, n-1\}$ are the usual generators:

$$p \quad \to \quad \frac{p - s_i(p)}{x_i - x_{i+1}} x_{i+1} \quad .$$

The extra generator π_0^B is defined by:

$$p \quad \to \quad \frac{x_1 p - \frac{1}{x_1} p'}{1/x_1 - x_1} \quad ,$$

where p' is obtained from p by changing x_1 into $1/x_1$. For example,

$$\pi_0^B(x_1^k) = -(x_1^k + x_1^{k-2} + \cdots + x_1^{-k+2} + x_1^{-k}) \quad .$$

```
ACE> BnIdcaOnPol([1], x1^6);

                                12       10       8        6        4        2
                              x1    + x1    + x1   + x1   + x1   + x1   + 1
                           - --------------------------------------------------
                                                     6
                                                   x1

ACE> expand(");

                               6       4       2           1       1       1
                           - x1   - x1   - x1   - 1 - --- - --- - ---
                                                           2       4       6
                                                         x1      x1      x1
```

The elements of the algebras can also be decoded in terms of either signed permutations (`BnaCode2Bar`) or palindromic permutations (`BnaCode2Palin`).

In both group algebra and nilCoxeter algebra of B_n, one defines a Yang-Baxter basis (`BnGaYang`, `BnNcaYang`), indexed by permutations, depending upon parameters $x_1, \ldots, x_n$ (by default) which can be specialized to $1, \ldots, n$ by the option `'num'`:

```
ACE> BnGaYang([1,3]);

                                                  3         2
      - 2 (x2 + x1) x2 [0, 2] + (2 x1  - 2 x2  x1 - 2 x1 - 2 x2) [1, 0]

                                                                 2      2
         - 4 (x1 - x2) (x2 + x1) x1 x2 [1, 3] + (- x1  + x2  + 1 + 4 x2 x1) [0, 0]

                   2                                   2              2
         + (- 2 x2  - 2 x2 x1) [1, 1] + (- 4 x2 x1  + 2 x2 + 4 x2  x1) [0, 1]

         + 2 (x1 - x2) (x2 + x1) x2 [0, 3] + 4 (x2 + x1) x1 x2 [1, 2]

ACE> BnGaYang([1,3], 'num');

       - 12 [0, 2] - 12 [1, 0] + 12 [0, 0] - 12 [1, 1] + 12 [0, 1] + 24 [1, 2]

       - 12 [0, 3] + 24 [1, 3]

ACE> BnNcaYang([1,3]);

         - 2 (x2 + x1) x2 [0, 2] + (- 2 x1 - 2 x2) [1, 0]

                                                                         2
         - 4 (x1 - x2) (x2 + x1) x1 x2 [1, 3] + [0, 0] + (- 2 x2  - 2 x2 x1) [1, 1]

         + 2 x2 [0, 1] + 2 (x1 - x2) (x2 + x1) x2 [0, 3] + 4 (x2 + x1) x1 x2 [1, 2]
```

2.13 Tableaux

They are two dimensional arrays obtained by filling the *Ferrers diagram* of a partition with numbers, in such a way that numbers are weakly increasing in rows from left to right, and strictly increasing from bottom to top. A tableau is *standard* if the numbers used are $1, 2, 3, \ldots$ without repetition. Tableaux also code linear bases of irreducible representations of the symmetric and linear groups.

We represent a tableau by the sequence of its rows:

3	5	
2	4	
1	1	3

will be coded in the package by `Tab:=[ [3,5] ,[2,4], [1,1,3] ]`. We recover from it the planar tableau using the `Tab2Mat` function:

```
ACE> Tab:=[ [3,5] ,[2,4], [1,1,3] ];

                    Tab := [[3, 5], [2, 4], [1, 1, 3]]

ACE> Tab2Mat(Tab);

                              [ 3  5    ]
                              [         ]
                              [ 2  4    ]
                              [         ]
                              [ 1  1  3 ]
```

A tableau can be considered as the *word* (`Tab2Word`) obtained by concatenating its rows. From this word, one reconstructs without ambiguity the tableau by cutting the word at each decrease.

More generally, Schensted has described an algorithm (a “bumping process”) to transform any word into a tableau: if `Tab` is the tableau obtained by this algorithm from the word w, then the tableau corresponding to the word wx is obtained by `Bump(Tab, x)`:

```
ACE> Bump(Tab, [7,8,6,7]);

                 [[3, 5], [2, 4, 7, 8], [1, 1, 3, 6, 7]]

ACE> Tab2Mat(");

                           [ 3  5          ]
                           [               ]
                           [ 2  4  7  8    ]
                           [               ]
                           [ 1  1  3  6  7 ]
```

Two words which give the same tableau are said to be *plactically* equivalent. As shown by Knuth, this equivalence on words is induced from the following equivalence on words of length 3 ($i < j < k$):

$$kij \equiv ikj, \quad jki \equiv jik, \quad jii \equiv iji, \quad jji \equiv jij \quad .$$

By pushing, thanks to the "jeu de taquin", a planar tableau into a north-east corner, one gets a *contretableau* which will be coded by the sequence of its columns, from left to right, each column being read from top to bottom. The system provides the functions `CTab2Word`, `CTab2Mat`, ..., which are exactly the analogues of the functions for tableaux. There is also an algorithm to get from any word a contretableau (`Word2CTab`), the elementary step being given by `BumpC`:

```
ACE> CTab:=Word2CTab([4,2,3,4,1,2,3,2]);

                    CTab := [[2, 1], [4, 3, 2], [4, 3, 2]]

ACE> CTab2Mat(CTab);

                              [ 2  4  4 ]
                              [         ]
                              [ 1  3  3 ]
                              [         ]
                              [    2  2 ]

ACE> CTab2Mat( BumpC(CTab, [7,8,6,7]) );

                           [ 2  4  4  7  8 ]
                           [               ]
                           [ 1  3  3  6  7 ]
                           [               ]
                           [          2  2 ]
```

Another way of coding standard tableaux is by *Yamanouchi words*, *i.e.* words w such that for each factorisation $w = u\,v$, then $|v|_1 \geq |v|_2 \geq |v|_3 \geq \cdots$, where $|v|_i$ denote the number of times the letter i appears in v. This coding just tells in which rows in the standard tableau are the letters $n, n-1, \ldots, 2, 1$. One has the inverse function, from Yamanouchi words to standard tableaux `Yama2StdTab`.

The *shape* of a tableau is the list of the lengths of its successive rows, and thus is a partition which is given by the function `Tab2Part`. The `ListTab` function returns the list of all tableaux of a given shape and a given evaluation: ✘

```
ACE> ListTab([3,2], [0,1,0,0,1,2,0,0,1]);

          [[[6, 9], [2, 5, 6]], [[5, 9], [2, 6, 6]], [[6, 6], [2, 5, 9]]]

ACE> map(Tab2Mat, ");

                     [ 6  9    ]  [ 5  9    ]  [ 6  6    ]
                    [[         ], [         ], [         ]]
                     [ 2  5  6 ]  [ 2  6  6 ]  [ 2  5  9 ]
```

Given a partition, the function `Part2ListStdTab` enumerates all the standard tableaux of shape this partition:

```
ACE> Part2ListStdTab([3,1]);

                    [[[2], [1, 3, 4]], [[3], [1, 2, 4]], [[4], [1, 2, 3]]]

ACE> map(Tab2Mat, ");

                           [ 2       ]  [ 3       ]  [ 4       ]
                          [[         ], [         ], [         ]]
                           [ 1  3  4 ]  [ 1  2  4 ]  [ 1  2  3 ]
```

The equivalent enumeration of Yamanouchi words is given by `ListYama`. It computes all Yamanouchi words corresponding to a given partition part. These words are exactly the words which give the tableaux $[\ldots, [3\ldots 3], [2\ldots 2], [1\ldots 1]]$ by Schensted's algorithm:

```
ACE> ListYama([4,2,1]);

[[1, 2, 1, 4, 3, 2, 1], [2, 1, 1, 4, 3, 2, 1], [2, 1, 4, 1, 3, 2, 1], [2, 4, 1, 1, 3, 2, 1],

 [4, 2, 1, 1, 3, 2, 1], [4, 2, 1, 3, 1, 2, 1], [4, 2, 3, 1, 1, 2, 1], [4, 3, 2, 1, 1, 2, 1],

 [4, 3, 2, 1, 2, 1, 1], [4, 3, 2, 2, 1, 1, 1], [4, 2, 3, 1, 2, 1, 1], [4, 2, 3, 2, 1, 1, 1],

 [4, 2, 1, 3, 2, 1, 1], [2, 4, 1, 3, 1, 2, 1], [2, 4, 3, 1, 1, 2, 1], [2, 4, 3, 1, 2, 1, 1],

 [2, 4, 3, 2, 1, 1, 1], [2, 4, 1, 3, 2, 1, 1], [2, 1, 4, 3, 1, 2, 1], [2, 1, 4, 3, 2, 1, 1],

 [1, 2, 4, 1, 3, 2, 1], [1, 4, 2, 1, 3, 2, 1], [4, 1, 2, 1, 3, 2, 1], [4, 1, 2, 3, 1, 2, 1],

 [4, 1, 3, 2, 1, 2, 1], [4, 3, 1, 2, 1, 2, 1], [4, 3, 1, 2, 2, 1, 1], [4, 1, 3, 2, 2, 1, 1],

 [4, 1, 2, 3, 2, 1, 1], [1, 4, 2, 3, 1, 2, 1], [1, 4, 3, 2, 1, 2, 1], [1, 4, 3, 2, 2, 1, 1],

 [1, 4, 2, 3, 2, 1, 1], [1, 2, 4, 3, 1, 2, 1], [1, 2, 4, 3, 2, 1, 1]]

ACE> map(Word2Tab, "):
ACE> Tab2Mat( op( {op(")} ) );

                                   [ 4       ]
                                   [         ]
                                   [ 3       ]
                                   [         ]
                                   [ 2  2    ]
                                   [         ]
                                   [ 1  1  1 ]
```

A tableau t gives a word in non commuting letters $1, 2, \ldots$, but also a monomial x^t by replacing each letter i by the variable x_i. Now, through this transformation, the sum of all tableaux of a given shape *part* gives a polynomial $\sum x^t$ which is exactly the Schur function of index the partition conjugate to *part*. The best way to explain why the sum of all tableaux gives a symmetric function is to introduce an action of the symmetric group on words, which sends tableaux to tableaux of the same shape, and is compatible with plactic equivalence. This action is given by the `PermOnWord` function.

Now, the set of tableaux of a given shape is globally invariant under this action. Of course, the induced action on the monomials x^t is the usual action of the symmetric group on the variables $x_1, x_2, \ldots$

Finally, to each word one associates two integral vectors, its *charge vector* (that can be computed by the `Word2Charge` function) and its *cocharge vector* (`Word2CoCharge`):

```
ACE> Word2Charge([3,2,2,1,4,1,1,3], 'vector');

                              [0, 0, 0, 0, 1, 0, 0, 1]

ACE> Word2CoCharge([3,2,2,1,4,1,1,3], 'vector');

                              [2, 1, 1, 0, 2, 0, 0, 1]
```

These two vectors are compatible with Schensted (`Schensted`) algorithm and the plactic equivalence, as well as with cyclic permutations on words. The sum of the entries in the cocharge vector is a number which is the height of the tableau, when the set of all tableaux is made into a rank poset by the operation of deleting the first letter of a tableau and inserting from the right:

$$t = [[xu][v] \cdots [w]] \to t' = \texttt{Word2Tab}([u, v, \ldots, w, x])$$

```
ACE> lt:=ListTab([3,2,1], [1,2,2,1]);

         lt := [[[4], [3, 3], [1, 2, 2]], [[4], [2, 3], [1, 2, 3]],

              [[3], [2, 4], [1, 2, 3]], [[3], [2, 3], [1, 2, 4]]]

ACE> map(Tab2Mat, lt);

                   [ 4       ]  [ 4       ]  [ 3       ]  [ 3       ]
                   [         ]  [         ]  [         ]  [         ]
                  [[ 3   3   ], [ 2   3   ], [ 2   4   ], [ 2   3   ]]
                   [         ]  [         ]  [         ]  [         ]
                   [ 1   2   2 ]  [ 1   2   3 ]  [ 1   2   3 ]  [ 1   2   4 ]

ACE> map(Tab2Word, lt);

    [[4, 3, 3, 1, 2, 2], [4, 2, 3, 1, 2, 3], [3, 2, 4, 1, 2, 3], [3, 2, 3, 1, 2, 4]]

ACE> map(Word2Charge, ");

                                 [3, 2, 2, 1]
```

On the other hand, the sum $\sum q^{chrg(t)} x^t$ on all tableaux of a given shape *part*, where $chrg(t)$ is the sum of the entries of the charge vector, is the *Hall-Littlewood* polynomial Q'_{part}.

2.14 Schubert polynomials

Let $\mathbf{x} := \{x_1, \ldots, x_n\}$ be a totally ordered set of variables. We denote by $s_i, i = 1, \ldots, n-1$ the elementary transposition seen as the operator on $\mathbb{Z}[\mathbf{x}]$ that interchanges x_i and x_{i+1} and fixes all other variables. Elementary transpositions satisfy the following relations:

$$s_i s_j = s_j s_i \qquad \text{if } |i-j| > 1 \quad , \tag{2.62}$$

$$s_i s_j s_i = s_j s_i s_j \qquad \text{if } j = i+1 \quad , \tag{2.63}$$

known as the famous *braid* relations. We also consider another operator $\partial_i, i = 1, \ldots, n-1$ on the polynomial ring defined by:

$$\mathbb{Z}[x_1, \ldots, x_n] \ni \quad p \longrightarrow \partial_i(p) := \frac{p - s_i\, p}{x_i - x_{i+1}} \quad , \tag{2.64}$$

where $s_i\, p$ stands for the polynomial p in which variables x_i and x_{i+1} have been exchanged. This linear operator was introduced by Newton and is called Newton's *divided difference*. In 1973, Bernstein, Gelfand & Gelfand and Demazure observed that divided differences fulfil, like elementary transpositions, the braid relations:

$$\partial_i \partial_j = \partial_j \partial_i \qquad \text{if } |i-j| > 1 \quad , \tag{2.65}$$

$$\partial_i \partial_j \partial_i = \partial_j \partial_i \partial_j \qquad \text{if } j = i+1 \quad . \tag{2.66}$$

It is well-known that the symmetric group $\mathfrak{S}_n$ of degree n is generated by elementary transpositions $s_i, i = 1, \ldots, n-1$, so that for any permutation $\mu \in \mathfrak{S}_n$, one can find an expression of μ as a product $s_{i_1} \ldots s_{i_k}$ of elementary transpositions. Such an expression is far from being unique—because elementary transpositions satisfy relations (2.62), (2.63) together with the extra relation $s_i^2 = 1, i = 1, \ldots, n-1$—but is said to be *reduced* when k is exactly the *number of inversions* of the permutation μ. Now, relations (2.65) and (2.66) imply that for any permutation $\mu \in \mathfrak{S}_n$, there exists a divided difference ∂_μ (Newton's case corresponding to elementary transpositions). Indeed, we may define unambiguously for any reduced decomposition—because relation $s_i^2 = 1$ is replaced by $\partial_i^2 = 0$ in the case of divided differences—of $\mu = s_{i_1} \ldots s_{i_k}$:

$$\partial_\mu := \partial_{i_1} \ldots \partial_{i_k} \quad . \tag{2.67}$$

Schubert polynomials, introduced in 1982 by Lascoux and Schützenberger [72] as a combinatorial tool for some issues in algebraic geometry, are the images under divided differences of the monomial $x^\delta = x_1^{n-1} x_2^{n-2} \ldots x_n^0$:

$$\mathbf{X}_\mu := \partial_{\mu^{-1}\omega}(x^\delta) \quad , \tag{2.68}$$

where $\omega = (n, n-1, \ldots, 2, 1)$ denotes the longest element of $\mathfrak{S}_n$. These are polynomials in n variables, indexed by permutations $\mu \in \mathfrak{S}_n$ which provide a convenient basis of the polynomial ring regarded as a free module over the ring of symmetric polynomials. Schubert polynomials are non-negative integral linear combinations of the monomials of the obvious basis $\{x_1^{i_1} \cdots x_n^{i_n} : 0 \leq i_k \leq n-k,\ 1 \leq k \leq n\}$, over which they present many advantages. Their geometric interpretation is the source of many non trivial identities that are liftings at the level of polynomials of classical or recent formulas of the Schubert calculus, whose original version was only valid in a certain quotient of $\mathbb{Z}[x_1, \ldots, x_n]$.

This quotient is the so-called ring of coinvariants of the symmetric group. It is the quotient $R = \mathbb{Z}[x_1, \ldots, x_n]/I^+$ of the ring of polynomials by the ideal I^+ generated by the symmetric polynomials without constant term. Geometrically, R can be interpreted as the cohomology ring $H^*(\mathcal{F}_n, \mathbb{Z})$ of the variety of complete flags in $\mathbb{C}^n$. With this interpretation, the class of the Schubert polynomial $\mathbf{X}_\mu$ is the Poincaré dual of the Schubert cycle $[\overline{B}_\mu]$.

Schubert polynomials are compatible with the embedding $\mathfrak{S}_n \hookrightarrow \mathfrak{S}_{n+1}$ that maps $(\mu(1), \ldots, \mu(n))$ onto $(\mu(1), \ldots, \mu(n), n+1)$. When $\mathfrak{S}_n$ is identified with its image in $\mathfrak{S}_{n+1}$, Schubert polynomials form a basis of the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates. But the most important key is that Schubert polynomials are compatible with divided differences, which turn to be the fundamental operators of the Schubert calculus. This was realized independently by Bernstein, Gelfand & Gelfand and by Demazure, who introduced such operators for any Weyl group and applied them to the description of the cohomology of generalized flag manifolds G/P where G is a complex semisimple Lie group and P a parabolic subgroup.

Algebraic calculation on Schubert polynomials involve both manipulations of combinatorial objects such as permutations, codes or diagrams, and a scalar product with self-adjoint operators. This combinatorics includes the combinatorics of Schur functions—partitions being considered as *Lehmer codes* of special permutations—and is moreover very similar to it. With Schubert polynomials, one can compute directly on polynomials, not passing to the quotient by the ideal I^+. The resulting theory appears as the natural generalization of the theory of Schur functions (which correspond to Grassmann manifolds) and provides some deep insights in the combinatorics of permutations.

2.14.1 Symmetric groups and divided difference operators

Consider a polynomial p in the variables $\ldots, x_i, \ldots, x_j, \ldots$, then the divided difference of p with respect to the variables x_i and x_j is defined by:

$$\partial_{x_i,x_j}(f) := \frac{f(\ldots, x_i, \ldots, x_j, \ldots) - f(\ldots, x_j, \ldots, x_i, \ldots)}{x_i - x_j} \quad . \tag{2.69}$$

This operator which maps polynomials onto polynomials and decreases the degrees by 1, satisfies the following properties:

$$\partial_{x_i,x_j} s_{x_i,x_j} = -\partial_{x_i,x_j} \quad , \tag{2.70}$$

$$s_{x_i,x_j} \partial_{x_i,x_j} = \partial_{x_i,x_j} \quad , \tag{2.71}$$

$$\partial^2_{x_i,x_j} = 0 \quad , \tag{2.72}$$

$$\partial_{x_i,x_j} \partial_{x_j,x_k} \partial_{x_i,x_j} = \partial_{x_j,x_k} \partial_{x_i,x_j} \partial_{x_j,x_k} \quad . \tag{2.73}$$

where s_{x_i,x_j} denotes the transposition that interchanges the variables x_i and x_j.

Consider now the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates. For each $i \geq 1$, one can define the two operators s_i and ∂_i by:

$$s_i := s_{x_i,x_{i+1}} \quad ,$$

$$\partial_i := \partial_{x_i,x_{i+1}} \quad ,$$

so that, if $p \in \mathbb{Z}[x_1, x_2, \ldots]$ and $i \geq 1$, then:

$$\partial_i(p) := \frac{p - s_i\, p}{x_i - x_{i+1}} \quad , \tag{2.74}$$

The resulting is a polynomial which is symmetric in x_i and x_{i+1}. It is very helpful to remark that both operators s_i and ∂_i separately satisfy the braid relations. This allows to define operators ∂_μ for any permutation $\mu \in \mathfrak{S}_n$. When considering that $\mathfrak{S}_n$ is embedded into $\mathfrak{S}_{n+1}$ by adding a fixed point $n+1$, then we can define the group:

$$\mathfrak{S}_\infty := \lim_{i \to \infty} \mathfrak{S}_i \quad ,$$

of permutations of the set of positive integers that fix all but a finite number of them. Thus, one can more generally consider operators ∂_μ for any permutation $\mu \in \mathfrak{S}_\infty$.

Let $p \in \mathbb{K}[x_1]$ be a polynomial in the unique variable x_1 of degree less than n and let also $\{y_1, \ldots, y_{n-1}\}$ denote any set of elements of $\mathbb{K}$. Then, the Newton interpolation formula can be written

$$p(x_1) = p(y_1) + \sum_{i=1}^{n-1} \partial_i \ldots \partial_1(p)(x_1 - y_1) \ldots (x_1 - y_i) \quad , \tag{2.75}$$

in which divided differences act on the alphabet $\{y_1, \ldots, y_{n-1}\}$.

A couple (i, j) of indices such that $1 \le i < j \le n$ and $\mu(i) > \mu(j)$ is called an *inversion* of μ. The *code* of μ is the vector $c(\mu) := (c_1, \ldots, c_n)$ of non-negative integers, c_i being the number of j such that (i, j) is an inversion of μ. The code of a permutation describes its inversions and is another coding of the permutation. In other words, there is a one-to-one correspondence between elements of $\mathfrak{S}_n$ and vectors of $\mathbb{N}^n$ or more generally between permutations of $\mathfrak{S}_\infty$ and vectors of $\mathbb{N}^{(\mathbb{N}^*)}$. It is also useful to consider the partition $\lambda(\mu)$ associated with the permutation μ. The parts of this partition are the non-zero entries of the vector $c(\mu)$, arranged in weakly decreasing order.
We introduce a special class of permutations known as the vexillary class. A permutation μ is said to be *vexillary* if there do not exist four positions $1 \le i < j < k < l$ such that $\mu(j) < \mu(i) < \mu(l) < \mu(k)$. For instance, all permutations of $\mathfrak{S}_1, \mathfrak{S}_2$ and $\mathfrak{S}_3$ are vexillary, while the only permutation in $\mathfrak{S}_4$ which is not vexillary is $\mu = (2, 1, 4, 3)$. Testing whether a permutation is vexillary or not is not so trivial but among vexillary permutations one has sub-families for which this test is easier. Let $\mu \in \mathfrak{S}_n$ and $c(\mu) = (c_1, \ldots, c_n)$ denote its code. Then, μ is said to be *dominant* when $c(\mu)$ is a partition, *i.e.* a weakly decreasing sequence of non-negative integers. On another hand, μ is said to be *Grassmannian* if and only if $c_1 \le \cdots \le c_k$ and $\forall i > k, c_i = 0$ for some k. In fact, dominant and Grassmannian permutations are vexillary ones.

We recall that $s_1, \ldots, s_{n-1}$ generate the symmetric group $\mathfrak{S}_n$ *i.e.* each permutation $\mu \in \mathfrak{S}_n$ can be expressed as a product of elementary (simple) transpositions. The *length* $l(\mu)$ of the permutation μ is the minimal number of simple transpositions required to express μ as such a product: this product is then called a *reduced decomposition* of μ. It is a classical result that the length of a permutation μ is also equal to the number of its inversions or equivalently to the sum of all components of its code.

Because of relations (2.62) and (2.63) the reduced decomposition of a permutation $\mu \in \mathfrak{S}_n$ is in general not unique, but thanks to relations (2.65) and (2.66) an operator:

$$\partial_\mu := \partial_{i_1} \ldots \partial_{i_k} \quad , \tag{2.76}$$

can be unambiguously defined for any reduced decomposition $s_{i_1} \ldots s_{i_k}$ of μ. Moreover, considering any sequence of positive integers $\mathbf{i} = (i_1, \ldots, i_p)$, the property $\partial_i^2 = 0, i = 1, \ldots, n-1$ shows that if $\mathbf{i}$ is not a reduced decomposition of a certain permutation, then $\partial_{\mathbf{i}} = \partial_{i_1} \ldots \partial_{i_p} = 0$.

2.14.2 Schubert polynomials

Schubert polynomials possess many interesting combinatorial properties related to the symmetric group. They contain as a subfamily Schur functions which are the fundamental symmetric polynomials. This section gives a brief exposition of some properties of Schubert polynomials that lead to efficient computations.
For each $\mu \in \mathfrak{S}_n$, the (simple) Schubert polynomial $\mathbf{X}_\mu$ is defined by:

$$\mathbf{X}_\mu := \partial_{\mu^{-1}\omega}(x^\delta) \quad ,$$

where ω denotes the longest—with respect to the number of inversions—element of $\mathfrak{S}_n$, μ^{-1} the inverse of μ and $x^\delta := x_1^{n-1} x_2^{n-2} \ldots x_n^0$. For instance, we compute $\mathbf{X}_\mu$ for $\mu = (1, 4, 2, 3)$ as follows:

we have $\omega = (4,3,2,1)$ and $\mu^{-1} = (1,3,4,2)$ so that $\mu^{-1}\omega = (2,4,3,1) = s_1 s_2 s_3 s_2$. Thus, by definition, $\mathbf{X}_\mu = \partial_{2,4,3,1}(x^\delta) = \partial_1\partial_2\partial_3\partial_2(x_1^3 x_2^2 x_3)$. Now, remember that, thanks to relation (2.66), $\partial_2\partial_3\partial_2 = \partial_3\partial_2\partial_3$:

$$
\begin{aligned}
\mathbf{X}_{1,4,2,3} &= \partial_1\partial_3\partial_2\partial_3(x_1^3 x_2^2 x_3) \\
&= \partial_1\partial_3\partial_2(x_1^3 x_2^2) \\
&= \partial_1\partial_3(x_1^3 x_2 + x_1^3 x_3) \\
&= \partial_1(x_1^3) = x_1^2 + x_1 x_2 + x_2^2 \quad .
\end{aligned}
$$

Some basic properties of Schubert polynomials are the following:

1. $\forall \mu \in \mathfrak{S}_n$, $\mathbf{X}_\mu$ is a homogeneous polynomial of degree $l(\mu)$ in the variables $x_1, \ldots, x_{n-1}$.
2. $\forall \mu \in \mathfrak{S}_n$, $\mathbf{X}_\mu$ is symmetric in x_i, x_{i+1} if and only if $\mu(i) < \mu(i+1)$.
3. $\mathbf{X}_\omega = x^\delta$ and $\mathbf{X}_{1,\ldots,n} = 1$.
4. Whenever μ belongs to a *Young subgroup* $\mathfrak{S}_i \times \mathfrak{S}_j$ *i.e.* $\mu = \mu'\mu''$, $\mu' \in \mathfrak{S}_i \times \{id\}$, $\mu'' \in \{id\} \times \mathfrak{S}_j$, we have $\mathbf{X}_\mu = \mathbf{X}_{\mu'}\mathbf{X}_{\mu''}$.

Schubert polynomials are sometimes indexed by sequences $\mathbf{i} = (i_1, \ldots, i_n) \subset \mathbb{N}^n$ by taking the code of μ instead of the permutation itself. We denote by $\mathbf{Y}_\mathbf{i}$ the Schubert polynomial corresponding to the sequence $\mathbf{i} \subset \mathbb{N}^n$. On account of the above properties, it is quite easy to deduce that $\mathbf{Y}_\mathbf{i}$, $\mathbf{i} \subset \mathbb{N}^n$ is homogeneous of degree $|\mathbf{i}| = i_1 + \cdots + i_n$. Calculations may be simplified by the following properties:

1. Let μ be a dominant permutation, then $\mathbf{X}_\mu = x^{\lambda(\mu)}$.
2. If μ is a Grassmannian permutation, then $\mathbf{X}_\mu = s_{\lambda(\mu)}(x_1, \ldots, x_r)$ where r is the unique descent of μ and $s_{\lambda(\mu)}$ is the Schur function corresponding to the partition $\lambda(\mu)$.
3. $\mathbf{Y}_{\underbrace{0\ldots0i}_{r}} = s_i(x_1, \ldots, x_r)$, $i > 0$.
4. $\mathbf{Y}_{\underbrace{0\ldots0}_{r-k}\underbrace{1\ldots1}_{k}} = s_{\underbrace{1\ldots1}_{k}}(x_1, \ldots, x_r)$.
5. In particular, $\mathbf{X}_{s_i} = x_1 + \cdots + x_i$, for $i = 1, \ldots, n-1$.

The function `TableX` in the `SP` package computes all Schubert polynomials corresponding to a given symmetric group. The following *Maple* example illustrates such a computation for $\mathfrak{S}_3$, the table being indexed by the 6 permutations of the group $\mathfrak{S}_3$:

```
ACE> TableX(3);
                          table([
                            [1, 2, 3] = 1
                            [2, 1, 3] = x1
                            [1, 3, 2] = x1 + x2
                            [2, 3, 1] = x1 x2
                                          2
                            [3, 1, 2] = x1
                                          2
                            [3, 2, 1] = x1  x2
                          ])
```

Schubert polynomials $\mathbf{X}_\mu$, $\mu \in \mathfrak{S}_n$ form a $\mathbb{Z}-$basis of the subspace of $\mathbb{Z}[x_1, \ldots, x_n]$ spanned by the monomials $x^\alpha, \alpha \subseteq (n-1, n-2, \ldots, 0)$, consequently Schubert polynomials $\mathbf{X}_\mu$, $\mu \in \mathfrak{S}_\infty$ form a $\mathbb{Z}-$basis of the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates (the function `ToX` expresses any general expression in the basis of Schubert polynomials). The function `x2X` converts any expression from the basis of monomials to the basis of Schubert polynomials while the function `X2x` realizes the reverse operation as illustrated by the following example:

```
ACE> x2X(x1^5 + x1*x2^3*x3);

          X[2, 5, 3, 1, 4] - X[3, 4, 2, 1] - X[4, 2, 3, 1] + X[6, 1, 2, 3, 4, 5]

ACE> X2x(");
                                  5         3
                                x1   + x1 x2   x3
```

Schubert polynomials $\mathbf{X}_\mu, \mu \in \mathfrak{S}_n$ also form a basis of the algebra of polynomials $\mathbb{Z}[x_1, \ldots, x_n]$ regarded as a free module over the ring of symmetric polynomials $\mathbf{Sym}[x_1, \ldots, x_n]$. This point exceeds the scope of this paper hence we refer the interested reader to [66] for more details and solely give an example of this interpretation of $\mathbb{Z}[x_1, \ldots, x_n]$. Considering the previous polynomial $p = x_1^5 + x_1 x_2^3 x_3$ for which the expansion in terms of Schubert polynomials involves permutations in $\mathfrak{S}_4$, $\mathfrak{S}_5$ and $\mathfrak{S}_6$, we have another expression of p that only involves Schubert polynomials in $\mathfrak{S}_3$ but now with coefficients that are symmetric functions in the variables x_1, x_2, x_3 expressed for instance in the Schur basis: $p = (s_{3,1,1} - s_{2,2,1})\mathbf{X}_{1,2,3} - s_{3,1}\mathbf{X}_{2,1,3} - s_{1,1,1}\mathbf{X}_{2,3,1} + (s_3 - s_{1,1,1})\mathbf{X}_{3,1,2} + s_{2,1,1}\mathbf{X}_{1,3,2}$.

The multiplicative structure in $\mathbf{Sym}[x_1, \ldots, x_n]$ is determined by *Pieri's formula* that expresses in the basis of Schur functions the product $h_i s_\lambda$ (respectively $e_i s_\lambda$), $i \geq 0$, λ being a partition, and where h_i (resp. e_i) denotes the i-th complete (resp. elementary) symmetric function. In the Schubert basis, the multiplicative structure relies on the multiplication by one variable. It involves a certain combinatorics on permutations described by *Monk's formula* [76], [75, formula (4.15')]:

$$x_r \mathbf{X}_\mu = \sum_{\nu = \mu\, t_{ri},\, i > r,\, l(\nu) = l(\mu)+1} \mathbf{X}_\nu - \sum_{\eta = \mu\, t_{ir},\, i < r,\, l(\eta) = l(\mu)+1} \mathbf{X}_\eta \quad , \tag{2.77}$$

where t_{ij} denotes the transposition that interchanges i and j. Because $\mathbf{X}_{s_r} = x_1 + \cdots + x_r$, it can be equivalently stated by:

$$\mathbf{X}_{s_r} \mathbf{X}_\mu = \sum_{\nu = \mu\, t_{ij},\, i \leq r < j,\, l(\nu) = l(\mu)+1} \mathbf{X}_\nu \quad .$$

The function Monk realizes this operation. For instance, to compute $x_4\mathbf{X}_{3,6,2,4,8,7,1,5}$:

```
ACE> Monk(4, X[3,6,2,4,8,7,1,5]);

     - X[3, 6, 4, 2, 8, 7, 1, 5] - X[4, 6, 2, 3, 8, 7, 1, 5] + X[3, 6, 2, 8, 4, 7, 1, 5]

     + X[3, 6, 2, 7, 8, 4, 1, 5] + X[3, 6, 2, 5, 8, 7, 1, 4]
```

A more general Pieri formula [73] provides a way to express the product $h_i\mathbf{X}_\mu$ (resp. $e_i\mathbf{X}_\mu$) in the Schubert basis. For the time being, Monk's formula appears as the fundamental tool for calculation on Schubert polynomials. *Exempli gratia*, we may choose r so that there is only one positive term in the product $x_r\mathbf{X}_\mu$. In that case, we obtain the following so-called *transition*:

$$\mathbf{X}_\nu = x_r\mathbf{X}_\mu + \sum_\eta \mathbf{X}_\eta \quad . \tag{2.78}$$

By iterating this same principle on $\mathbf{X}_\mu$ and all $\mathbf{X}_\eta$, it gives an efficient way to develop a Schubert polynomial $\mathbf{X}_\nu$ in the basis of monomials (this algorithm is used by the function X2x). Now, starting from an identity of type (2.78), one writes:

$$\mathbf{X}_\nu \longrightarrow \sum_\eta \mathbf{X}_\eta \quad , \tag{2.79}$$

and calls it a *truncated transition*. This amounts to ignore the extra term $x_r\mathbf{X}_\mu$ (this can be of course realized by putting $x_r = 0$). To define the *maximal transition* of ν, we start from equation (2.78) in which one chooses r to be the last descent of the permutation—this ensures the validity of the equation (2.79) except for the identity permutation. The maximal transition corresponding to a given permutation may be computed by the function MaxTr as illustrated in the sequel:

```
ACE> MaxTr([3,1,5,2,4]);

       X[3, 1, 5, 2, 4] = x3 X[3, 1, 4, 2, 5] + X[3, 4, 1, 2, 5] + X[4, 1, 3, 2, 5]
```

To any permutation ν, one can associate a tree defined as follows: if ν is vexillary, the tree is reduced to one leaf indexed by ν; otherwise, considering the truncated maximal transition $\mathbf{X}_\nu \longrightarrow \sum \mathbf{X}_\eta$, one builds all edges from μ to all sub-trees corresponding to each η.

The action $\longrightarrow$ that corresponds in fact to an alphabet restriction, allows to compute the symmetric part of a Schubert polynomial as a sum of Schur functions (which are Schubert polynomials corresponding to Grassmannian permutations). A special case of the algorithm [70] gives the decomposition of a product of two Schur functions. This algorithm which is totally different from the famous *Littlewood-Richardson rule*, is implemented in SP. The following example gives the expansion of the product $s_{3,1}s_{2,1}$ in terms of Schur functions:

```
ACE> LRrbyX(s[3,1], s[2,1]);

           s[5, 2] + s[5, 1, 1] + s[4, 3] + 2 s[4, 2, 1] + s[4, 1, 1, 1]

         + s[3, 3, 1] + s[3, 2, 2] + s[3, 2, 1, 1]
```

In relation with the geometry of flag manifolds, it is natural to define a scalar product on $\mathbb{Z}[x_1, \ldots, x_n]$ for which elements of the ring of symmetric polynomials $\mathbf{Sym}[x_1, \ldots, x_n]$ are scalars. The use of the ∂_ω, $(\omega \in \mathfrak{S}_n)$ operator seems to be the best adapted so that the scalar product may be defined by:

$$\langle f, g \rangle := \partial_\omega(fg) \quad . \tag{2.80}$$

For this scalar product, divided differences ∂_i, $i = 1, \ldots, n-1$ are self-adjoint and more generally we get the next property on Schubert polynomials: let μ, $\nu \in \mathfrak{S}_n$ such that $l(\mu) + l(\nu) = \binom{n}{2}$, then:

$$\langle \mathbf{X}_\mu, \mathbf{X}_\nu \rangle = \begin{cases} 1 & \text{if } \nu = \omega\mu \quad , \\ \\ 0 & \text{otherwise} \quad . \end{cases} \tag{2.81}$$

Thus, Schubert polynomials of degree k are adjoint to Schubert polynomials of degree $\binom{n}{2} - k$. The following *Maple* instructions show how to compute all scalar products between Schubert polynomials in $\mathfrak{S}_3$ using the function `ScalarPol` that efficiently implements definition (2.80), expressing the result in terms of Schur functions:

```
ACE> Sn:=3:
ACE> scal:=array(1..Sn!, 1..Sn!): l:=ListPerm(Sn):
ACE> for i from 1 to Sn! do
      for j from i to Sn! do
        scal[i, j] := ScalarPol(X2x(X[op(op(i, l))]), X2x(X[op(op(j, l))]), Sn);
        scal[j, i] := scal[i, j]
      od
    od:
ACE> print(l);

          [[1, 2, 3], [1, 3, 2], [3, 1, 2], [2, 1, 3], [2, 3, 1], [3, 2, 1]]
```

```
ACE> print(scal);
                       [ 0    0     0     0     0        1    ]
                       [                                      ]
                       [ 0    0     1     0     0      s[1]   ]
                       [                                      ]
                       [ 0    1     0     0    s[1]    s[2]   ]
                       [                                      ]
                       [ 0    0     0     0     1      s[1]   ]
                       [                                      ]
                       [ 0    0    s[1]   1     0     s[1, 1] ]
                       [                                      ]
                       [ 1  s[1]  s[2]  s[1] s[1, 1]  s[2, 1] ]
```

The next section is devoted to the study of double Schubert polynomials and our point of view will be to shed some new light on simple Schubert polynomials.

2.14.3 Double Schubert polynomials

Let us take two independent sets of indeterminates $\mathbf{x} := \{x_1, \ldots, x_n\}$ and $\mathbf{y} := \{y_1, \ldots, y_n\}$. The *maximal double Schubert polynomial* $\mathbb{X}_\omega$ is defined by:

$$\mathbb{X}_\omega(\mathbf{x}, \mathbf{y}) := \prod_{i+j \leq n} (x_i - y_j) \quad , \tag{2.82}$$

where ω denotes the longest element of $\mathfrak{S}_n$, and in general for any permutation $\mu \in \mathfrak{S}_n$ the *double Schubert polynomial* $\mathbb{X}_\mu$ is defined to be the image under divided differences of $\mathbb{X}_\omega(\mathbf{x}, \mathbf{y})$:

$$\mathbb{X}_\mu(\mathbf{x}, \mathbf{y}) := \partial_{\mu^{-1}\omega}(\mathbb{X}_\omega(\mathbf{x}, \mathbf{y})) \quad , \tag{2.83}$$

in which divided differences act only on the $\mathbf{x}$ variables. These polynomials which belong to the polynomial ring $\mathbb{Z}[x_1, \ldots, x_{n-1}, y_1, \ldots, y_{n-1}]$ have the following properties:

1. $\forall \mu \in \mathfrak{S}_n$, $\mathbb{X}_\mu$ is homogeneous of degree $l(\mu)$ in the variables $x_1, \ldots, x_{n-1}, y_1, \ldots, y_{n-1}$.
2. Specializing all y_i's to 0, $\mathbb{X}_\mu(\mathbf{x}, \mathbf{y}) = \mathbf{X}_\mu(\mathbf{x})$ because $\mathbb{X}_\omega(\mathbf{x}, 0) = x^\delta$.
3. $\mathbb{X}_{1,\ldots,n}(\mathbf{x}, \mathbf{y}) = 1$.
4. $\forall \mu \in \mathfrak{S}_n$, $\mathbb{X}_{\mu^{-1}}(\mathbf{x}, \mathbf{y}) = \mathbb{X}_\mu(-\mathbf{y}, -\mathbf{x})$.

The `SP` package provides the function `TableXX` to compute the table of all double Schubert polynomials corresponding to $\mathfrak{S}_n$. The example shown below illustrates the use of this function for $n = 3$:

```
ACE> TableXX(3);
                    table([
                        [1, 2, 3] = 1
                        [2, 1, 3] = x1 - y1
                        [1, 3, 2] = x1 - y1 + x2  - y2
                        [2, 3, 1] = (x1 - y1) (x2 - y1)
                        [3, 1, 2] = (x1 - y1) (x1 - y2)
                        [3, 2, 1] = (x1 - y1) (x1 - y2) (x2 - y1)
                    ])
```

Double Schubert polynomials generalize the coefficients $(x_1 - y_1)\ldots(x_1 - y_i)$ in the Newton interpolation formula for functions of one variable (2.75) to the case of any number of variables. In other words, considering one set of indeterminates $\mathbf{x} := \{x_1, x_2, \ldots\}$ and $\mathbf{y} := \{y_1, \ldots, y_{n-1}\}$ denoting any set of elements of a field $\mathbb{K}$, the Newton interpolation formula for $p \in \mathbb{K}[x_1, x_2, \ldots]$ becomes:

$$p(\mathbf{x}) \equiv \sum_{\mu} \partial_\mu(p)(\mathbf{y})\mathbb{X}_\mu(\mathbf{x}, \mathbf{y}) \quad , \tag{2.84}$$

in which divided differences act on the $\mathbf{y}$ variables. This formula becomes an exact formula if one takes a sufficiently big symmetric group: it then coincides with the result of the `x2XX` function. The following *Maple* commands explicit the function `NewtonInterp` that realizes the Newton interpolation, the last argument being the degree of the symmetric group:

```
ACE> NewtonInterp(x1^2, 2);

                                  2
                                y1  XX[1, 2] + (y1 + y2) XX[2, 1]
ACE> XX2x(");
                                       y1 x1 + y2 x1 - y2 y1

ACE> Tox( NewtonInterp(x1^5 + x1*x2^3*x3, 6) );

                                            5        3
                                          x1  + x1 x2  x3
```

in which `XX2x` converts any expression in terms of double Schubert polynomials to the basis of monomials and `Tox` develops any expression in the basis of monomials.

Because in definition (2.83) divided differences act on the $\mathbf{x}$ variables, double Schubert polynomials are sums of monomials x^α, $\alpha \subseteq \delta$. Thus, double Schubert polynomials can be expressed as linear combinations of simple Schubert polynomials in the variables $\mathbf{x}$ with coefficients in $\mathbb{Z}[\mathbf{y}]$. Double Schubert polynomials may be considered as generating functions for families of simple Schubert polynomials as illustrated by the relation:

$$\mathbb{X}_\mu(\mathbf{x}, \mathbf{y}) = \sum_{\nu} \mathbf{X}_\nu(\mathbf{x})\, \mathbf{X}_{\nu\omega}(-\mathbf{y}) \quad , \tag{2.85}$$

in which μ and ν lie in $\mathfrak{S}_n$. As in the case of simple Schubert polynomials (2.77), we have a Monk formula to multiply a double Schubert polynomial by a variable x_r:

$$x_r\mathbb{X}_\mu \quad = \quad y_{\mu(r)}\mathbb{X}_\mu + \sum_{\nu=\mu\, t_{rj},\, j>r,\, l(\nu)=l(\mu)+1} \mathbb{X}_\nu - \sum_{\eta=\mu\, t_{jr},\, j<r,\, l(\eta)=l(\mu)+1} \mathbb{X}_\eta \quad . \tag{2.86}$$

Thus, working with two sets of variables involves only adding an extra term [66] in Monk's formula (2.77), as shown below:

```
ACE> Monk(3, XX[2,3,5,1,4]);

            - XX[2, 5, 3, 1, 4] + XX[2, 3, 6, 1, 4, 5] + y5 XX[2, 3, 5, 1, 4]
```

Furthermore, as in the case of simple Schubert polynomials, r may be chosen such that:

$$\mathbb{X}_\nu = (x_r - y_{\mu(r)})\mathbb{X}_\mu + \sum_\eta \mathbb{X}_\eta \quad . \tag{2.87}$$

This provides a convenient way to compute the development of a double Schubert polynomial in the basis of monomials but the most interesting point to notice is that it gives an efficient manner to specialize double Schubert polynomials. Indeed, in relation (2.87) the term $(x_r - y_{\mu(r)})\mathbb{X}_\mu$ will be very soon specialized to zero whenever the specialization of the second alphabet to a permutation of the first one maps the factor $x_r - y_{\mu(r)}$ onto zero. The `TableXX` function also gives the specialization of all double Schubert polynomials by adding a second argument which is a permutation:

```
ACE> TableXX(4, [2,4,1,3]);

           table([
             [2, 4, 3, 1] = 0
             [2, 1, 3, 4] = x1 - x2
             [2, 3, 4, 1] = 0
             [4, 2, 1, 3] = 0
             [2, 3, 1, 4] = 0
             [3, 4, 1, 2] = 0
             [3, 1, 4, 2] = (x3 - x4) (- x4 + x1) (x1 - x2)
             [1, 2, 3, 4] = 1
             [3, 4, 2, 1] = 0
             [4, 2, 3, 1] = 0
             [3, 2, 4, 1] = 0
             [3, 2, 1, 4] = 0
             [2, 4, 1, 3] = 0
             [4, 1, 2, 3] = 0
             [1, 3, 2, 4] = - x4 + x1
             [1, 4, 2, 3] = (x2 - x1) (- x4 + x1) + (- x4 + x1) (x1 - x2)
             [4, 3, 2, 1] = 0
             [1, 2, 4, 3] = x3 - x4
             [2, 1, 4, 3] = (x3 - x1) (x1 - x2) + (- x4 + x1) (x1 - x2)
             [4, 1, 3, 2] = 0
             [3, 1, 2, 4] = (- x4 + x1) (x1 - x2)
             [1, 4, 3, 2] = (x3 - x4) ((x2 - x1) (- x4 + x1) + (- x4 + x1) (x1 - x2))
             [4, 3, 1, 2] = 0
             [1, 3, 4, 2] = (x3 - x4) (- x4 + x1)
           ])
```

More generally, the specialization of the second alphabet (`Specialize`) is related to many interesting properties, which justify using double Schubert polynomials instead of their specialization $y_i = 0$.

2.14.4 Examples from algebraic geometry

Chern classes of flag manifolds

According to Borel, the cohomology ring of the flag manifold $\mathcal{F}_n$ is isomorphic to $\mathbb{Z}[x_1, \dots, x_n]/I^+$ where I^+ is the ideal generated by the symmetric polynomials in the variables $x_1, \dots, x_n$ without constant term.

The flag manifold is equipped with tautological line bundles $\mathrm{L}_1, \dots, \mathrm{L}_n$, whose respective first *Chern classes* are equal to the classes $\gamma_1, \dots, \gamma_n$ of $x_1, \dots, x_n$ modulo the ideal I^+. It follows from the splitting principle (see [64], §13) that the tangent bundle has the same Chern classes as the direct sum $\bigoplus_{i>j} \mathrm{L}_i \mathrm{L}_j^*$, that is:

$$c(\mathcal{F}_n) = \prod_{i<j} (1 + \gamma_i - \gamma_j) \quad .$$

The following sequence of instructions gives the expansion of $c(\mathcal{F}_4)$ in the basis of Schubert cycles:

```
ACE> chern:=proc(n) local i, j, res;
       res:=1;
       for j to n do
         for i to j-1 do
           res:=res * (1+x.i-x.j)
         od
       od;
       RETURN(res)
     end:
ACE> Flag(4):
ACE> chern(4);

                    (1 + x1 - x2) (1 + x1 - x3) (1 + x2 - x3)

                    (1 + x1 - x4) (1 + x2 - x4) (1 + x3 - x4)

ACE> ToX(");

      6 X[2, 3, 1, 4] +  6 X[3, 1, 2, 4]  + 4 X[2, 1, 4, 3] +  6 X[1, 4, 2, 3]

      + 6 X[3, 2, 1, 4] + 16 X[3, 1, 4, 2] + 2 X[1, 3, 2, 4] + 2 X[1, 2, 4, 3]

      + X[1, 2, 3, 4] + 6 X[1, 3, 4, 2]  +  2 X[2, 1, 3, 4] + 22 X[4, 1, 2, 3]

      + 24 X[4, 3, 2, 1] + 22 X[2, 3, 4, 1] + 20 X[2, 4, 1, 3] + 28 X[3, 2, 4, 1]

      + 44 X[3, 4, 1, 2] + 24 X[4, 2, 1, 3] + 24 X[2, 4, 3, 1] + 28 X[4, 1, 3, 2]

      +  6 X[1, 4, 3, 2] + 48 X[4, 2, 3, 1] + 36 X[4, 3, 1, 2] + 36 X[3, 4, 2, 1]
```

Moreover, Schubert polynomials are compatible with the embeddings $H^*(\mathcal{F}_{i_1 \cdots i_r}) \hookrightarrow H^*(\mathcal{F}_n)$ of the cohomology rings of incomplete flag varieties given by the fibrations:

$$(V_0 \subset V_1 \subset \cdots \subset V_n) \longrightarrow (V_{i_1} \subset V_{i_1+i_2} \subset \cdots \subset V_{i_1+\cdots+i_r}) \quad .$$

For example, the Chern classes of the Grassmannian $\mathrm{G}(2,4)$ are given by:

```
ACE> Flag(4):
ACE> res:=1:
ACE> for i from 1 to 2 do for j from 3 to 4 do res:=res * (1+x.i-x.j) od od:
ACE> ToX(res);

                7 X[2, 3, 1, 4] + 7 X[1, 4, 2, 3]  + 4 X[1, 3, 2, 4]

                + X[1, 2, 3, 4] + 12 X[2, 4, 1, 3] + 6 X[3, 4, 1, 2]
```

To obtain the result in the usual basis of Schur functions (indexation of Schubert cycles of $\mathrm{G}(k,n)$ by partitions contained in the rectangle $k \times (n-k)$), we can use the function `X2Y`:

```
ACE> X2Y(");

                7 Y[1, 1, 0, 0] + 7 Y[0, 2, 0, 0]  + 4 Y[0, 1, 0, 0]

                + Y[0, 0, 0, 0] + 12 Y[1, 2, 0, 0] + 6 Y[2, 2, 0, 0]
```

When the code of a permutation has the form $I = J.0^{n-1}$, where J is a partition (weakly increasing vector), the Schubert polynomial $\mathbf{Y}_I$ corresponds to the Schubert cycle s_J of $\mathrm{G}(k, n-k)$. The properties of the Chern classes of flag manifolds are discussed in [72], [69].

Projective degrees of Schubert varieties

Let $i : \mathcal{F}_n \hookrightarrow \mathbb{P}^N$ be a projective embedding of $\mathcal{F}_n$, corresponding to the line bundle:

$$\eta = \mathrm{L}_1^{m_1} \mathrm{L}_2^{m_2} \dots \mathrm{L}_n^{m_n} = i^* \mathcal{O}_{\mathbb{P}^N}(1) \quad ,$$

and let $h \in \mathrm{H}^1(\mathbb{P}^N)$ be the class of a hyperplane. The projective degree of a subvariety Z of codimension k in $\mathcal{F}_n$ is:

$$d(\mathrm{Z}) = \varepsilon\left(i^*([i\mathrm{Z}].h^k)\right) = \varepsilon([\mathrm{Z}].Y^k) \quad ,$$

where $\varepsilon : \mathrm{H}^{top}(\mathcal{F}_n) \longrightarrow \mathbb{Z}$ is the morphism sending to 1 the class of a point, and $Y = i^*h = \sum m_i x_i$.

For example, with the Plücker embedding we get $Y = (n-1)x_1 + \cdots + x_{n-1}$ and for this embedding, the degree of the Schubert variety $\underline{\mathbf{X}}_{2,1,4,3}$ of $\mathcal{F}_4$ is computed by the following sequence of commands:

```
ACE> y:=3*x1 + 2*x2 + x3:
ACE> ToX(y^4 * X[2,1,4,3]);

                                  78 X[4, 3, 2, 1]
```

so that the required degree is 78. For more details about how to effectively compute the projective degrees of Schubert cycles, the reader may refer to [71].

The double Schubert polynomials also have a geometrical interpretation [63]. When the variables x_i and y_j are interpreted as Chern classes of certain line bundles over a variety M, they give the cohomology classes corresponding to various degeneracy loci of maps of flagged bundles over M.

2.15 Computations in the ring of polynomials in several variables

✘

Computations in the ring $\mathbb{Z}[x_1, \ldots, x_n]$ of polynomials in several variables are not so easy because dimensions grow very quickly with the number of variables and with the degree. The problem is to find linear bases adapted to some specific problems without losing track of the multiplicative structure. For $n = 1$, such bases are provided by interpolation theory. In higher dimensions, a polynomial can have some symmetry, which cannot be efficiently exploited if the polynomial is just written as a sum of monomials.

For example, if a polynomial is totally symmetrical, we can have recourse to many combinatorial objects which render computations independent of the number of variables. To be able to use symmetry methods in the case of general polynomials, we shall look at the ring of polynomials in n variables $x_1, \ldots, x_n$ as a module over the ring $\mathbf{Sym}[x_1, \ldots, x_n]$ of symmetric polynomials. It is easy to show that the set of monomials of degree less than $(n-1, \ldots, 2, 1, 0)$ (componentwise) is a linear basis of this module, which is free of rank $n!$.

We shall use a more interesting basis for this free module over $\mathbf{Sym}[x_1, \ldots, x_n]$, consisting of Schubert polynomials in the x_i and y_i (where the y_i are a second set of indeterminates). These Schubert polynomials are the universal coefficients in the generalized Newton interpolation formula (they generalize Newton's interpolation polynomials $(x_1 - y_1) \ldots (x_1 - y_n)$). Moreover they are the natural basis in the geometric interpretation of the ring of polynomials as the equivariant cohomology ring of the flag manifold. Last, but not least, their combinatorics extends the one of Schur functions (the natural basis of the ring $\mathbf{Sym}[x_1, \ldots, x_n]$), with constructions on permutations replacing the handling of partitions and Young tableaux.

2.15.1 Schubert bases

Let w be a permutation in $\mathfrak{S}_n$ and $c(w) = (c_1, c_2, \ldots, c_n)$ its code, then:

$$\mathbf{X}_w = x_1^{c_1} x_2^{c_2} \ldots x_n^{c_n} + \sum \alpha_i \; x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} \;, \tag{2.88}$$

where $\alpha_i > 0$, $i_1 \leq n-1, i_2 \leq n-2, \ldots, i_n \leq 0$, and $(i_1, i_2, \ldots, i_n)$ lexicographically smaller than $(c_1, c_2, \ldots, c_n)$.

Schubert polynomials are compatible with the embedding $\mathfrak{S}_n \hookrightarrow \mathfrak{S}_{n+1}$. Thus, $\{\mathbf{X}_w, w \in \mathfrak{S}_\infty\}$ is a $\mathbb{Z}$-basis of $\mathbb{Z}[x_1, x_2, \ldots]$. More precisely, let $\eta : \mathbb{Z}[x_1, x_2, \ldots] \to \mathbb{Z}$ be the constant term homomorphism. Then [75, (4.14)], one has:

$$\forall p \in \mathbb{Z}[x_1, x_2, \ldots], \qquad p = \sum_{w \in \mathfrak{S}_\infty} \eta(\partial_w p) \, \mathbf{X}_w \;. \tag{2.89}$$

Example 2.15.1 Let $p := x_1^5 + x_1 x_2^3 x_3$. Then p is equal to:

$$\begin{aligned} p &= \mathbf{X}_{6,1,2,3,4,5} - \mathbf{X}_{3,4,2,1} - \mathbf{X}_{4,2,3,1} + \mathbf{X}_{2,5,3,1,4} \\ &= \quad x_1^5 \quad - x_1^2 x_2^2 x_3 - x_1^3 x_2 x_3 + (x_1 x_2^3 x_3 + x_1^2 x_2^2 x_3 + x_1^3 x_2 x_3) \;. \end{aligned}$$

It should be noted that this expression of p involves simple Schubert polynomials indexed by permutations in $\mathfrak{S}_4, \mathfrak{S}_5$ and $\mathfrak{S}_6$ while the original polynomial only depends upon the three variables x_1, x_2 and x_3. It is possible to give an expression of p involving only simple Schubert polynomials in $\mathfrak{S}_3$, but the coefficients are now symmetric functions in x_1, x_2, x_3:

$$\begin{aligned} p \;=\; & \Big(x_1x_2x_3^2 + x_1x_2^2x_3 + x_1^2x_2x_3 - x_1^2x_2^2x_3 - x_1^2x_2x_3^2 - x_1x_2^2x_3^2\Big)\, \mathbf{X}_{1,2,3} - \Big(x_1^3x_2 \\ & + x_1^3x_3 \;+\; x_1^2x_2^2 + 2x_1^2x_2x_3 + x_1^2x_3^2 + x_1x_2^3 + 2x_1x_2^2x_3 + 2x_1x_2x_3^2 + x_1x_3^3 + x_2^3x_3 \\ & + x_2^2x_3^2 + x_2x_3^3\Big)\, \mathbf{X}_{2,1,3} + \Big(x_1^3 + x_1^2x_2 + x_1^2x_3 + x_1x_2^2 + x_1x_3^2 + x_2^3 + x_2^2x_3 + x_2x_3^2 \\ & + x_3^3\Big)\, \mathbf{X}_{3,1,2} - \Big(x_1x_2x_3\Big)\, \mathbf{X}_{2,3,1} + \Big(x_1^2x_2x_3 + x_1x_2^2x_3 + x_1x_2x_3^2\Big)\, \mathbf{X}_{1,3,2} \;. \end{aligned}$$

For now on, we fix the integer n and use only Schubert polynomials $\mathbf{X}_w$ or $\mathbb{X}_w, w \in \mathfrak{S}_n$. Let $\mathbf{Sym}[\mathbf{x}] := \mathbf{Sym}[x_1, \ldots, x_n]$ be the subring of symmetric polynomials in $x_1, \ldots, x_n$. In fact, one has [75, Chap. V] the following proposition:

Proposition 2.15.1 *The space $\mathbb{Z}[\mathbf{x}]$ is a free module of dimension $n!$ over the ring $\mathbf{Sym}[\mathbf{x}]$ with basis the simple Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_n$:*

$$\mathbb{Z}[x_1, \ldots, x_n] \;=\; \bigoplus_{w \in \mathfrak{S}_n} \mathbf{Sym}[x_1, \ldots, x_n]\; \mathbf{X}_w \;. \tag{2.90}$$

In other words, any polynomial $p \in \mathbb{Z}[\mathbf{x}]$ may be uniquely expressed as a sum:

$$p = \sum_{w \in \mathfrak{S}_n} c_w \, \mathbf{X}_w \;,$$

where the c_w's belong to $\mathbf{Sym}[\mathbf{x}]$, as illustrated by the previous expansion of $x_1^5 + x_1x_2^3x_3$.

2.15.2 Multiplicative structure

We study the ring $\mathbb{Z}[x_1, \ldots, x_n]$ of polynomials seen as a free module of dimension $n!$ over the ring $\mathbf{Sym}[x_1, \ldots, x_n]$ of symmetric polynomials, with basis the simple Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_n$. For this purpose, we have to recover the multiplicative structure on this module. As explained previously, Monk's formula gives the multiplicative structure of the ring in an infinite number of variables, and does not preserve the subspace with basis the $\mathbf{X}_w, w \in \mathfrak{S}_n$. This formula does not use the fact that coefficients can belong to $\mathbf{Sym}[x_1, \ldots, x_n]$. As there is at most one term (in Monk's formula) indexed by a permutation μ outside of $\mathfrak{S}_n$, we are thus led to express certain simple Schubert polynomials as a sum with coefficients in $\mathbf{Sym}[x_1, \ldots, x_n]$ of simple Schubert polynomials indexed by permutations of $\mathfrak{S}_n$. More generally, every simple Schubert polynomial in $\mathbb{Z}[x_1, \ldots, x_n]$ has an expression in the Schubert basis $\mathbf{X}_w, w \in \mathfrak{S}_n$.

We need only consider the w' of the type $(n+1, n, \ldots, \widehat{m}, \ldots, 2, 1, m)$ because using the divided differences $\partial_1, \ldots, \partial_{n-1}$, we get all others. We consider operators from $\mathfrak{S}_{n-1}$ into $\mathfrak{S}_n$, $P_0^n, \ldots, P_k^n$ defined by:

$$\begin{aligned} P_0^n(\mu) &\longrightarrow n\; \mu \\ P_1^n(\mu) &\longrightarrow \mu_1\; n\; \mu_2 \ldots \\ \vdots\;\; &\longrightarrow \;\vdots \\ P_k^n(\mu) &\longrightarrow \mu_1 \ldots \mu_{k-1}\; n\; \mu_k \ldots \end{aligned}$$

Proposition 2.15.2 *Let n, m be two integers such that $m \leq n$. Let $\mathcal{R}(n,m)$ be the polynomial recursively defined by:*

$$\mathcal{R}(n,m) := \begin{cases} e_1 \mathbf{X}_{n,n-1,\ldots,2,1} - \sum_{k=1}^{n-1} P_{n-k}^{n}\Big(\mathcal{R}(n-1,k)\Big)_{e_j \leftarrow e_{j+1}}, & \text{if } m = n, \\ \\ P_0^n\Big(\mathcal{R}(n-1,m)\Big)_{e_j \leftarrow e_{j+1}} - P_1^n\Big(\mathcal{R}(n-1,m)\Big)_{e_j \leftarrow e_{j+2}}, & \text{otherwise.} \end{cases}$$

Then, $\mathcal{R}(n,m)$ is exactly equal to the Schubert polynomial which is indexed by the permutation $(n+1, n, \ldots, \widehat{m}, \ldots, 2, 1, m)$.

Using the divided differences $\partial_1, \ldots, \partial_{n-1}$, which commute with the e_i, one can deduce from proposition 2.15.2 the expression of any Schubert polynomial in $\mathfrak{S}_{n+1}$.

2.15.3 Applications

The `FM` package provides functions to compute with this structure of free module over symmetric polynomials. Proposition 2.15.2 has modified Monk's formula (2.77) into a formula involving only permutations in $\mathfrak{S}_n$ (`Monkfix`) and thus, allows to express the product of any Schubert polynomial $\mathbf{X}_w, w \in \mathfrak{S}_n$ by any polynomial in $x_1, \ldots, x_n$. The `FM_n` function is used to fix the number of variables:

```
ACE> Monk(3, X[3,1,4,2]);

                    - X[3, 4, 1, 2] - X[4, 1, 3, 2] + X[3, 1, 5, 2, 4]

ACE> FM_n(4);

                                        4

ACE> Monkfix(3, X[3,1,4,2]);

             - X[3, 4, 1, 2] - X[4, 1, 3, 2] + s[1, 1, 1, 1] X[1, 2, 3, 4]

                  - s[1, 1, 1] X[2, 1, 3, 4] + s[1] X[3, 1, 4, 2]
```

More generally, it allows to multiply any linear combination of Schubert polynomials with coefficients in $\mathbf{Sym}[x_1, \ldots, x_n]$ by a polynomial with coefficients also in $\mathbf{Sym}[x_1, \ldots, x_n]$. To reduce the symmetric coefficients, one uses distinguished bases of symmetric functions and corresponding algorithms. These algorithms are furnished by combinatorial properties of partitions or Young tableaux and are independent of the number of variables. The `ToXfix` function can be used to convert any expression in the basis of Schubert polynomials indexed by permutations of $\mathfrak{S}_n$:

```
ACE> FM_n(4):
ACE> ToXfix(x3^3*X[2,4,1,3]^2, collect);

     s[2, 1, 1, 1] X[3, 4, 1, 2] - s[2, 2, 1, 1] X[1, 4, 3, 2]

   + (s[2, 2, 1] + s[2, 1, 1, 1]) X[2, 4, 3, 1] + s[2, 2, 2, 2] X[1, 3, 2, 4]

   - s[2, 1, 1] X[3, 4, 2, 1] + (s[2, 2, 2] - s[2, 2, 1, 1]) X[2, 3, 4, 1]

   + s[2, 2, 2, 2] X[1, 2, 4, 3] - s[2, 2, 2, 1] X[1, 3, 4, 2]
```

We also provide the `ToXfixScal` function which gives the same result as `ToXfix` but uses scalar products rather than the modified Monk formula.

As an application of the methods described in the preceeding section, we shall mention the cohomology ring of a relative *Grassmann* variety. More explicitely, let A and B be two disjoint alphabets, $A := \{x_1, \ldots, x_m\}$ and $B :=:= \{x_{m+1}, \ldots, x_n\}$ of respective cardinal m and $n - m$. We shall denote by $A + B$ the alphabet $\{x_1, \ldots, x_m, x_{m+1}, \ldots, x_n\}$.

The above mentioned ring is the space of polynomials in $x_1, \ldots, x_n$ which are symmetrical separately in $x_1, \ldots, x_m$ and $x_{m+1}, \ldots, x_n$. This space is a free module over the ring of symmetric polynomials $\mathbf{Sym}\,[x_1, \ldots, x_n]$. It has a canonical basis (the *Schubert cycles*) of cardinal $\binom{n}{m}$, consisting of *Schur functions* [82] indexed by partitions smaller than $(n-m)^m$. For example, in the case $m = 1$, the basis consists in $x_1^0, x_1^1, \ldots, x_1^{n-1}$ and the decomposition of a polynomial into this basis is given by *Euclid*'s algorithm. For a more general m, the product of two Schur functions belonging to the basis, given by the *Littlewood-Richardson* rule [82], will involve partitions not smaller than $(n-m)^m$.

However, one can express this product of Schur functions, thanks to the preceding section, in terms of Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_n$. These Schubert polynomials must be symmetrical in $x_1, \ldots, x_m$ and therefore are Schur functions [75]. Because the indexing permutations are in $\mathfrak{S}_n$, then the corresponding Schur functions are indexed by partitions smaller or equal to $(n-m)^m$, *i.e.* belong to the canonical basis. In other words, the previous section gives a method to decompose any element of the cohomology ring in the basis of Schubert cycles. For instance, for $m = 3$, $n = 5$, the square of the Schur function $s_{21}(x_1, x_2, x_3)$ is equal to $e_4 s_2 - e_3 s_{21} + e_1^2 s_{22} + (e_4 - e_3 e_1) s_{11} + (e_4 e_1 + e_5) s_1 + (e_1^2 - e_2) s_{211} - (e_2 e_1 + e_3) s_{111} + s_{222} + 2 e_1 s_{221} + e_5 e_1$:

```
ACE> FM_n(5);

                         5

ACE> CLG_n(3);

                         3
```

```
ACE> sf := s2x_n(s[2,1]);

                  2        2                       2        2     2           2
     sf := x3  x2 + x3  x1 + 2 x3 x2 x1 + x3 x2  + x3 x1  + x2  x1 + x2 x1

ACE> x2XfixScal(sf^2, collect);

      s[2, 1, 1, 1, 1] X[1, 2, 3, 4, 5]

    - (- 2 s[1, 1, 1, 1, 1] - s[2, 1, 1, 1]) X[1, 2, 4, 3, 5]

    + s[1, 1, 1, 1] X[1, 2, 5, 3, 4] - s[2, 1, 1] X[1, 3, 4, 2, 5]

    - s[1, 1, 1] X[1, 3, 5, 2, 4] - (2 s[1, 1, 1] + s[2, 1]) X[2, 3, 4, 1, 5]

    + (s[2] + s[1, 1]) X[1, 4, 5, 2, 3] + s[2] X[2, 3, 5, 1, 4]

    + 2 s[1] X[2, 4, 5, 1, 3] + X[3, 4, 5, 1, 2]

ACE> simplify( Xfix2x(") );

             2   3       3   3            4     2   4        2   2   2
      6 x3 x2  x1  + 2 x2  x1  + 2 x3 x2 x1  + x2  x1  + 10 x3  x2  x1

          2       3     2   4     4   2      4            4           3   3
     + 6 x3  x2 x1  + x3  x1  + x2  x1  + 2 x2  x3 x1 + 2 x3  x2 x1 + 2 x3  x1

          3   3     4   2     4   2     4   2      3   2          3     2
     + 2 x3  x2  + x2  x3  + x3  x2  + x3  x1  + 6 x3  x2  x1 + 6 x3  x2 x1

          2   3           3   2
     + 6 x3  x2  x1 + 6 x3 x2  x1
```

while the Littlewood-Richardson rule gives $s_{42}+s_{411}+s_{33}+2s_{321}+s_{222}$ that involves partitions outside of (222):

```
ACE> CLG_n(3);

                                          3

ACE> Tos_n(s[2,1]^2);

           s[4, 2] + s[4, 1, 1] + s[3, 3] + 2 s[3, 2, 1] + s[2, 2, 2]
```

3
Installation

Être ce que nous sommes et devenir ce que nous
sommes capables de devenir, tel est le seul but de la vie.

Robert Louis Balfour Stevenson.

This chapter is devoted to the installation of ACE. Before starting the installation, we should be aware of the following remark: even if ACE is available for any UNIX, MS-DOS or MACINTOSH system, for any release 1, 2 or 3 of *Maple V*, we get different specific versions of the environment. That is to say, we get the UNIX version for *Maple V* release 2 which is not exactly the same as the UNIX version for *Maple V* release 3, and which is quite different from any MS-DOS version. We cannot say that sources differ between the UNIX, MS-DOS and MACINTOSH versions, but some of them have special instructions for instance for the release 3 of *Maple V*. In any case, there are some specific programs (given only with UNIX versions) to compile the environment on a UNIX system, whereas there are other programs (given only with MS-DOS versions of ACE) for the installation on a MS-DOS system. ✘

Before installing the environment, we should know the version which corresponds to our system (UNIX, MS-DOS or MACINTOSH) and its *Maple V* release (1, 2 or 3), and then refer to the corresponding section. The ACE environment is now distributed for each UNIX or MS-DOS version, as 3 archive files: the first one contains sources, the second one includes pre-computed data, and the last one contains the documentation. In the case of the MACINTOSH versions, the archive files are different for some technical reasons. The first archive just contains all sources of ACE, the second one contains all sources and pre-computed data and the last one contains the documentation. ✘

3.1 Installing ACE on a UNIX system

This section is devoted to installation of ACE in a UNIX environment.

The ACE environment is distributed as a compressed archive of files. First of all, we have to restore these files. According to the *Maple V* release ACE has to be installed for, we have the corresponding compressed archive file, `ACE1UNIX.tgz` (*Maple V* release 1), `ACE2UNIX.tgz` (*Maple V* release 2) or `ACE3UNIX.tgz` (*Maple V* release 3). These files contain the sources and on-line help files of ACE together with the installation programs.

In the following, we will suppose that we get the `ACE3UNIX.tgz`, or in other words, we have to install the ACE environment on a UNIX operating system for *Maple V* release 3. Anyway, the same work has to be done for the other archive files.

3.1.1 Restoring sources

The ACE3UNIX.tgz file is a "gziped" file, that is to say, it has been compressed by the UNIX program gzip. At first, we have to uncompress this file by the following command:

```
gunzip ACE3UNIX.tgz
```

Note that the gunzip command can be replaced by gzip -d. This command creates a new file named ACE3UNIX.tar while the ACE3UNIX.tgz has disappeared.

This file has been created by using the tar command. To see the detail on the files contained in the archive, we run the UNIX command:

```
tar tvf ACE3UNIX.tar
```

and to restore the ACE sources, we run the following UNIX command:

```
tar xvf ACE3UNIX.tar
```

that creates a directory ACE_PACK in which all files will be extracted from the archive file.

We can obtain informations about (almost) each file in the ACE_PACK directory by using the UNIX command what: for instance, in the directory ACE_PACK/BIN, we find the file runACE which can be useful to run *Maple* and load ACE at the same time, on a UNIX system. The what command gives:

```
$ what runACE
runACE:
        runACE (shell script) Wed Mar  1 14:22:23 Frh 1995
```

that indicates that runACE is a shell script created on Wednesday, the 1^{st} of March 1995.

3.1.2 Restoring pre-computed data

The DAT3UNIX.tgz file contains the pre-computed data available with ACE. Because data files are quite huge, they are distributed in a separate archive file DAT3UNIX.tgz. This archive contains both data files and installation programs. These installation programs are the same as those in ACE3UNIX.tgz but include special commands to install pre-computed data. Moreover, it is not so easy to install data on a personal computer because it uses a lot of memory for compiling the different files. Hence, we recommend not to install data on PC's except for computers like PENTIUM's with at least 16Mb of memory. Note that the file DAT3UNIX.tgz is not necessary to install ACE.

Suppose we don't want to install data, then we can skip the next lines of this part. Now, if we want to install pre-computed tables of data, then we just have to extract files from the DAT3UNIX.tgz compressed archive file. We proceed as in the case of ACE3UNIX.tgz.

First, we move the DAT3UNIX.tgz file in the directory which contains the ACE_PACK subdirectory. Then:

```
gunzip DAT3UNIX.tgz
```

to produce the DAT3UNIX.tar file from which we extract all files by:

```
tar xvf DAT3UNIX.tar
```

All files are extracted in the ACE_PACK subdirectories, that is all data files in the corresponding data subdirectories of each package, and all installation programs replacing those coming from the ACE3UNIX.tgz file. Then, to install ACE, we follow the same instructions as in the case when we don't want the data.

3.1.3 Restoring the documentation

The *Maple* on-line documentation is contained in the ACE3UNIX.tgz file but the postscript documentation and a printable text documentation for each function is distributed in the extra compressed archive file DOC3UNIX.tgz. This file is not needed for the installation, and moreover the postscript documentation and all files contained in this archive can be extracted after the installation of ACE.

As for the DAT3UNIX.tgz file, we move the DOC3UNIX.tgz file in the directory which contains the ACE_PACK subdirectory. Then, we run the UNIX command:

```
gunzip DOC3UNIX.tgz
```

to produce the DOC3UNIX.tar file from which we extract all files by:

```
tar xvf DOC3UNIX.tar
```

All files are extracted in some subdirectories of the ACE_PACK/DOCUMENT directory.

3.1.4 Description of the ACE_PACK directory

This directory contains the sources of the ACE environment, together with the documentation.

The BIN subdirectory contains the .modelinit file that is a model of the .mapleinit *Maple* file that we should modify before installing ACE. It also contains the installation program, *i.e.* the INSTALL.ACE file (which is a C-shell program), the INSTALL.SR1, INSTALL.SR2 and INSTALL.HL1, INSTALL.HL2 files that describe how to respectively install source files and help files. We also give the INSTALL.DAT file which describes how to install pre-computed data. Finally, we provide in the BIN subdirectory the runACE program (that is a shell script) which can be used to run *Maple* and to automatically load ACE.

The CPRIGHTS subdirectory contains the rights file that gives the copyright notice. To see this copyright, we can run the UNIX command: what < rights.

The DOCUMENT is empty if files of the archive file DOC3UNIX.tgz have not been extracted. Otherwise, it contains the documentation of ACE. This documentation is stored in various locations. Thus, we find in the DOCUMENT directory, the MANUAL directory in which we have a postscript version of the LaTeX manual stored in the LaTeX directory. The DATA directory stores the printable text documentation for each function. Each package has its corresponding subdirectory in which the help of each function is stored. This help is exactly the on-line *Maple* help but given as a printable (readable) file in the ASCII directory. We provide such files because *Maple* help files are not easily readable.

The PACKAGES subdirectory contains the declarations of all packages included in ACE, whereas the SOURCES subdirectory contains the sources of all functions, together with some files devoted to the installation. Moreover, the data subdirectories of each directory of the SOURCES directory contains the corresponding pre-computed tables of data. These files come from the DAT3UNIX.tgz archive file.

3.1.5 Description of a typical package

Let us choose the FM package as a typical one to present the organization of any package on the disk. It is essentially stored in the FM directory of respectively the SOURCES directory for the *Maple* sources, installation files and data, and the PACKAGES subdirectory for the *Maple* declaration of the package.

The SOURCES/FM directory contains the src directory in which the *Maple* code of all functions of FM is, the help directory that clusters the *Maple* help files for each function of the package. Moreover, the files directory is used by the installation programs like COMPILE.FM. Finally, the data subdirectory contains (if installed) the pre-computed tables of data for the FM package.

The PACKAGES/FM directory contains three files: FM that declares the package, FMext which declares all functions used in the package and finally, FMhlp that declares to *Maple* where to find the compiled help files for the FM package.

3.1.6 Step by step compilation

Having restored all the environment on the disk, we are now able to make the installation.

First of all, we have to tell *Maple* where to store the compiled files (*i.e.* the .m files). This is done in the .mapleinit file in the HOME directory, by the use of a variable per package. Since it is a non-trivial work to set all the variables, ACELib, SGLib, SPLib, TABLib and so on, we provide a model of the .mapleinit file, so-called .modelinit in the BIN directory. Here is this file for *Maple V* release 3:

```
##########
# File @(#) .modelinit (Maple) Sun Jan  1 17:01:45 Frh 1995
##########
#
############################################################
#                                                          #
# Insert the following lines in the $HOME/.mapleinit file. #
#                                                          #
############################################################
```

```
#
## ACE package...
#
####################################################################
#                                                                  #
# Modify the following line so that the ACELib variable  contains  #
# the directory where one wants maple to store the compiled files. #
#                                                                  #
####################################################################
                     #
                     #
                  #######
                   #####
                    ###
                     #
#
ACELib:=`/people/roger/maple/lib`:
#
BNLib  :=``.ACELib.`/ACE`:            BNALib :=``.ACELib.`/ACE`:
CLGLib :=``.ACELib.`/ACE`:            CGLib  :=``.ACELib.`/ACE`:
ERRLib :=``.ACELib.`/ACE`:            FMLib  :=``.ACELib.`/ACE`:
FOCKLib:=``.ACELib.`/ACE`:            HEKALib:=``.ACELib.`/ACE`:
IDCALib:=``.ACELib.`/ACE`:            NCALib :=``.ACELib.`/ACE`:
NCSFLib:=``.ACELib.`/ACE`:            SGALib :=``.ACELib.`/ACE`:
SGLib  :=``.ACELib.`/ACE`:            SHRILib:=``.ACELib.`/ACE`:
SPLib  :=``.ACELib.`/ACE`:            SYMFLib:=``.ACELib.`/ACE`:
TABLib :=``.ACELib.`/ACE`:            TEXLib :=``.ACELib.`/ACE`:
TYPLib :=``.ACELib.`/ACE`:
#
ACELibSeq:=SGLib,  SPLib,  SYMFLib, TABLib,  TYPLib,
           HEKALib, ERRLib, CLGLib, TEXLib,  SGALib,
           NCALib, IDCALib, NCSFLib, SHRILib, FMLib,
           BNLib, BNALib, CGLib, FOCKLib:
#
## Where to find the packages...for Maple V Release 1
#
# _liblist:=[ACELib, ACELibSeq]:
#
## Where to find the packages...for Maple V Release 2
#
# libname:=libname, ACELib, ACELibSeq:
#
## Where to find the packages...for Maple V Release 3
#
  libname:=libname, ACELib, ACELibSeq:
```

Thus, to tell *Maple* where to compile (and later, where to find) the packages, we copy the `.modelinit` file into the existing `.mapleinit` file in the `HOME` directory. If we don't have any `.mapleinit` file, then we create it by directly copying the `.modelinit` file into a new `.mapleinit` file. Then, we modify the `ACELib` variable in the `.mapleinit` file, setting it to an **existing** directory. Note that, according to the given `.modelinit` file, all packages will be compiled into subdirectories of the `ACELib` directory but even if it is not necessary, it is easier to manage.

Global installation

Having modified the `.mapleinit` file, we are ready to install all packages by using the `INSTALL.ACE` program in the `BIN` directory. Being in this directory, enter the UNIX command, `csh INSTALL.ACE` which prints some information about the installation among which it is said that this program takes as argument the directory where the packages must be installed, and this directory is exactly the same as the directory specified by the `ACELib` variable in the `.mapleinit` file. Then, to install the environment, enter a UNIX command like:

```
csh INSTALL.ACE /people/roger/maple/lib
```

Individual installation

We also provide the possibility to compile each package individually. For this purpose, we give for each package its own installation program in its corresponding directory. For instance, we can compile the `CLG` package by running a UNIX command that looks like:

```
csh COMPILE.CLG /people/roger/maple/lib/ACE
```

As usually, we enter the UNIX command `csh COMPILE.CLG` to get more information and for instance to see that we can compile one file among all by running a command like:

```
csh COMPILE.CLG /people/roger/maple/lib/ACE src/Toe
```

3.1.7 Having some troubles

There could be some troubles whether for instance the *Maple* program is not in the `PATH`, that is to say, the *Maple* command is unknown for the system. Since we run the `csh` shell to interpret the installation program, we should have the *Maple* program in the `PATH` variable while running this shell so that this trouble could be due to an incomplete `.cshrc` file (which is the configuration file of the `csh` shell program) in the `HOME` directory. Please refer to a documentation about `csh` to tell it where to find the *Maple* program: for instance, enter the UNIX command `man csh` that should give some hints about the problem.

Whenever there is a `writting error` message from *Maple* that looks like:

```
Error, could not open ... for writing
```

be sure that the corresponding directory exists. If it does not, be sure of having correctly set the corresponding variable in the `.mapleinit` file.

3.1.8 Using the environment

Several possibilities are available. First of all, we can run *Maple* by the (usual) UNIX command `maple`, and then to load the complete `ACE` environment, that is to load all packages, we enter the *Maple* command `with(ACE);`. We can only load a specific package, say `SYMF`, by the *Maple* command `with(SYMF);`.

Another way to use the `ACE` environment is to execute the `runACE` UNIX command located in the `BIN` directory. It runs *Maple*, automatically prints the `ACE` banner and loads the `ACE` environment. We can add the `BIN` directory to our `PATH` variable to have the `runACE` command accessible from anywhere. It can done for instance in the corresponding shell configuration file. We also have to put the correct rights on the `runACE` file, with a command like: `chmod 700 runACE`.

3.2 Installing ACE on a MS-DOS system

This section deals with the installation of `ACE` in a MS-DOS environment. First, we have to extract files from the compressed archive files. According to the *Maple V* release we want `ACE` to be installed for, we get the corresponding compressed archive file, `ACE1DOS.tgz` (*Maple V* release 1), `ACE2DOS.tgz` (*Maple V* release 2) or `ACE3DOS.tgz` (*Maple V* release 3).

In the following, we will suppose that we get the `ACE3DOS.tgz`. `ACE` environment on a MS-DOS operating system for *Maple V* release 3. In any case, the same work has to be done for other releases.

3.2.1 Restoring sources

The `ACE3DOS.tgz` is a "gziped" file, that is to say, it has been compressed by the UNIX program `gzip`. We assume that this file is stored on a floppy disk named `a:` in the following.

To simplify the restoring phase on a MS-DOS system, we provide a program that can be used to uncompress the archive file and extract from it, in the same time, all files: this is the `tar.exe` file given on the floppy disk which is a "special" version of a UNIX compatible `tar` program including the `gunzip` program.

At first, being on the hard disk in the right directory, and assuming that we get this "special" `tar` program, we can uncompress the archive and extract all files by the following command:

```
a:tar.exe .xvf a:ACE3DOS.tgz
```

that creates a directory `ACE_PACK`. If we do not have this `tar.exe` program, files can be restored by first uncompressing the archive file and then extracting them from it (refer to the UNIX installation section).

3.2.2 Restoring pre-computed data

The `DAT3DOS.tgz` file contains the pre-computed data available with `ACE`. Because data files are quite huge, they are distributed in a separate archive file `DAT3DOS.tgz`. This archive contains both data files and installation programs. These installation programs are the same as those in `ACE3DOS.tgz` but include special commands to install pre-computed data. Moreover, it is not so easy to install data on a personal computer because it uses a lot of memory for compiling the different files. Hence, we recommend not to install data on PC's except for computers like PENTIUM's with at least 16Mb of memory. Note that the file `DAT3DOS.tgz` is not necessary to install `ACE`.

Suppose we don't want to install data, then we can skip the next lines of this part. Now, if we want to install pre-computed tables of data, then we just have to extract files from the `DAT3DOS.tgz` compressed archive file. We proceed as in the case of `ACE3DOS.tgz`.

First, we copy the `DAT3DOS.tgz` file in the directory which contains the `ACE_PACK` subdirectory. Then:

```
a:tar.exe .xvf a:DAT3DOS.tgz
```

to extract all files in the `ACE_PACK` subdirectories, that is all data files in the corresponding `data` subdirectories of each package, and all installation programs replacing those coming from the `ACE3DOS.tgz` file. Then, to install `ACE`, we follow the same instructions as in the case when we don't want the data.

3.2.3 Description of the `ACE_PACK` directory

This directory contains all sources of the `ACE` environment, together with the documentation.

The `BIN` subdirectory contains the `MAPLE.MOD` file which is a model of the `MAPLE.INI` *Maple* file that we should modify to install `ACE`. It also contains the installation program, *i.e.* the `INSTALL.BAT` file (that is a batch file), the `INSTALL.SR1`, `INSTALL.SR2` files and `INSTALL.HL1`, `INSTALL.HL2` files which describe how to respectively install source files and help files.

The `CPRIGHTS` subdirectory contains the `rights` file that gives the copyright notice. To see this copyright, we can run the UNIX command: `what < rights`.

The `DOCUMENT` is empty if files of the archive file `DOC3DOS.tgz` have not been extracted. Otherwise, it contains the documentation of `ACE`. This documentation is stored in various locations. Thus, we find in the `DOCUMENT` directory, the `MANUAL` directory in which we have a postscript version of the LaTeX manual stored in the `LaTeX` directory. Another directory in the `DOCUMENT` directory, is the `DATA` one that stores the printable text documentation for each function. Each package has its corresponding subdirectory in which the help of each function is stored. This help is exactly the on-line *Maple* help but given as a printable (readable) file in the `ASCII` directory. We provide such files because *Maple* help files are not easily readable.

The `PACKAGES` subdirectory contains the declarations of all packages included in `ACE`, whereas the `SOURCES` subdirectory contains the sources of all functions, together with some files devoted to the installation. Moreover, the `data` subdirectories of each directory of the `SOURCES` directory contains the corresponding pre-computed tables of data. These files come from the `DAT3DOS.tgz` archive file.

3.2.4 Description of a typical package

Let us choose the `FM` package as a typical one to present the organization of any package on the disk. It is essentially stored in the `FM` directory of respectively the `SOURCES` directory for the *Maple* sources, installation files and data, and the `PACKAGES` subdirectory for the *Maple* declaration of the package.

The `SOURCES/FM` directory contains the `src` directory in which the *Maple* code of all functions of `FM` is, the `help` directory that clusters the *Maple* help files for each function of the package. Moreover, the `files` directory is used by the installation programs like `COMPFM.BAT`. The `FM.ini` file gives the lines that must be added to the `MAPLE.INI` file in the *Maple* library directory[(1)]. Finally, the `data` subdirectory contains (if installed) the pre-computed tables of data for the `FM` package.

[(1)]which is usually the `\MAPLEV\LIB` directory.

The PACKAGES/FM directory contains three files: FM that declares the package, FMEXT which declares all functions used in the package and finally, FMHLP that declares to *Maple* where to find the compiled help files for the FM package.

3.2.5 Step by step compilation

Having restored all the environment on the disk, we are now able to make the installation.

First of all, we have to tell *Maple* where to store the compiled files (*i.e.* the .m files). This is done in the MAPLE.INI file in the *Maple* library directory (by default \MAPLEV\LIB), by using one variable per package. Since it is a non-trivial work to set all the variables, ACELib, SGLib, SPLib, TABLib and so on, we provide a model of rows that must be added in the MAPLE.INI file, so-called MAPLE.MOD in the BIN directory. Here is this file for *Maple V* release 3:

```
##########
# File @(#) MAPLE.MOD (Maple) Wed May 10 09:48:17 Fre 1995
##########
#
################################################################
#                                                              #
# Insert the following lines in the \MAPLEV\LIB\maple.ini file. #
#                                                              #
################################################################
#
## ACE package...
#
###################################################################
#                                                                 #
# Modify the following line so that the ACELib variable  contains #
# the directory where one wants maple to store the compiled files. #
#                                                                 #
###################################################################
               #
               #
            #######
             #####
              ###
               #
#
ACELib:=`/MAPLEV/LIB`:
#
BNLib  :=``.ACELib.`/ACE`:              BNALib :=``.ACELib.`/ACE`:
CLGLib :=``.ACELib.`/ACE`:              CGLib  :=``.ACELib.`/ACE`:
ERRLib :=``.ACELib.`/ACE`:              FMLib  :=``.ACELib.`/ACE`:
FOCKLib:=``.ACELib.`/ACE`:              HEKALib:=``.ACELib.`/ACE`:
```

```
IDCALib:=``.ACELib.`/ACE`:              NCALib :=``.ACELib.`/ACE`:
NCSFLib:=``.ACELib.`/ACE`:              SGALib :=``.ACELib.`/ACE`:
SGLib  :=``.ACELib.`/ACE`:              SHRILib:=``.ACELib.`/ACE`:
SPLib  :=``.ACELib.`/ACE`:              SYMFLib:=``.ACELib.`/ACE`:
TABLib :=``.ACELib.`/ACE`:              TEXLib :=``.ACELib.`/ACE`:
TYPLib :=``.ACELib.`/ACE`:
#
ACELibSeq:=SGLib,  SPLib,  SYMFLib, TABLib,  TYPLib,
           HEKALib, ERRLib, CLGLib, TEXLib,  SGALib,
           NCALib, IDCALib, NCSFLib, SHRILib, FMLib,
           BNLib, BNALib, CGLib, FOCKLib:
#
## Where to find the packages...for Maple V Release 1
#
# _liblist:=[ACELib, ACELibSeq]:
#
## Where to find the packages...for Maple V Release 2
#
# libname:=libname, ACELib, ACELibSeq:
#
## Where to find the packages...for Maple V Release 3
#
  libname:=libname, ACELib, ACELibSeq:
```

Thus, to tell *Maple* where to compile (and later, where to find) the packages, we insert all lines contained in the `MAPLE.MOD` file into the existing `MAPLE.INI` file in the *Maple* library directory (avoid to insert these lines at the end of the `MAPLE.INI` file)

Then, we modify the `ACELib` variable in the `MAPLE.INI` file, setting it to an **existing** directory that differs from the `\MAPLEV\LIB` one, but we recommand not to change this variable since one can have some troubles with the installation program to create the needed directories.

Note that, according to the given `MAPLE.MOD` file, all packages will be compiled into subdirectories of the `ACELib` directory but even if it is not necessary, it is easier to manage.

Global installation

Having modified the `MAPLE.INI` file, we are ready to install all packages by using the `INSTALL.ACE` program in the `BIN` directory. Being in this directory, enter the MS-DOS command, `INSTALL.BAT` which prints some information about the installation among which it is said that this program takes as argument the directory where the packages must be installed, and this directory is exactly the same as the directory specified by the `ACELib` variable in the `MAPLE.INI` file. Then, to install the environment, enter a MS-DOS command like:

```
INSTALL.BAT \MAPLEV\LIB
```

3.2.6 Using the environment

We can use ACE by running *Maple* and then loading the complete ACE environment (all packages) by the *Maple* command `with(ACE);`. We can only load a specific package, say SYMF, by the *Maple* command `with(SYMF);`.

3.3 Installing ACE on a MACINTOSH system

✘

This section deals with the installation of ACE on a MACINTOSH. First, we have to extract files from the BinHexed archive files. According to the *Maple V* release we want ACE to be installed for, we get the corresponding compressed archive file, ACE1MAC.sit.hqx (*Maple V* release 1), ACE2MAC.sit.hqx (*Maple V* release 2) or ACE3MAC.sit.hqx (*Maple V* release 3). Once downloaded, we have to extract all files from the archive using the Stuffit Expander application. This application is shareware available on every **infomac archive** on the Internet.

Remark. Note that we have not tested the installation of the MACINTOSH version on several systems. We only have installed ACE on a MACINTOSH running a very old release of *Maple V*. Do not hesitate to contact us if you have any trouble with the MACINTOSH installation and furthermore please send us emails if you have ideas to improve the installation of ACE on a MACINTOSH system. Because we had not a recent version of *Maple V*, we were not able to do a better automatic installation program.

In the following, we will suppose that we get the ACE3MAC.sit.hqx, or in other words, we have to install the ACE environment on a MACINTOSH system for *Maple V* release 3.

3.3.1 Restoring sources

The ACE3MAC.sit.hqx file is a BinHexed archive. To extract all files from the archive, we have to **drop** the ACE3MAC.sit.hqx file on the Stuffit Expander icon. It will create a folder ACE_PACK in which all files will be extracted from the archive file. It will also create a folder ACE (containing subfolders but without any file), and a MAPLEINIT file. These two folders and the MAPLEINIT have to be moved into the lib folder next to the Maple folder, before installing the ACE environment.

3.3.2 Restoring pre-computed data

These data are contained in the BinHexed archive ACE+DAT3MAC.sit.hqx. In order to install these data, we have to follow the same instructions as for the ACE3MAC.sit.hqx archive but on the file ACE+DAT3MAC.sit.hqx. It will then extract both sources and data.

3.3.3 Restoring the documentation

The *Maple* on-line documentation is contained in the ACE3MAC.sit.hqx (or ACE+DAT3MAC.sit.hqx) file but the postscript documentation and a printable text documentation for each function is distributed in the extra BinHexed archive file DOC3MAC.sit.hqx. This file is not needed for the installation, and moreover the postscript documentation and all files contained in this archive can be extracted after the installation of ACE.

All files of the DOC3MAC.sit.hqx are extracted in some subdirectories of a ACE_DOC folder.

3.3.4 Description of the ACE_PACK and ACE_DOC folder

The ACE_PACK folder contains all sources of the ACE environment, whereas the ACE_DOC folder contains the postscript and readable text documentation.

The ACE_PACK/BIN folder contains the installation program, *i.e.* the INSTALL.ACE file (which is a *Maple* file), the INSTALL.SR1, INSTALL.SR2 files and INSTALL.HL1, INSTALL.HL2 files which describe how to respectively install source files and help files.

The ACE_PACK/CPRIGHTS subfolder contains the rights file that gives the copyright notice.

The ACE_PACK/PACKAGES subfolder contains the declarations of all packages included in ACE, whereas the ACE_PACK/SOURCES subfolder contains the sources of all functions, together with some files devoted to the installation. Moreover, the data subfolders of each folder of the ACE_PACK/SOURCES folder contains the corresponding pre-computed tables of data. All files contained in the data folders come from the ACE+DAT3MAC.sit.hqx archive file.

The ACE_DOC folder contains the documentation of ACE. This documentation is stored in various locations. Thus, we find the ACE_DOC/MANUAL folder in which we have a postscript version of the LaTeX manual stored in the LaTeX folder. Another folder in the ACE_DOC folder, is the DATA one that stores the printable text documentation for each function. Each package has its corresponding subfolder in which the help of each function is stored. This help is exactly the on-line *Maple* help but given as a printable (readable) file in the ASCII folder. We provide such files because *Maple* help files are not easily readable.

3.3.5 Description of a typical package

Let us choose the FM package as a typical one to present the organization of any package on the disk. It is essentially stored in the FM folder of respectively the SOURCES folder for the *Maple* sources, installation files and data, and the PACKAGES subfolder for the *Maple* declaration of the package.

The SOURCES/FM folder contains the src folder in which the *Maple* code of all functions of FM is, the help folder that clusters the *Maple* help files for each function of the package. Finally, the data subfolder contains (if installed) the pre-computed tables of data for the FM package.

The PACKAGES/FM folder contains three files: FM that declares the package, FMext which declares all functions used in the package and finally, FMhlp that declares to *Maple* where to find the compiled help files for the FM package.

3.3.6 Step by step compilation

Having restored all the environment on the disk, we are now able to make the installation.

First of all, we have to tell *Maple* where to store the compiled files (*i.e.* the .m files). This is done in the MAPLEINIT file in the *Maple* library directory (by default :lib), by using one variable per package. Since it is a non-trivial work to set all the variables, ACELib, SGLib, SPLib, TABLib and so on, we provide a model of the MAPLEINIT file that we have previously moved into the lib folder next to the Maple folder. Here is this file for *Maple V* release 3:

```
##########
# File @(#) MAPLEINIT (Maple) Wed Feb  7 17:29:13 MET 1996
##########
#
###########################################################
#                                                         #
```

```
# Insert the following lines in the MAPLEINIT file in #
# the folder  given  by  the libname MAPLE variable.  #
# It should be by default :lib.                       #
#                                                     #
#######################################################
#
## ACE package...
#
#########################################################
#                                                       #
# DO NOT  modify the following line  except  if  Maple  #
# library has been moved and is not in the folder :lib. #
#                                                       #
#########################################################
               #
               #
            #######
             #####
              ###
               #
#
ACELib:=``.libname:
#
BNLib  :=``.ACELib.`:ACE`:    BNALib :=``.ACELib.`:ACE`:
CLGLib :=``.ACELib.`:ACE`:    CGLib  :=``.ACELib.`:ACE`:
ERRLib :=``.ACELib.`:ACE`:    FMLib  :=``.ACELib.`:ACE`:
FOCKLib:=``.ACELib.`:ACE`:    HEKALib:=``.ACELib.`:ACE`:
IDCALib:=``.ACELib.`:ACE`:    NCALib :=``.ACELib.`:ACE`:
NCSFLib:=``.ACELib.`:ACE`:    SGALib :=``.ACELib.`:ACE`:
SGLib  :=``.ACELib.`:ACE`:    SHRILib:=``.ACELib.`:ACE`:
SPLib  :=``.ACELib.`:ACE`:    SYMFLib:=``.ACELib.`:ACE`:
TABLib :=``.ACELib.`:ACE`:    TEXLib :=``.ACELib.`:ACE`:
TYPLib :=``.ACELib.`:ACE`:
#
ACELibSeq:=SGLib,  SPLib, SYMFLib,  TABLib,  TYPLib,
           HEKALib,  ERRLib, CLGLib, TEXLib, SGALib,
           NCALib, IDCALib, NCSFLib, SHRILib, FMLib,
           BNLib, BNALib, CGLib, FOCKLib:
#
## Where to find the packages...for Maple V Release 1
#
# _liblist:=[ACELib, ACELibSeq]:
#
## Where to find the packages...for Maple V Release 2
#
# libname:=libname, ACELib, ACELibSeq:
#
## Where to find the packages...for Maple V Release 3
#
  libname:=libname, ACELib, ACELibSeq:
```

We do not in general have to modify the **ACELib** variable in the **MAPLEINIT** file. *Maple* should find installed libraries.

Global installation

We are ready to install all packages by using the `INSTALL.ACE` *Maple* program in the `ACE_PACK/BIN` folder. Run *Maple* and enter the following *Maple* command:

```
read `:lib:ACE_PACK:BIN:INSTALL.ACE`;
```

which prints instructions to install `ACE`. To follow the instructions, we recommand to select all lines (output of the previous *Maple* command), put them as `input` lines, and save them as a *Maple* `Worksheet`. Then, quit and click on the created icon to run *Maple* and **execute one block of commands each time**. For that purpose, we recommand `cut` and `copy` technic because we have to run *Maple* about 20 times[2]. Blocks of commands are separated by an empty line and each block tells *Maple* to terminate the session each time.

Remark. We can see the result of each block of commands by clicking on `Cancel` when *Maple* asks us to confirm we want to terminate the session. This can be useful to detect that there is a problem.

3.3.7 Using the environment

We can use `ACE` by running *Maple* and then loading the complete `ACE` environment (all packages) by the *Maple* command `with(ACE);`. We can only load a specific package, say `SYMF`, by the *Maple* command `with(SYMF);`.

[2] We did not succeed in having an easier way to install `ACE`, but fill free to inform us if you have better ideas.

4 Examples

This chapter presents some examples provided with the environment. These examples are given in the subdirectories of the `EXAMPLES` directory. First, we compute Grothendieck polynomials and Key polynomials, then a section deals with projective degrees of Schubert varieties and Todd classes of flag varieties. The next section is devoted to computations with the `NCSF` package, followed by a section about the combinatorial interpretation of Schur functions as sums of tableaux of a given shape. The last section deals with generating functions for Schubert polynomials. ✘

4.1 Grothendieck polynomials

We compute Grothendieck polynomials using the action of the Demazure operators of the idCoxeter algebra, on the Grothendieck polynomial corresponding to the maximal permutation. The following lines illustrates thiscomputation.

The `GrothMax` function returns the Grothendieck polynomial corresponding to the maximal permutation of $\mathfrak{S}_n$ (it corresponds to the Schubert variety which is a point):

```
GrothMax:=proc(n)
  local
    gm,   # the result...
    i, j; # variables for loop...
  option remember;
  gm:=1;
  for i from 1 to n-1 do
    for j from 1 to n-i do
      gm:=gm*(1 - (y.j)/(x.i))
    od
  od;
  RETURN(gm)
end;
```

`Groth`(*perm*) computes the Grothendieck polynomial of index *perm* (*i.e.* the class of the structure sheaf of Schubert variety X_{perm} in the Grothendieck ring of the flag manifold):

```
Groth:=proc(perm)
  local
    n,     # the degree of the symmetric group...
    wperm; # = MultPerm(InvPerm(perm), LastPerm(perm))...
  n:=nops(perm);
  wperm:=MultPerm(InvPerm(perm), LastPerm(n));
  RETURN(simplify(IdcaOnPol(Demazure(wperm), GrothMax(n))))
end;
```

The next function computes the table of all Grothendieck polynomials for $\mathfrak{S}_n$:

```
TableGG:=proc(n)
  local
    p, # a permutation...
    t; # the result...
  t:=table();
  for p in ListPerm(n) do
    t[p]:=Groth(p)
  od;
  RETURN(eval(t))
end;
```

The following command demonstrates the use of this function:

```
ACE> TableGG(3);

                          table([
                                          (x1 - y2) (- x1 + y1)
                              [3, 1, 2] = - ---------------------
                                                     2
                                                   x1
                                          - x1 + y1
                              [2, 1, 3] = - ---------
                                              x1
                                        (- x1 + y1) (- x2 + y1)
                              [2, 3, 1] = -----------------------
                                                 x1 x2
                                        (- x1 + y1) (x1 - y2) (- x2 + y1)
                              [3, 2, 1] = ---------------------------------
                                                        2
                                                      x1  x2
```

```
                    [1, 2, 3] = 1
                                   - x2 x1 + y2 y1
                    [1, 3, 2] = - ---------------
                                       x1 x2
                ])
```

When specializing $y_i \longrightarrow 1$ and $x_i \longrightarrow 1/(1-x_i)$, Grothendieck polynomials become equal to Schubert polynomials $\mathbf{X}_{perm}+$ terms of higher degree. The following function computes the table of all Grothendieck polynomials for $\mathfrak{S}_n$ expressed in the $\mathbf{X}$ Schubert basis:

```
TableGG2X:=proc(n)
  local
    i, # variable for sequence...
    t; # a table...
  t:=TableGG(n);
  t:=subs(seq(y.i=1,i=1..n-1), eval(t));
  t:=subs(seq((x.i)=1/(1-(x.i)),i=1..n-1), eval(t));
  RETURN(map(x2X, map(simplify, eval(t))))
end;
```

The next example illustrates the use of this function for the symmetric group $\mathfrak{S}_3$:

```
ACE> TableGG2X(3);

                    table([
                        [3, 1, 2] = X[3, 1, 2]
                        [2, 1, 3] = X[2, 1]
                        [2, 3, 1] = X[2, 3, 1]
                        [3, 2, 1] = X[3, 2, 1]
                        [1, 2, 3] = X[1]
                        [1, 3, 2] = - X[2, 3, 1] + X[1, 3, 2]
                    ])
```

4.2 Key polynomials

A key polynomial is the image of a dominant monomial $x_1^{i_1}x_2^{i_2}\ldots$, $i_1 \geq i_2 \geq \cdots$, under a Demazure operator. The following program gives a way to compute these polynomials using the `ACE` environment.

The `ListDom` function returns the list of all partitions which are smaller than $[n-1, n-2, ..., 0]$:

```
ListDom:=proc(n)
  local
    i; # variable for sequence...
  RETURN(map(proc(l, n) [op(l), 0$(n-nops(l))] end,
            `SYMF/ListPartIn`([seq(-i, i=-(n-1)..-1)]), n))
end;
```

`KeyPol`(*cod*) computes the key polynomial corresponding to a permutation given by its code:

```
KeyPol:=proc(cod)
  local
    i,    # variable for sequence...
    j,    # variable for loop...
    pos,  # a position...
    perm, # a permutation...
    rcod, # a code...
    tmp;  # a temporary variable...
  rcod:=sort(cod,proc(x,y) evalb(x>=y) end);
  perm:=NULL;
  tmp:=rcod;
  for j from 1 to nops(cod) do
    member(cod[j], tmp, 'pos');
    perm:=perm, pos;
    tmp:=subsop(pos=-1, tmp);
  od;
  perm:=[perm];
  RETURN(IdcaOnPol(Demazure(`SG/InvPerm`(perm)),
                   convert([seq(x.i^(rcod[i]), i=1..nops(cod))],`*`)))
end;
```

The following function returns the table of all key polynomials for the symmetric group $\mathfrak{S}_n$, the table being indexed by codes of permutations:

```
TableK:=proc(n)
  local
    cod, # a code...
    t;   # a table...
  t:=table();
  for cod in Interval(LastPerm(n)) do
    t[cod]:=KeyPol(cod)
  od;
  RETURN(eval(t))
end;
```

The following line gives the table of all key polynomials for $\mathfrak{S}_3$:

```
ACE> TableK(3);
                              table([
                                                  2
                                  [2, 0, 0] = x1
                                  [1, 1, 0] = x2 x1
                                  [1, 0, 0] = x1
                                  [0, 1, 0] = x2 + x1
                                  [0, 0, 0] = 1
                                                  2
                                  [2, 1, 0] = x1  x2
                              ])
```

Schubert polynomials and key polynomials coincide in the case of vexillary permutations. The following function returns the table of key polynomials in the **Y** Schubert basis.

```
TableKY:=proc(n)
  RETURN(eval(map(ToY, TableK(n))))
end;
```

Since the permutation $w := [2,1,4,3]$ (whose Lehmer code is $[1,0,1,0]$) is the only non-vexillary permutation of $\mathfrak{S}_4$, we have the following result:

```
ACE> t:=TableKY(4):
ACE> t[ [1,0,1,0] ];

                            - Y[2, 0, 0] + Y[1, 0, 1, 0]
```

4.3 Projective degree and Todd class

4.3.1 Projective degree of a Schubert variety

The class of an hyperplane section of the flag manifold $\mathcal{F}(\mathbb{C}^n)$, for the Plücker embedding, is equal to $h := (n-1)x_1 + (n-2)x_2 + \cdots + x_{n-1}$. The projective degree of a Schubert variety is by definition the coefficient of $\mathbf{X}_\omega$ in the product $\mathbf{X}_{perm} \times h^{(\binom{n}{2} - l(perm))}$ in which ω denotes the maximal permutation of $\mathfrak{S}_n$ and $l(perm)$ the length of the permutation $perm$:

```
h:=proc(n)
  local
    i; # variable for sequence...
  option remember;
  RETURN(convert([seq((n-i)*x.i, i=1..n-1)],`+`))
end;

ProjDegree:=proc(perm)
  local
    r,     # the result...
    n,     # degree of the symmetric group...
    sFlag, # a value...
    d;     # the projective degree of the Schubert variety...
  n:=nops(perm);
  d:=n*(n-1)/2 - Perm2Length(perm);
  sFlag:=_Sn;
  Flag(n);
  r:=subs(X[op(LastPerm(n))]=1, ToX(h(n)^d*X[op(perm)]));
  if (sFlag<>-1) then Flag(sFlag) else Flag() fi;
  RETURN(r)
end;
```

4.3.2 Todd class of the flag variety

The total Todd class of the flag variety $\mathcal{F}(\mathbb{C}^n)$ is by definition equal to the class of $e^{\sum_{1\leq i<j\leq n}(x_i-x_j)}$, in the cohomology ring, and thus coincides with $e^{h(n)}$ where $h(n)$ has been defined in the previous subsection. The following function computes this Todd class:

```
Todd:=proc(n)
  local
    pol,   # a polynomial...
    hk,    # power of h...
    sFlag, # a value...
    k;     # variable for loop...
  pol:=1;
  hk:=1;
  sFlag:=_Sn;
  Flag(n);
  for k from 1 to n*(n-1)/2 do
    hk:=ToX(hk*h(n));
    pol:=pol + hk/factorial(k)
  od;
  if (sFlag<>-1) then Flag(sFlag) else Flag() fi;
  RETURN(pol)
end;
```

4.4 Some computations with NCSF

It has been shown by Klyachko that for a primitive n-th root of unity ε, the element:

$$\kappa_n = \frac{1}{n} \sum_{\sigma \in \mathfrak{S}_n} \varepsilon^{\mathrm{maj}(\sigma)} \sigma \quad ,$$

is a Lie idempotent. In terms of noncommutative symmetric functions, this element is the specialization at $q = \varepsilon$ of the symmetric function $\frac{1}{n} K_n(q)$, where $K_n(q)$ is given by the generating series:

$$\sum_{n \geq 0} \frac{K_n(q)}{(q)_n} = \overleftarrow{\prod_{k \geq 0}} \sigma(q^k) \quad .$$

Let us check these properties on the example of κ_4. The corresponding symmetric function is constructed by the sequence:

```
ACE> C:=ListCompo(4):
ACE> K:=0:
ACE> for x in C do K:=K+ q^MajorIndex(x)*R[op(x)] od:
ACE> K:=K/4:
ACE> K:=subs(q=I,K):
```

To check that it is an idempotent, we enter:

```
ACE> Internal(K,K) - ToS(K);
                                  0
```

We have to compare with `ToS(K)` because the output of `Internal` is given in the S-basis.

To see that κ_4 projects onto the Free Lie algebra, we may compute its left and right products with a known Lie idempotent, for example Dynkin's element $\theta_4 := \frac{1}{4}[[[1,2],3],4]$, which corresponds to the symmetric function $\frac{1}{4}\Psi_4$:

```
ACE> T:=ToS(Ps[4]/4):
ACE> Internal(T,K)-K:
ACE> ToR(");
                                  0

ACE> Internal(K,T)-T:
ACE> ToR(");
                                  0
```

The symmetric functions of the virtual alphabet $A/(1-q)$ are defined by the equation:

$$S_n\left(\frac{A}{1-q}\right) = \frac{K_n(q)}{(q)_n} \quad ,$$

where A is just a label. In the package, this transformation is computed by the function `SpDirect`:

```
ACE> SpDirect(S[3], q):
ACE> ToR( (1-q)*(1-q^2)*(1-q^3)*");

                                          2                          3
                    R[3] + R[2, 1] q  + R[1, 2] q + R[1, 1, 1] q
```

The inverse transformation is given by the function `SpInverse`. For example, $\Theta_n(q) := S^n((1-q)A)/(1-q)$ corresponds in the descent algebra to the q-bracket operator:

```
ACE> SpInverse(S[4],q):
ACE> ToR("/(1-q));

                                            2                         3
               R[4] - q R[1, 3] + R[1, 1, 2] q  - R[1, 1, 1, 1] q
```

One can show that the element:

$$\varphi_n(q) = \frac{1-q^n}{n}\Psi_n\left(\frac{A}{1-q}\right) \quad ,$$

is a Lie idempotent [41, 49], interpolating between several known ones:

```
ACE> SpDirect(S[4],q):
ACE> P:=ToR("/(1-q));
                              2                                            2
                             q  R[3, 1]       R[2, 2] q      R[2, 1, 1] q
        P := 1/4 R[4] - 1/4 ---------- - 1/4 ---------- + 1/4 -------------
                              2                2                 2
                             q  + q + 1       q  + q + 1        q  + q + 1

                R[1, 3]         q R[1, 2, 1]       R[1, 1, 2]
        - 1/4 ---------- + 1/4 ------------ + 1/4 ---------- - 1/4 R[1, 1, 1, 1]
                2                2                  2
               q  + q + 1       q  + q + 1         q  + q + 1

ACE> subs(q=1,P):ToPh(");
                                       1/4 Ph[4]

ACE> subs(q=I,P);

          1/4 R[4] - 1/4 I R[3, 1] - 1/4 R[2, 2] + 1/4 I R[2, 1, 1]

               + 1/4 I R[1, 3] + 1/4 R[1, 2, 1] - 1/4 I R[1, 1, 2]

                                                    - 1/4 R[1, 1, 1, 1]
```

This leads to a q-deformation of the Eulerian idempotents, which can be computed by the following program:

```
#
# Collects terms indexed by compositions of the same length.
#
split:=proc(x,t);
  if member(whattype(x),{`*`,`+`}) then map(split,x,t)
  elif whattype(x)=`indexed` then t^nops(x)*x else x fi
end;
#
# Generating function of the q-Eulerians.
#
GFqE:=proc(n,q,t) local k,res; option remember;
  res:=ToPs(SpInverse(S[n],q));
  res:=split(res,t); res:=SpDirect(res,q);
end;
#
# Individual q-Eulerians in the R-basis.
#
qEuler:=proc(n,k,q);
  ToR( coeff(expand(GFqE(n,q,`t`)), `t`, k) );
end;
```

4.5 Combinatorial interpretation of Schur functions

Schur functions can be regarded as sum of tableaux of a given shape. The `SchurTab` function returns the list of all tableaux of a given shape on the alphabet $1,\ldots,$`card`:

```
ListEval:=proc(card, weight)
  local
    i,     # variable for sequence...
    compo, # variable for loop...
    res;   # the result...
  res:=NULL;
  ListCompo(weight+card) ;
  for compo in " do
    if (nops(compo)=card) then
      res:=res, [seq(compo[i]-1, i=1..card)]
    fi
  od;
  RETURN([res])
end;

SchurTab:=proc(part, card)
  local
    levalu, # list of evaluations...
    evalu,  # variable for loop...
    res;    # the result...
```

```
  levalu:=ListEval(card, convert(part, `+`));
  res:=NULL;
  for evalu in levalu do
    res:=res, op(ListTab(part, evalu));
  od;
  RETURN([res])
end;
```

Then, we give the Schurx function which computes the expression of a Schur functions on the alphabet $x_1, \ldots, x_{\mathtt{card}}$, in the basis of monomials:

```
#
# The Schur function s[part] on the alphabet x1, ..., x.card.
#
Schurx:=proc(part, card)
  local
    ltab, # list of tableaux...
    wrd,  # variable for loop...
    i,    # variable for sequence...
    sx,   # used for substitution...
    res;  # the result...
  ltab:=map(Tab2Word, SchurTab(part, card));
  sx:=seq(i=x.i, i=1..card);
  res:=0;
  for wrd in ltab do
    res:=res+convert(subs(sx, wrd), `*`)
  od;
  RETURN(res)
end;
```

Here is an example in which we compute the expansion of the Schur function $s_{3,2,2,1}$:

```
ACE> CLG_n(4):
ACE> Schurx([3,2,2,1], 4);

              2   2   3     2       2   3     2   2       3         2   3   2
        x4 x3  x2  x1  + x4  x3 x2  x1  + x4  x3  x2 x1  + x4 x3  x2  x1

           2      3   2         3   2   2       2   2   2   2     3       2   2
      + x4  x3 x2  x1  + x4 x3  x2  x1  + 3 x4  x3  x2  x1  + x4  x3 x2  x1

           2   3      2     3   2       2     2   2   3         2   3   2
      + x4  x3  x2 x1  + x4  x3  x2 x1  + x4  x3  x2  x1 + x4  x3  x2  x1
```

```
         3   2   2
    + x4  x3  x2  x1
ACE> x2m_n(");

                        m[3, 2, 2, 1] + 3 m[2, 2, 2, 2]

ACE> Tos_n(");

                                 s[3, 2, 2, 1]
```

4.6 Generating functions for Schubert polynomials ($\mathfrak{S}_n$ and B_n)

Schubert polynomials for the symmetric group can be expressed as certain coefficients in the algebra of divided differences [74] or nilCoxeter algebra [80]. The generating function of Fomin and Kirillov is:

$$\mathfrak{S}(\mathbf{x};\mathbf{y}) = C_1(x_1)\,C_2(x_2)\ldots C_{n-1}(x_{n-1}) \quad , \tag{4.1}$$

$$= \sum_{\mu\in\mathfrak{S}_n} \mathbb{X}_\mu(\mathbf{x},\mathbf{y})\ \partial_\mu \quad , \tag{4.2}$$

in which $C_i(x) = \Big(1 + (x - y_{n-i})\partial_{n-1}\Big)\Big(1 + (x - y_{n-i-1})\partial_{n-2}\Big)\ldots\Big(1 + (x - y_1)\partial_i\Big)$ and the x_i and y_i commute with the ∂_μ.

Here is a program computing this generating function. The function `C` computes a $C_i(x_i)$ by evaluating the product of all `(1 + (x - y_alphabet[j-i+1])*Transpo(j, n))` in the nilCoxeter algebra (`NcaMult`):

```
C:=proc(n, i, x, y_alphabet)
  local
    j, # variable for loop...
    r; # the result...
  r:=1;
  for j from i to n-1 do
    r := NcaMult((1 + (x - y_alphabet[j-i+1])*Transpo(j, n)), r)
  od;
  RETURN(r)
end;
```

The call `Transpo(j, n)` returns the elementary transposition exchanging j and $j+1$ in $\mathfrak{S}_n$. Then, the generating function of double Schubert polynomials corresponds to the following instructions:

```
An_Schubertfg:=proc(x_alphabet, y_alphabet)
  local
    i, # variable for loop...
    n, # degree of the symmetric group...
    r; # the result...
  n:=nops(x_alphabet);
  r:=1;
  for i from 1 to n-1 do
    r := NcaMult(r, C(n, i, x_alphabet[i], y_alphabet))
  od;
  RETURN(r)
end;
```

For instance, we have for $\mathfrak{S}_3$:

```
ACE> An_Schubertfg([x1, x2, x3], [y1, y2, y3]);

           (x1 - y1) [2, 1, 3] + [1, 2, 3] + (x2 - y1 + x1 - y2) [1, 3, 2]

         + (x1 - y2) (x1 - y1) [3, 1, 2] - (x1 - y1) (- x2 + y1) [2, 3, 1]

         - (x1 - y2) (x1 - y1) (- x2 + y1) [3, 2, 1]
```

B_n can be canonically embedded into $\mathfrak{S}_{2n}$. More explicitly, it amounts to identify the generators $u_1, \ldots, u_{n-1}$ of the B_n-nilCoxeter algebra with:

$$\begin{aligned} u_{n-1} &= \partial_1 \partial_{2n-1} \quad , \\ u_{n-2} &= \partial_2 \partial_{2n-2} \quad , \\ \vdots \quad &\vdots \qquad \vdots \\ u_1 &= \partial_{n-1} \partial_{n+1} \quad , \end{aligned}$$

and the extra generator u_0 to ∂_n.

The generating function of B_n-Schubert polynomials of Fomin and Kirillov is:

$$\begin{aligned} \mathrm{FK}(\mathbf{x}) &= B(x_1)\, B(x_2) \ldots B(x_n)\, C'_1(-x_1)\, C'_2(-x_2) \ldots C'_{n-1}(-x_{n-1}) && (4.3) \\ &= \sum_{\nu \in B_n} \mathfrak{X}_\nu(\mathbf{x})\ u_\nu && (4.4) \\ &= \sum \mathfrak{X}_\nu(\mathbf{x})\ \partial_\mu \quad , && (4.5) \end{aligned}$$

μ being the image of ν in $\mathfrak{S}_{2n}$, $B(x) = (1 + xu_{n-1}) \ldots (1 + xu_1)\,(1 + xu_0)\,(1 + xu_1) \ldots (1 + xu_{n-1})$ and $C'_i(x) = (1 + xu_{n-1})\,(1 + xu_{n-2}) \ldots (1 + xu_i)$. First of all, the `BnTranspoPalin` function realizes the embedding of B_n into $\mathfrak{S}_{2n}$, for a simple transposition. Now, the call `B(n, x)` stands for $B(x)$ whereas `Bn_C(n, i, x)` computes $C'_i(x)$:

```
B:=proc(n, x)
  local
    j, # variable for loop...
    r; # the result...
  r:=1;
  for j from n-1 by -1 to 0 do
    r := BnNcaMult(r, (1 + x*`BN/BnPalin2Code`(`BN/BnTranspoPalin`(j, n))))
  od;
  for j from 1 to n-1 do
    r := BnNcaMult(r, (1 + x*`BN/BnPalin2Code`(`BN/BnTranspoPalin`(j, n))))
  od;
  RETURN(r)
end;

Bn_C:=proc(n, i, x)
  local
    j, # variable for loop...
    r; # the result...
  r:=1;
  for j from i to n-1 do
    r := BnNcaMult((1 + x*`BN/BnPalin2Code`(`BN/BnTranspoPalin`(j, n))), r)
  od;
  RETURN(r)
end;

Bn_Schubertfg:=proc(x_alphabet)
  local
    i, # variable for loop...
    n, # B(n)...
    r; # the result...
  n:=nops(x_alphabet);
  r:=1;
  for i from 1 to n do
    r := BnNcaMult(r, B(n, x_alphabet[i]))
  od;
  for i from 1 to n-1 do
    r := BnNcaMult(r, Bn_C(n, i, -x_alphabet[i]))
  od;
  RETURN(r)
end;
```

For instance, we obtain for $n = 2$:

```
ACE> map(factor, Bn_Schubertfg([x1, x2]));

                            2
            (x1 + x2)  [0, 2] + (x1 + 2 x2) [0, 1] + (x1 + x2) [1, 0]
```

```
                                2
                  + x2  (x1 + x2) [0, 3] + x2 x1 (x1 + x2) [1, 2] + [0, 0]

                                2
                  + x2  x1 (x1 + x2) [1, 3] + x2 (x1 + x2) [1, 1]

ACE> BnaCode2Palin(");

                  2
          (x1 + x2)  [2, 4, 1, 3] + (x1 + 2 x2) [2, 1, 4, 3] + (x1 + x2) [1, 3, 2, 4]

                2
           + x2  (x1 + x2) [4, 2, 3, 1] + x2 x1 (x1 + x2) [3, 4, 1, 2] + [1, 2, 3, 4]

                2
           + x2  x1 (x1 + x2) [4, 3, 2, 1] + x2 (x1 + x2) [3, 1, 4, 2]
```

Their are other families of B_n-Schubert polynomials. Fomin and Kirillov have defined a second one and shown the link with a third one due to Biley and Haiman. Pragacz and Ratajski [77] describe still another one related to geometrical considerations, and a fifth one is due to Fulton.

5

Main routines

The reasonable man
adapts himself to the world;
the unreasonable one
persists in trying to adapt the world to himself.
Therefore all progress depends on the unreasonable man.

George Bernard Shaw.

This chapter describes the main routines of the Algebraic Combinatorics Environment. Each section is devoted to a package that defines an entire collection of functions dealing with a specific subject such as the symmetric group, symmetric functions, noncommutative symmetric functions, Schubert polynomials, hyperoctahedral groups, algebras related to hyperoctahedral groups, classical Lie groups, computations in the ring of polynomials in several variables, ...

Each function description has the following format: after a one-line description showing how the function can be called, we find informations about the expected arguments. Then, we give a brief exposition about the function, and finally we provide some examples of the use of the function. Some related functions or topics finish the description.

In each description, names in a `typewriter font` as for instance in the **Calling Sequence** section, are parameters which are to be used verbatim, while *italicized* names are prototypes that may be replaced by any *Maple* expression according to the type specification noted in the **Parameters** section. The optional parameters are so-called extra parameters and may therefore be omitted when the function is called.

The first section of this chapter is devoted to present the `ACE` package. It includes all other packages so that it is suffcient to load `ACE` to have access to any function of the "*environment*". Note that all *Maple* functions are still available after having loaded the `ACE` package.

Note also that the notion of package is a very powerful *Maple* concept but we will talk about an "environment" (in algebraic combinatorics) to refer to the `ACE` package since it clusters various packages dealing with specific objects, all related to algebraic combinatorics. Thus, it sounds a little bit more general than a simple package.

5.1 About the ACE package

✓

Calling Sequence:

`ACE[<function>]`$(args)$

`<function>`$(args)$

Synopsis:

- This package is an environment for Algebraic Combinatorics computation. It includes various packages providing functions to work with classical objects and more recent ones.
- The available packages are:

BN	BNA	CG	CLG
FM	FOCK	HEKA	IDCA
NCA	NCSF	SG	SGA
SHRI	SP	SYMF	TAB

- The two packages TYP and ERR are used to control the objects and to print errors.
- The TEX package provides functions to produce an output suitable with a TEX processor.
- Whenever there is a conflict between a function name in ACE and another name used in the same session, use the long form `ACE[<function>]`.
- The available functions are:

Banner	BuildData	LoadData	SysInfo
Time	UnloadData	WhichIs	

- For help with a particular function do either `?ACE[<function>]` or `?ACE,<function>` where `<function>` is one from the above list.

Examples:

```
> with(ACE):
> CTab2Mat(Word2CTab([4,5,4,1,6,3,5,3,4,3,4,3,2,6]));

                    [ 1  4  4  5  6  6 ]
                    [                  ]
                    [       3  3  4  5 ]
                    [                  ]
                    [             3  4 ]
                    [                  ]
                    [                3 ]
                    [                  ]
                    [                2 ]

> Time(TableXX, 't', 6):
```

```
> t;

                                2.750

> SysInfo();

number of CPU seconds used        : 5.65
total number of words requested   : 1000012
total number of words allocated   : 851929
number of garbage collection      : 1
```

5.1.1 Banner — print a banner about ACE

Calling Sequence:

Banner()

ACE[Banner]()

Parameters:

Synopsis:

- The Banner function prints a banner about ACE. It includes the release, the author's address and the list of all packages available with ACE.
- Whenever there is a conflict between the function name Banner and another name used in the same session, use the long form ACE['Banner'].

Examples:

```
> with(ACE):
> Banner();
################################################################
# Algebraic Combinatorics Environment             June 2, 1996 #
################################################################
# Sebastien Veigneau & Co               Institut Gaspard Monge #
# Universite de Marne-la-Vallee, 2 rue de la Butte Verte 93166 #
# Noisy-le-Grand CEDEX FRANCE.           Email: sv@univ-mlv.fr #
# URL: http://www-igm.univ-mlv.fr/~veigneau/HTML/ACE_PAGE.html #
################################################################
# ACE  Algebraic  Combinatorics  Environment #
#                                              #
# FM   Free Module over symmetric polynomials    #
# NCSF NonCommutative Symmetric  Functions     #
# CLG  Characters of Linear Groups           #
# SP   Schubert Polynomials                #
# SYMF SYMmetric Functions               #
# TAB  TABleaux                       #
#                                  #
# SHRI Hecke algebra Representations   #
# FOCK FOCK space representations    #    __/__/ __/_/ __/_/__/
#                                  #    __/ __/ __/   __/
```

```
# BNA  hyperoctahedral Algebras  #    __/_/__/ __/   __/_/
# SGA  Symmetric Group Algebra #    __/   __/ __/   __/
# NCA  NilCoxeter Algebra    #    __/    __/ __/_/ __/_/__/
# IDCA IDCoxeter Algebra   #
# HEKA HEcKe Algebra        #
#                           #
# BN   hyperoctahedral groups    #
# SG   Symmetric Groups        #
# CG   Classical Groups      #
#                          #
# ERR  ERRors            #
# TEX  TEX output      #
# TYP  TYPe checking #
######################
```

SEE ALSO:

5.1.2 BuildData — build and store data

Calling Sequence:

BuildData(*pack*, *data*)

ACE[BuildData](*pack*, *data*)

Parameters:

pack — any name of package

data — any sequence of names

Synopsis:

- The BuildData function is used to build and store data on the disk. The first argument *pack* specifies the name of the package or more precisely the name of the sub-directory that contains the data on the disk. The second argument is in fact a sequence of arguments that specifies the name of the data to be read on the disk.
- The *pack* name should be one of the list of package names _ACELPN.
- It returns true when it is able to store the computed data on the disk and false otherwise: in case of such a problem, the data which has been computed is usable but it (because it is not saved on the disk) should be recomputed each session.
- Refer to ACE['LoadData'] to see the list of data you can build.
- Whenever there is a conflict between the function name BuildData and another name used in the same session, use the long form ACE['BuildData'].

Examples:

```
> with(ACE):
> BuildData(FM, XmXfix, 4);

                         true
```

SEE ALSO: LoadData, UnloadData

5.1.3 LoadData — load data from disk

✘

Calling Sequence:

LoadData(*pack*, *data*)

ACE[LoadData](*pack*, *data*)

Parameters:

pack — any name of package

data — any sequence of names

Synopsis:

- The LoadData function is used to load data from the disk. The first argument *pack* specifies the name of the package or more precisely the name of the sub-directory that contains the data on the disk. The second argument is in fact a sequence of arguments that specifies the name of the data to be read on the disk.
- The *pack* name should be one of the list of package names _ACELPN.
- Whenever the corresponding file does not exist on the disk, then it runs the ACE['BuildData'] command to build the data and to store it on the disk.
- It returns true when the data is loaded and false otherwise, that is for instance when there is a problem to store the data built by BuildData on the disk: in case of such a problem, the data has been computed and is usable but it (because it is not saved on the disk) should be recomputed each session.
- It does not reload the data if it is already loaded: it then returns true.
- Here is the list of data you can load together with the corresponding command to load it:
  - Multiplication table between Schubert polynomials in a given symmetric group $\mathfrak{S}_n$. The result is expressed in the free module over symmetric polynomials in n variables (FM package). For instance, the command LoadData(FM, XmXfix, 4); is used to load the multiplication table for the symmetric group of degree 4: this table is then FM/data/XmXfix4.
  - Multiplication table between Schubert polynomials in a given symmetric group $\mathfrak{S}_n$ and all variables $x_i, i = 1, \ldots, n$. The result is expressed in the free module over symmetric polynomials in n variables (FM package). For instance, the command LoadData(FM, MXfix, 4); is used to load the multiplication table for the symmetric group of degree 4: this table is then FM/data/XMfix4.
- Whenever there is a conflict between the function name LoadData and another name used in the same session, use the long form ACE['LoadData'].

Examples:

```
> with(ACE):
> LoadData(FM, XmXfix, 4);

                              true
```

SEE ALSO: BuildData, UnloadData

5.1.4 SysInfo — give information on the system

Calling Sequence:

`SysInfo()`

`ACE[SysInfo]()`

Parameters:

Synopsis:

- The `SysInfo` function gives some information on the system. It may be useful to be aware of the memory or the CPU time you need to make some computation.
- Whenever there is a conflict between the function name `SysInfo` and another name used in the same session, use the long form `ACE['SysInfo']`.

Examples:

```
> with(ACE):
> Tos(s[3,2,1]^3):
> SysInfo();

number of CPU seconds used        : 74.833
total number of words requested   : 11001390
total number of words allocated   : 1507259
number of garbage collection      : 11
```

SEE ALSO:

5.1.5 Time — give time information

Calling Sequence:

`Time`$(f,'t',a_1,...,a_k)$

`ACE[Time]`$(f,'t',a_1,...,a_k)$

Parameters:

f	—	any name of function
t	—	any name of variable (formal parameter)
$a_1,...,a_k$	—	optional extra arguments

Synopsis:

- The `Time` function runs $f(a_1,\ldots,a_k)$ and saves in the parameter $'t'$ the computation time.
- The value of `Time`$(f,'t',a\_1,...,a\_k)$ is the value of $f(a_1,\ldots,a_k)$.
- For composite functions, transform it into a single function in order to use `Time`.
- Whenever there is a conflict between the function name `Time` and another name used in the same session, use the long form `ACE['Time']`.

Examples:

```
> with(ACE):
> Time(Tos, 'time', s[2,2,1]^2);

 s[3, 3, 1, 1, 1, 1] + s[3, 3, 2, 2] + 2 s[3, 3, 2, 1, 1] + s[3, 3, 3, 1]

    + s[3, 2, 2, 1, 1, 1] + 2 s[3, 2, 2, 2, 1] + s[2, 2, 2, 2, 2]

    + s[2, 2, 2, 2, 1, 1] + s[4, 4, 2] + s[4, 3, 3] + 2 s[4, 3, 2, 1]

    + s[4, 2, 2, 2] + s[4, 3, 1, 1, 1] + s[4, 2, 2, 1, 1] + s[4, 4, 1, 1]

> time;

                                       .316

> scal:=proc(part1, part2, part3, part4)
>   ScalarSf( MuirSf( m[op(part1)], s[op(part2)] ),
>             MuirSf( m[op(part3)], s[op(part4)] ))
> end:
> Time(scal, 'time', [2,1], [3,1,1], [2], [4,1,1]);

                                       3

> time;

                                     1.900
```

SEE ALSO:

5.1.6 `UnloadData` — unload data

Calling Sequence:

`UnloadData`$(pack, data)$

`ACE[UnloadData]`$(pack, data)$

Parameters:

pack — any name of package

data — any sequence of names

Synopsis:

- The `UnloadData` function is used to unload *data* that is to forget the corresponding data, during the current session. Refer to the function `LoadData` to see the list of possible data to unload.
- The function returns true if the data was loaded before you run the `UnloadData` command and false otherwise.
- Note that the file (if it exists) that stores the corresponding data on disk is not removed.
- Whenever there is a conflict between the function name `UnloadData` and another name used in the same session, use the long form `ACE['UnloadData']`.

Examples:

```
> with(ACE):
> UnloadData(FM, XmXfix, 4);
```

false

SEE ALSO: BuildData, LoadData

5.1.7 WhichIs — look for a function in a list of packages

✘

Calling Sequence:

WhichIs(f)

WhichIs(f,$'r'$)

ACE[WhichIs](f)

ACE[WhichIs](f,$'r'$)

Parameters:

f — any name of function

r — any name of variable (formal parameter)

Synopsis:

- The WhichIs function looks for a function whose name is f. It returns true in the case it finds such a function and false otherwise. It looks for the name f among all functions of all packages contained in the list _ACELPN.
- When called with a second formal parameter, say $'r'$, r contains the long form name of the first function whose name matches the name f (this is the case when WhichIs returns true). When WhichIs returns false, then r contains the list of functions whose names contain the substring f.
- Whenever there is a conflict between the function name WhichIs and another name used in the same session, use the long form ACE['WhichIs'].

Examples:

```
> with(ACE):
> WhichIs(NewtonInterp, 'n');
```

true

```
> n;
```

SP[NewtonInterp]

```
> WhichIs(On, 'l');
```

false

```
> l;

   [TAB[PermOnWord], HEKA[HekaOnPol], HEKA[HekaOnX], SGA[SgaOnX],

    SGA[SgaOnPol], NCA[NcaOnY], NCA[NcaOnXX], NCA[NcaOnPol],

    NCA[NcaOnX], NCA[NcaOnYY], IDCA[IdcaOnX], IDCA[IdcaOnPol]]
```

SEE ALSO:

5.2 SG, a package for the symmetric group

✓

Calling Sequence:

SG[<function>](*args*)

<function>(*args*)

Synopsis:

- This package provides functions to work with the symmetric group.
- Do either ?TYP,Permutation or ?TYP[Permutation] to get more information on permutations.
- Whenever there is a conflict between a function name in SG and another name used in the same session, use the long form SG[<function>].
- The available functions are:

Betti	Bruhat	CCCard	Code2Perm
CompPerms	Cycle2Perm	DimRep	FactorPerm
FirstPerm	Interval	InvPerm	Inversions
IsDominant	IsGrassmannian	IsVexillar	LastPerm
ListDominant	ListGrassmannian	ListPerm	ListVexillar
Mat2Perm	MultPerm	NbCC	NbDominant
NbGrassmannian	NbVexillar	NextPerm	Part2CC
Perm2Code	Perm2Cycle	Perm2CycleType	Perm2Desc
Perm2Length	Perm2ListRd	Perm2Mat	Perm2NbRd
Perm2Part	Perm2RRep	Perm2Rd	Perm2Rise
Perm2Rothe	Perm2Sign	PrevPerm	RRep2Perm
RandDominant	RandGrassmannian	RandPerm	Rd2Perm
Rothe2Perm	SnCharTable	Transpo	Vect2Perm

- For help with a particular function do either ?SG[<function>] or ?SG,<function> where <function> is one from the above list.

Examples:

```
> with(SG):
> Perm2Rothe([5,6,2,4,1,3]);
```

```
[ 1  2  3  4  x  . ]
[                  ]
[ 2  3  4  5  .  x ]
[                  ]
[ 3  x  .  .  .  . ]
[                  ]
[ 4  .  5  x  .  . ]
[                  ]
[ x  .  .  .  .  . ]
[                  ]
[ .  .  x  .  .  . ]
```

5.2.1 `MultPerm` — multiply permutations

Calling Sequence:

`MultPerm`$(perm_1, perm_2)$

`SG[MultPerm]`$(perm_1, perm_2)$

Parameters:

$perm_1, perm_2$ — any lists denoting permutations

Synopsis:

- The `MultPerm` function calculates the product of a sequence (of arbitrary length) of permutations.
- `MultPerm`$(perm_1, perm_2)$ computes $perm_1(perm_2)$.
- Permutations need not to belong to the same symmetric group: $\mathfrak{S}_n$ is embedded into $\mathfrak{S}_{n+1}$ by adding the fixed point $n+1$.
- Whenever there is a conflict between the function name `MultPerm` and another name used in the same session, use the long form `SG['MultPerm']`.

Examples:

```
> with(SG):
> MultPerm([1,3,2], [2,3,1]);
```

```
[3, 2, 1]
```

```
> MultPerm([1,3,2], [2,3,1], [1,3,2,4]);
```

```
[3, 1, 2, 4]
```

SEE ALSO: `TYP[Permutation]`

5.2.2 Betti — compute the Betti polynomial

Calling Sequence:

Betti($p,'q'$)

SG[Betti]($p,'q'$)

Parameters:

p — any list denoting any permutation

Synopsis:

- The Betti function computes the generating function of the lengths of all permutations that are smaller than a given permutation p, according to Bruhat order.
- The second parameter $'q'$ can be arbitrarily choosen.
- Whenever there is a conflict between the function name Betti and another name used in the same session, use the long form SG['Betti'].

Examples:

```
> with(SG):
> Betti([3,4,1,2], 'q');

                         4      3      2
                        q  + 4 q  + 5 q  + 3 q + 1
```

SEE ALSO: Bruhat, CompPerms, Interval

5.2.3 Bruhat — compare two permutations according to the Bruhat order

Calling Sequence:

Bruhat($perm_1, perm_2$)

SG[Bruhat]($perm_1, perm_2$)

Parameters:

$perm_1, perm_2$ — any lists denoting permutations

Synopsis:

- The Bruhat function compares two permutations. It returns true if $perm_1 \geq perm_2$ and false otherwise.
- Whenever there is a conflict between the function name Bruhat and another name used in the same session, use the long form SG['Bruhat'].

Examples:

```
> with(SG):
> Bruhat([4,1,2,3], [2,3,1,4]);

                                   false
```

```
> Bruhat([3,1,4,2], [3,1,2,4]);
```

true

SEE ALSO: CompPerms

5.2.4 CCCard — compute the cardinal of a conjugacy class

Calling Sequence:

CCCard(*part*)

SG[CCCard](*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The CCCard function gives the cardinal of a conjugacy class.
- The argument must be a partition, that is a weakly decreasing list of positive integers.
- Whenever there is a conflict between the function name CCCard and another name used in the same session, use the long form SG['CCCard'].

Examples:

```
> with(SG):
> CCCard([2,1,1]);
```

6

SEE ALSO: CycleType, Zee

5.2.5 SnCharTable — compute the table of characters

✓

Calling Sequence:

SnCharTable(n)

SnCharTable($n,'index'$)

SnCharTable($n, part_1, part_2$)

SG[SnCharTable](n)

SG[SnCharTable]($n,'index'$)

SG[SnCharTable]($n, part_1, part_2$)

Parameters:

n — any positive integer denoting the degree of a symmetric group

$part_1, part_2$ — any lists denoting partitions

Synopsis:

- The `SnCharTable` gives the table of characters for the symmetric group of order n. Let t be this matrix, $part_1$, $part_2$ be two partitions of n, of index i, j in the list of partitions `SnCharTable`$(n,'index')$, then $t[i,j]$ is equal to the value of the irreducible character of index $part_1$ on the conjugacy class indexed by $part_2$.
- One may extract some rows or columns by the `linalg[row]` and `linalg[col]` commands of the `linalg` package.
- Whenever there is a conflict between the function name `SnCharTable` and another name used in the same session, use the long form `SG['SnCharTable']`.

Examples:

```
> with(SG):
> SnCharTable(3);

                              [  1   1  1 ]
                              [           ]
                              [ -1   0  2 ]
                              [           ]
                              [  1  -1  1 ]

> linalg[row](", 2);

                               [ -1, 0, 2 ]

> SnCharTable(3, 'index');

                          [[3], [2, 1], [1, 1, 1]]

> SnCharTable(3, [2,1], [1,1,1]);

                                    2
```

SEE ALSO: `TYP[IsPart]`, `linalg[col]`, `linalg[row]`

5.2.6 `Code2Perm` — calculate a permutation from its code

Calling Sequence:

`Code2Perm`$(code)$

`Code2Perm`$(code, n)$

`SG[Code2Perm]`$(code)$

`SG[Code2Perm]`$(code, n)$

Parameters:

code	—	a list denoting the code of a permutation
n	—	the degree of a symmetric group

Synopsis:

- The Code2Perm function calculates a permutation from its Lehmer code.
- Let $[c_1, c_2, \ldots]$ be the code of *perm*, then c_i is equal to the number of indices $j > i$ such that $perm[j] < perm[i]$.
- The argument n specifies the degree of the symmetric group.
- Whenever there is a conflict between the function name Code2Perm and another name used in the same session, use the long form SG['Code2Perm'].

Examples:

```
> with(SG):
> Code2Perm([2,4,0,1]);

                         [3, 6, 1, 4, 2, 5]

> Code2Perm([2,4,0,1], 8);

                       [3, 6, 1, 4, 2, 5, 7, 8]
```

SEE ALSO: Perm2Code

5.2.7 Perm2CycleType — compute the cycle type of a permutation

Calling Sequence:

Perm2CycleType(*perm*)

SG[Perm2CycleType](*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The Perm2CycleType function calculates the cycle type of the permutation *perm*.
- A permutation can be expressed as a product of disjoint cycles. The corresponding partition is obtained by listing in weakly decreasing order the lengths of these cycles.
- Whenever there is a conflict between the function name Perm2CycleType and another name used in the same session, use the long form SG['Perm2CycleType'].

Examples:

```
> with(SG):
> p:=[7,1,3,6,5,4,2]:
> Perm2CycleType(p);

                            [3, 2, 1, 1]

> Perm2Cycle(p);

                         [[1, 7, 2], [4, 6]]
```

SEE ALSO: CCCard, Cycle2Perm, Perm2Cycle

5.2.8 Cycle2Perm — calculate the permutation from its decomposition into disjoint cycles

Calling Sequence:

`Cycle2Perm`(dc)

`SG[Cycle2Perm]`(dc)

Parameters:

dc — any list of list(s)

Synopsis:

- The `Cycle2Perm` function calculates the permutation corresponding to its decomposition dc into disjoint cycles. Note however that the main format for permutations is the list notation.
- Fixed points need not to be given in the input.
- Whenever there is a conflict between the function name `Cycle2Perm` and another name used in the same session, use the long form `SG['Cycle2Perm']`.

Examples:

```
> with(SG):
> Cycle2Perm([[1, 7, 2], [4, 6], [3], [5]]);

                              [7, 1, 3, 6, 5, 4, 2]

> Cycle2Perm([[1, 7, 2], [4, 6]]);

                              [7, 1, 3, 6, 5, 4, 2]
```

See Also: `CCCard, NbCC, Perm2Cycle, Perm2CycleType, Vect2Perm`

5.2.9 DimRep — compute the dimension of a representation

✘

Calling Sequence:

`DimRep`($part$)

`DimRep`($part, part_{in}$)

`SG[DimRep]`($part$)

`SG[DimRep]`($part, part_{in}$)

Parameters:

$part, part_{in}$ — any lists denoting partitions

Synopsis:

- The `DimRep` function computes the dimension of the irreducible representation of index $part$, for the symmetric group of order the weight of $part$.
- When called with a second argument $part_{in}$, it computes the dimension of the representation indexed by the skew partition $part/part_{in}$.

- Adding the parameter `'q'`, it computes the specialization of s_{part} or $s_{part/part_{in}}$ in $x_1 = 1, x_2 = q, \dots$
- Whenever there is a conflict between the function name `DimRep` and another name used in the same session, use the long form `SG['DimRep']`.

Examples:

```
> with(SG):
> DimRep([4,4,3,2,1,1,1]);

                                   512512

> DimRep([4,2,1,1], [2,1]);

                                     30

> DimRep([4,2,1,1], [2,1], 'q');

                 2            2         4    3    2
               (q  + q + 1) (q  + 1) (q  + q  + q  + q + 1) q
```

SEE ALSO: `SnCharTable`

5.2.10 `FactorPerm` — test whether a permutation factorizes into a direct product

Calling Sequence:

`FactorPerm`(*perm*,'*l*')

`SG[FactorPerm]`(*perm*,'*l*')

Parameters:

perm — any list denoting a permutation

Synopsis:

- Given a permutation *perm* in $\mathfrak{S}_n$, the `FactorPerm` function tests whether *perm* belongs to a Young sub-group $\mathfrak{S}_i \times \mathfrak{S}_j$. If it factorizes, the second formal parameter $'l'$ is set to the minimal length of a left block.
- Whenever there is a conflict between the function name `FactorPerm` and another name used in the same session, use the long form `SG['FactorPerm']`.

Examples:

```
> with(SG):
> FactorPerm([3,1,2,5,4,7,6], 'l');

                                     true
> l;

                                      3
```

SEE ALSO:

5.2.11 FirstPerm — compute the first permutation $[1, 2, \ldots, n]$

Calling Sequence:

FirstPerm(n)

SG[FirstPerm](n)

Parameters:

n — any positive number denoting the degree of a symmetric group

Synopsis:

- The FirstPerm function calculates the first permutation according to the lexicographic order.
- Whenever there is a conflict between the function name FirstPerm and another name used in the same session, use the long form SG['FirstPerm'].

Examples:

```
> with(SG):
> FirstPerm(6);

                    [1, 2, 3, 4, 5, 6]
```

SEE ALSO: LastPerm, NextPerm, PrevPerm

5.2.12 Interval — compute the list of codes of all permutations below a given one

Calling Sequence:

Interval(*perm*)

SG[Interval](*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The Interval function computes the list of the Lehmer codes of all permutations that are smaller than *perm*, according to the Bruhat order.
- Whenever there is a conflict between the function name Interval and another name used in the same session, use the long form SG['Interval'].

Examples:

```
> with(SG):
> Interval([3,1,2]);

            [[2, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 0]]

> map(Code2Perm, Interval([3,2,1]));

    [[3, 2, 1], [3, 1, 2], [2, 3, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3]]
```

SEE ALSO: Betti, Bruhat, CompPerms, IDCA[Demazure]

5.2.13 `Inversions` — calculate the set of all inversions of a permutation

Calling Sequence:

`Inversions`(*perm*)

`SG[Inversions]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Inversions` function calculates the list of all inversions of a permutation. The result is in lexicographic ordering.
- The set of inversions is a set of pairs of positions (i, j) such that $i < j$ and $perm[j] < perm[i]$.
- Whenever there is a conflict between the function name `Inversions` and another name used in the same session, use the long form `SG['Inversions']`.

Examples:

```
> with(SG):
> Inversions([4,1,3,2]);

                    [{1, 2}, {1, 3}, {1, 4}, {3, 4}]
```

See Also: `Perm2Length`

5.2.14 `InvPerm` — invert a permutation

Calling Sequence:

`InvPerm`(*perm*)

`SG[InvPerm]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `InvPerm` function inverts a permutation *perm*.
- Whenever there is a conflict between the function name `InvPerm` and another name used in the same session, use the long form `SG['InvPerm']`.

Examples:

```
> with(SG):
> InvPerm([3,1,2,4]);

                              [2, 3, 1, 4]
```

See Also:

5.2.15 `IsDominant` — test whether a permutation is dominant

Calling Sequence:

`IsDominant`(*perm*)

`SG[IsDominant]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `IsDominant` function tests whether a permutation is a dominant one or not.
- A permutation is dominant if and only if its code is a partition (*i.e.* a weakly decreasing vector of non-negative integers).
- Whenever there is a conflict between the function name `IsDominant` and another name used in the same session, use the long form `SG['IsDominant']`.

Examples:

```
> with(SG):
> IsDominant([1,4,2,3]);
                                    false
> IsDominant([5,3,4,1,2]);
                                    true
> Perm2Code([5,3,4,1,2]);
                                [4, 2, 2, 0, 0]
```

SEE ALSO: `IsVexillar, IsGrassmannian`

5.2.16 `IsGrassmannian` — test whether a permutation is Grassmannian

Calling Sequence:

`IsGrassmannian`(*perm*)

`SG[IsGrassmannian]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `IsGrassmannian` function tests whether a permutation is a Grassmannian one or not.
- A permutation is Grassmannian if and only if it has only one descent, *i.e.* there exists only one i such that $perm[i] > perm[i+1]$.
- Whenever there is a conflict between the function name `IsGrassmannian` and another name used in the same session, use the long form `SG['IsGrassmannian']`.

Examples:

```
> with(SG):
> IsGrassmannian([4,1,5,2,3]);

                              false

> IsGrassmannian([1,3,4,5,2,7,8]);

                              true
```

SEE ALSO: `IsVexillar`, `IsDominant`

5.2.17 `IsVexillar` — test whether a permutation is vexillary

Calling Sequence:

`IsVexillar`(*perm*)

`SG[IsVexillar]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `IsVexillar` function tests whether a permutation is a vexillary one or not.
- Given *perm*, let *part*(*perm*) be the partition obtained by reordering the components of the code of *perm*. Then *perm* is vexillary if and only if $part(perm^{-1})$ is the conjugate of *part*(*perm*).
- Whenever there is a conflict between the function name `IsVexillar` and another name used in the same session, use the long form `SG['IsVexillar']`.

Examples:

```
> with(SG):
> IsVexillar([2,1,4,3]);

                              false

> perm:=[5,3,4,1,2]:
> IsVexillar(perm);

                              true
> Perm2Part( InvPerm(perm) );

                           [3, 3, 1, 1]
> Perm2Part(perm);

                            [4, 2, 2]
```

SEE ALSO: `IsDominant`, `IsGrassmannian`, `Perm2Part`

5.2.18 LastPerm — compute the last permutation $[n, n-1, \ldots, 1]$

Calling Sequence:

LastPerm(n)

SG[LastPerm](n)

Parameters:

n — any positive number denoting the degree of a symmetric group

Synopsis:

- The LastPerm function calculates the last permutation according to the lexicographic order.
- Whenever there is a conflict between the function name LastPerm and another name used in the same session, use the long form SG['LastPerm'].

Examples:

```
> with(SG):
> LastPerm(6);

                              [6, 5, 4, 3, 2, 1]
```

SEE ALSO: FirstPerm, NextPerm, PrevPerm

5.2.19 Perm2Length — calculate the length of a permutation

Calling Sequence:

Perm2Length(*perm*)

SG[Perm2Length](*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The Perm2Length function calculates the length of the permutation *perm*, that is its number of inversions. This is also the minimal length of an expression of the permutation as a product of simple transpositions, as well as the sum of the components of its code.
- Whenever there is a conflict between the function name Perm2Length and another name used in the same session, use the long form SG['Perm2Length'].

Examples:

```
> with(SG):
> perm:=[5,3,2,1,4]:
> Perm2Length(perm);

                                    7

> Perm2Rd(perm);
```

```
                    [4, 3, 2, 1, 3, 2, 3]
> Perm2Code(perm);
                    [4, 2, 1, 0, 0]
```

SEE ALSO: `Perm2Code, Perm2Rd`

5.2.20 `ListDominant` — list all dominant permutations

Calling Sequence:

`ListDominant`$(n,'code')$

`SG[ListDominant]`$(n,'code')$

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `ListDominant` function returns the list of all dominant permutations belonging to the symmetric group of degree n. Adding the argument $'code'$, it returns a list of codes instead (it is more efficient).
- Whenever there is a conflict between the function name `ListDominant` and another name used in the same session, use the long form `SG['ListDominant']`.

Examples:

```
> with(SG):
> ListDominant(3);

        [[3, 2, 1], [2, 3, 1], [3, 1, 2], [2, 1, 3], [1, 2, 3]]

> ListDominant(3, code);

        [[2, 1, 0], [1, 1, 0], [2, 0, 0], [1, 0, 0], [0, 0, 0]]
```

SEE ALSO: `ListGrassmannian, NbDominant, NbVexillar`

5.2.21 `ListGrassmannian` — list all Grassmannian permutations

Calling Sequence:

`ListGrassmannian`$(n,'code')$

`SG[ListGrassmannian]`$(n,'code')$

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `ListGrassmannian` function returns the list of codes of all Grassmannian permutations belonging to the symmetric group of degree n.

- Whenever there is a conflict between the function name `ListGrassmannian` and another name used in the same session, use the long form `SG['ListGrassmannian']`.

Examples:

```
> with(SG):
> ListGrassmannian(3);

          [[1, 2, 3], [2, 1, 3], [3, 1, 2], [1, 3, 2], [2, 3, 1]]

> ListGrassmannian(4, 'code');

   [[0, 0, 0, 0], [1, 0, 0, 0], [2, 0, 0, 0], [3, 0, 0, 0], [0, 1, 0, 0],

    [1, 1, 0, 0], [0, 2, 0, 0], [1, 2, 0, 0], [2, 2, 0, 0], [0, 0, 1, 0],

    [0, 1, 1, 0], [1, 1, 1, 0]]
```

SEE ALSO: `ListDominant`, `NbGrassmannian`, `NbVexillar`

5.2.22 `Perm2ListRd` — list all reduced decompositions of a given permutation

Calling Sequence:

`Perm2ListRd`(*perm*)

`SG[Perm2ListRd]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2ListRd` function returns the list of all the reduced decompositions of a given permutation *i.e.* all expressions of *perm* as a product of simple transpositions (of minimal length).

- Whenever there is a conflict between the function name `Perm2ListRd` and another name used in the same session, use the long form `SG['Perm2ListRd']`.

Examples:

```
> with(SG):
> Perm2ListRd([3,1,5,2,4]);

   [[4, 2, 3, 1], [2, 4, 3, 1], [4, 2, 1, 3], [2, 4, 1, 3], [2, 1, 4, 3]]
```

SEE ALSO: `Perm2NbRd`, `Perm2Rd`, `Rd2Perm`

5.2.23 `ListVexillar` — list all vexillar permutations

✘

Calling Sequence:

`ListVexillar`$(n,'code')$

`SG[ListVexillar]`$(n,'code')$

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `ListVexillar` function returns the list of codes of all vexillar permutations belonging to the symmetric group of degree n (using an algorithm due to J. West).
- Whenever there is a conflict between the function name `ListVexillar` and another name used in the same session, use the long form `SG['ListVexillar']`.

Examples:

```
> with(SG):
> ListVexillar(4):
> nops(");

                                    23

> ListVexillar(4, 'code');

  [[3, 2, 1, 0], [2, 2, 1, 0], [2, 1, 1, 0], [2, 1, 0, 0], [3, 1, 1, 0],

   [1, 2, 1, 0], [1, 1, 1, 0], [1, 1, 0, 0], [3, 1, 0, 0], [1, 2, 0, 0],

   [1, 0, 0, 0], [3, 2, 0, 0], [2, 2, 0, 0], [2, 0, 1, 0], [2, 0, 0, 0],

   [3, 0, 1, 0], [0, 2, 1, 0], [0, 1, 1, 0], [0, 1, 0, 0], [3, 0, 0, 0],

   [0, 2, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]
```

SEE ALSO: `ListDominant`, `ListGrassmannian`

5.2.24 `Mat2Perm` — calculate a permutation from its matrix of permutation

Calling Sequence:

`Mat2Perm`(sqm)

`SG[Mat2Perm]`(sqm)

Parameters:

sqm — any square permutation matrix

Synopsis:

- The `Mat2Perm` function calculates a permutation from a matrix. Given a permutation $perm$, the corresponding matrix has entries 1 in position $(i, perm[i])$ and 0 otherwise.

- Whenever there is a conflict between the function name `Mat2Perm` and another name used in the same session, use the long form `SG['Mat2Perm']`.

Examples:

```
> with(SG):
> Perm2Mat([2,4,3,1]);

                    [ 0  1  0  0 ]
                    [            ]
                    [ 0  0  0  1 ]
                    [            ]
                    [ 0  0  1  0 ]
                    [            ]
                    [ 1  0  0  0 ]

> Mat2Perm(");

                    [2, 4, 3, 1]
```

SEE ALSO: `Perm2Mat`, `Perm2RRep`, `RRep2Perm`

5.2.25 `NbCC` — compute the number of conjugacy classes

Calling Sequence:

`NbCC`(n)

`SG[NbCC]`(n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `NbCC` function calculates the number of conjugacy classes for the symmetric group of degree n. Since conjugacy classes exactly correspond to partitions, `NbCC`(n) is also the number of partitions of n.
- Whenever there is a conflict between the function name `NbCC` and another name used in the same session, use the long form `SG['NbCC']`.

Examples:

```
> with(SG):
> NbCC(20);

                    627
```

SEE ALSO: `CCCard`

5.2.26 NbDomimant — compute the number of dominant permutations

Calling Sequence:

NbDominant(n)

SG[NbDominant](n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The NbDominant function calculates the number of dominant permutations belonging to the symmetric group of degree n.
- Whenever there is a conflict between the function name NbDominant and another name used in the same session, use the long form SG['NbDominant'].

Examples:

```
> with(SG):
> NbDominant(10);
```

```
16796
```

SEE ALSO: NbGrassmannian, NbVexillar

5.2.27 NbGrassmannian — compute the number of Grassmannian permutations

Calling Sequence:

NbGrassmannian(n)

SG[NbGrassmannian](n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The NbGrassmannian function calculates the number of Grassmannian permutations belonging to the symmetric group of degree n.
- Whenever there is a conflict between the function name NbGrassmannian and another name used in the same session, use the long form SG['NbGrassmannian'].

Examples:

```
> with(SG):
> NbGrassmannian(10);
```

```
1014
```

SEE ALSO: NbDominant, NbVexillar

5.2.28 Perm2NbRd — compute the number of reduced decompositions

Calling Sequence:

Perm2NbRd(*perm*)

SG[Perm2NbRd](*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The Perm2NbRd function computes the number of reduced decompositions of a given permutation.
- Whenever there is a conflict between the function name Perm2NbRd and another name used in the same session, use the long form SG['Perm2NbRd'].

Examples:

```
> with(SG):
> Perm2NbRd([6,5,4,3,2,1]);

                                292864

> Perm2NbRd([2,5,1,4,3]);
                                  11
```

SEE ALSO: Perm2ListRd

5.2.29 NbVexillar — compute the number of vexillary permutations

Calling Sequence:

NbVexillar(n)

SG[NbVexillar](n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The NbVexillar function calculates the number of vexillary permutations belonging to the symmetric group of degree n.
- Whenever there is a conflict between the function name NbVexillar and another name used in the same session, use the long form SG['NbVexillar'].

Examples:

```
> with(SG):
> NbVexillar(10);

                                586590
```

SEE ALSO: NbDominant, NbGrassmannian

5.2.30 `NextPerm` — compute the next permutation in lexicographic ordering

Calling Sequence:

`NextPerm`($perm$)

`SG[NextPerm]`($perm$)

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The `NextPerm` function calculates the next permutation according to the lexicographic order. The function returns the `NULL` value when the argument is the last permutation $[n, n-1, ..., 2, 1]$.
- Whenever there is a conflict between the function name `NextPerm` and another name used in the same session, use the long form `SG['NextPerm']`.

Examples:

```
> with(SG):
> NextPerm([2,1,3]);
                                    [2, 3, 1]
```

SEE ALSO: `FirstPerm`, `TYP[IsPerm]`, `LastPerm`, `PrevPerm`

5.2.31 `CompPerms` — compare two permutations

Calling Sequence:

`CompPerms`($perm_1, perm_2, kind$)

`SG[CompPerms]`($perm_1, perm_2, kind$)

Parameters:

$perm_1, perm_2$ — any lists denoting permutations

$kind$ — `'lexic'`, `'cixel'`

Synopsis:

- The `CompPerms` function compares two permutations. The ordering is set by the argument kind, and the function returns true if $perm_1 \geq perm_2$ according to the choosen ordering.
- $kind$=`'lexic'` for lexicographic ordering.
- $kind$=`'cixel'` for inverse lexicographic ordering, that is lexicographic ordering but from right to left.
- Whenever there is a conflict between the function name `CompPerms` and another name used in the same session, use the long form `SG['CompPerms']`.

Examples:

```
> with(SG):
> CompPerms([3,1,2], [1,3,2], 'lexic');
```

true

```
> CompPerms([3,1,2], [1,3,2], 'cixel');
```

false

SEE ALSO: Bruhat

5.2.32 Part2CC — compute a conjugacy class

Calling Sequence:

Part2CC(*part*)

SG[Part2CC](*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The Part2CC function computes the list of all permutations whose cycle type is the given partition *part*.
- Whenever there is a conflict between the function name Part2CC and another name used in the same session, use the long form SG['Part2CC'].

Examples:

```
> with(SG):
> Part2CC([2,1]);
```

[[1, 3, 2], [2, 1, 3], [3, 2, 1]]

SEE ALSO: CCCard, NbCC

5.2.33 Perm2Code — calculate the code of a permutation

Calling Sequence:

Perm2Code(*perm*)

SG[Perm2Code](*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The Perm2Code function calculates the Lehmer code of a permutation.

- Let $[c_1, c_2, ...]$ be the code of $perm$, then c_i is equal to the number of indices $j > i$ such that $perm[j] < perm[i]$.
- Whenever there is a conflict between the function name `Perm2Code` and another name used in the same session, use the long form `SG['Perm2Code']`.

Examples:

```
> with(SG):
> Perm2Code([4,1,5,2,3]);
                              [3, 0, 2, 0, 0]

> Code2Perm(");
                              [4, 1, 5, 2, 3]
```

SEE ALSO: `Code2Perm`

5.2.34 `Perm2Cycle` — decompose a permutation into disjoint cycles

Calling Sequence:

`Perm2Cycle`($perm$)

`SG[Perm2Cycle]`($perm$)

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The `Perm2Cycle` function calculates the decomposition of a permutation into disjoint cycles. It does not print the fixed points (cycles of length 1).
- Whenever there is a conflict between the function name `Perm2Cycle` and another name used in the same session, use the long form `SG['Perm2Cycle']`.

Examples:

```
> with(SG):
> Perm2Cycle([5,4,3,2,1]);
                              [[1, 5], [2, 4]]
```

SEE ALSO: `CCCard, Cycle2Perm, NbCC, Perm2CycleType, Vect2Perm`

5.2.35 `Perm2Desc` — compute the list of descents of a permutation

Calling Sequence:

`Perm2Desc`(*perm*)

`SG[Perm2Desc]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2Desc` function computes the list of descents of the permutation *perm*, *i.e.* the list of positions i, such that $perm[i] > perm[i+1]$.
- When called with the second argument $'nb'$, it returns the number of descents of the permutation *perm*.
- When called with the second argument $'maj'$, it returns the sum of the descent positions.
- When called with the second argument $'comaj'$, it returns the sum of all values (n - descent position), where n is equal to `nops`(*perm*).
- Whenever there is a conflict between the function name `Perm2Desc` and and another name used in the same session, use the long form `SG['Perm2Desc']`.

Examples:

```
> with(SG):
> perm:=[3,1,4,2,6,5,7]:
> Perm2Desc(perm);

                                  [1, 3, 5]

> Perm2Desc(perm, 'nb');

                                      3

> Perm2Desc(perm, 'maj');

                                      9

> Perm2Desc(perm, 'comaj');

                                      12
```

SEE ALSO: `Perm2Rise`

5.2.36 `Perm2Mat` — calculate the permutation matrix

Calling Sequence:

`Perm2Mat`(*perm*)

`SG[Perm2Mat]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2Mat` function calculates the permutation matrix corresponding to *perm*. This matrix has entries 1 in position $(i, perm[i])$ and 0 otherwise.
- Whenever there is a conflict between the function name `Perm2Mat` and another name used in the same session, use the long form `SG['Perm2Mat']`.

Examples:

```
> with(SG):
> Perm2Mat([2,4,3,1]);
```

```
[ 0  1  0  0 ]
[            ]
[ 0  0  0  1 ]
[            ]
[ 0  0  1  0 ]
[            ]
[ 1  0  0  0 ]
```

SEE ALSO: `Mat2Perm`, `Perm2RRep`, `RRep2Perm`

5.2.37 `Perm2RRep` — calculate the matrix representing a permutation in the regular representation

Calling Sequence:

`Perm2RRep`(*perm*)

`SG[Perm2RRep]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2RRep` function calculates the matrix representing *perm* in the regular representation. It amounts to consider *perm* as the linear operator acting on permutations by left multiplication.
- Whenever there is a conflict between the function name `Perm2RRep` and another name used in the same session, use the long form `SG['Perm2RRep']`.

Examples:

```
> with(SG):
> Perm2RRep([2,3,1]);
```

```
[ 0  0  0  0  1  0 ]
[                  ]
[ 0  0  0  0  0  1 ]
```

```
                 [                 ]
                 [ 0  1  0  0  0  0 ]
                 [                 ]
                 [ 1  0  0  0  0  0 ]
                 [                 ]
                 [ 0  0  0  1  0  0 ]
                 [                 ]
                 [ 0  0  1  0  0  0 ]
```

```
> RRep2Perm(");

                              [2, 3, 1]
```

SEE ALSO: `Mat2Perm`, `Perm2Mat`, `RRep2Perm`

5.2.38 `Perm2Rd` — calculate a canonical reduced decomposition of a permutation

Calling Sequence:

`Perm2Rd`(*perm*)

`SG[Perm2Rd]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2Rd` function calculates a canonical reduced decomposition of *perm*. It is obtained by reading the Rothe diagram.
- Whenever there is a conflict between the function name `Perm2Rd` and another name used in the same session, use the long form `SG['Perm2Rd']`.

Examples:

```
> with(SG):
> Perm2Rd([4,2,6,1,3,5]);

                         [3, 2, 1, 2, 5, 4, 3]
```

SEE ALSO: `Perm2Rothe`, `Rd2Perm`

5.2.39 `Perm2Rise` — compute the list of rises of a permutation

Calling Sequence:

`Perm2Rise`(*perm*)

`SG[Perm2Rise]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2Rise` function computes the list of rises of the permutation *perm*, *i.e.* the list of positions i, such that $perm[i] < perm[i+1]$.
- When called with the second argument $'nb'$, it returns the number of rises of the permutation *perm*.
- When called with the second argument $'rin'$, it returns the sum of the rise positions (rise index).
- When called with the second argument $'corin'$, it returns the sum of the values (n - rise position), where n is equal to `nops`($perm$).
- Whenever there is a conflict between the function name `Perm2Rise` and another name used in the same session, use the long form `SG['Perm2Rise']`.

Examples:

```
> with(SG):
> perm:=[3,1,4,2,6,5,7,8]:
> Perm2Rise(perm);

                              [2, 4, 6, 7]

> Perm2Rise(perm, 'nb');

                                   4

> Perm2Rise(perm, 'rin');

                                   9

> Perm2Rise(perm, 'corin');

                                   13
```

SEE ALSO: `Perm2Desc`

5.2.40 `Perm2Rothe` — calculate the Rothe diagram of a permutation

Calling Sequence:

`Perm2Rothe`($perm$)

`SG[Perm2Rothe]`($perm$)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2Rothe` function calculates the Rothe diagram of a permutation.
- The matrix of the `x`'s is the matrix representing the permutation.
- Reading the entries from right to left and top to bottom, one obtains a canonical reduced decomposition of the permutation (i stands for the simple transposition exchanging i and $i+1$).

- Whenever there is a conflict between the function name `Perm2Rothe` and another name used in the same session, use the long form `SG['Perm2Rothe']`.

Examples:

```
> with(SG):
> Perm2Rothe([6,1,4,5,3,2]);

                    [ 1  2  3  4  5  x ]
                    [                  ]
                    [ x  .  .  .  .  . ]
                    [                  ]
                    [ .  3  4  x  .  . ]
                    [                  ]
                    [ .  4  5  .  x  . ]
                    [                  ]
                    [ .  5  x  .  .  . ]
                    [                  ]
                    [ .  x  .  .  .  . ]

> Perm2Rd([6,1,4,5,3,2]);

                 [5, 4, 3, 2, 1, 4, 3, 5, 4, 5]
```

SEE ALSO: `Perm2Rd, Rothe2Perm`

5.2.41 `Perm2Part` — calculate the shape of a permutation

Calling Sequence:

`Perm2Part`(*perm*)

`SG[Perm2Part]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2Part` function calculates the shape of *perm*, that is the partition obtained by reordering the components of its Lehmer code.
- Whenever there is a conflict between the function name `Perm2Part` and another name used in the same session, use the long form `SG['Perm2Part']`.

Examples:

```
> with(SG):
> Perm2Code([2,1,5,4,3]);

                       [1, 0, 2, 1, 0]

> Perm2Part([2,1,5,4,3]);

                         [2, 1, 1]
```

SEE ALSO: `Code2Perm, Perm2Code`

5.2.42 `ListPerm` — generate all permutations

✓

Calling Sequence:

`ListPerm`(n, str)

`ListPerm`(e, str)

`SG[ListPerm]`(n, str)

`SG[ListPerm]`(e, str)

Parameters:

n — any positive integer denoting the degree of a symmetric group

e — any list or set of elements

str — `'nb'`, `'lexic'`, `'cixel'` or `'level'`

Synopsis:

- The `ListPerm` function generates the permutations.
- If the first argument is an integer, say n, then the function returns the list of all permutations of the symmetric group of degree n.
- If the first argument is a list l, then it generates all distinct permutations of the elements of the list. It does the same work if the first argument is a set.
- `ListPerm`$(x,$ `'nb'`) returns `nops(ListPerm(x))`.
- `ListPerm`$(n,$ `'lexic'`) or `ListPerm`$(n,$ `'cixel'`) returns all permutations in lexicographic or reverse lexicographic order.
- `ListPerm`$(n,$ `'level'`) returns a list of lists. First list contains all permutations with 0 inversion, second one contains all permutations with 1 inversion, ..., last one contains all permutations with $n(n-1)/2$ inversions.
- Whenever there is a conflict between the function name `ListPerm` and another name used in the same session, use the long form `SG['ListPerm']`.

✂ ....

Examples:

```
> with(SG):
> ListPerm(7, 'nb');

                                        5040

> ListPerm([a,a,b], 'nb');

                                         3

> ListPerm(3, 'cixel');

    [[3, 2, 1], [2, 3, 1], [3, 1, 2], [1, 3, 2], [2, 1, 3], [1, 2, 3]]

> ListPerm(3, 'level');
```

```
[[[1, 2, 3]], [[1, 3, 2], [2, 1, 3]], [[3, 1, 2], [2, 3, 1]],
 [[3, 2, 1]]]
```

SEE ALSO: `FirstPerm`, `LastPerm`, `NextPerm`, `PrevPerm`

5.2.43 `PrevPerm` — compute the previous permutation according to the lexicographic order

Calling Sequence:

`PrevPerm`(*perm*)

`SG[PrevPerm]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `PrevPerm` function calculates the previous permutation according to the lexicographic order. The function returns the `NULL` value when the argument is the first permutation.
- Whenever there is a conflict between the function name `PrevPerm` and another name used in the same session, use the long form `SG['PrevPerm']`.

Examples:

```
> with(SG):
> PrevPerm([5,2,4,3,1]);
```

```
[5, 2, 4, 1, 3]
```

SEE ALSO: `FirstPerm`, `LastPerm`, `NextPerm`

5.2.44 `RRep2Perm` — calculate a permutation from its matrix in the regular representation

Calling Sequence:

`RRep2Perm`(m)

`SG[RRep2Perm]`(m)

Parameters:

m — a square matrix

Synopsis:

- The `RRep2Perm` function calculates a permutation from its matrix in the regular representation, that is to say the matrix of the linear operator "left multiplication by *perm*".
- Whenever there is a conflict between the function name `RRep2Perm` and another name used in the same session, use the long form `SG['RRep2Perm']`.

Examples:

```
> with(SG):
> Perm2RRep([3,1,2]);

                              [ 0  0  0  1  0  0 ]
                              [                  ]
                              [ 0  0  1  0  0  0 ]
                              [                  ]
                              [ 0  0  0  0  0  1 ]
                              [                  ]
                              [ 0  0  0  0  1  0 ]
                              [                  ]
                              [ 1  0  0  0  0  0 ]
                              [                  ]
                              [ 0  1  0  0  0  0 ]

> RRep2Perm(");

                                   [3, 1, 2]
```

SEE ALSO: `Mat2Perm`, `Perm2Mat`, `Perm2RRep`

5.2.45 `RandDominant` — return a random dominant permutation

Calling Sequence:

`RandDominant`(n)

`SG[RandDominant]`(n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `RandDominant` function generates a random dominant permutation.
- Whenever there is a conflict between the function name `RandDominant` and another name used in the same session, use the long form `SG['RandDominant']`.

Examples:

```
> with(SG):
> Perm2Code( RandDominant(10) );

                       [7, 7, 3, 3, 1, 1, 0, 0, 0, 0]
```

SEE ALSO: `IsDominant`, `Perm2Code`

5.2.46 RandGrassmannian — return a random Grassmannian permutation

Calling Sequence:

RandGrassmannian(n)

SG[RandGrassmannian](n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The RandGrassmannian function generates a random Grassmannian permutation.
- Whenever there is a conflict between the function name RandGrassmannian and another name used in the same session, use the long form SG['RandGrassmannian'].

Examples:

```
> with(SG):
> RandGrassmannian(10);

                    [2, 4, 5, 6, 7, 8, 9, 10, 1, 3]
```

SEE ALSO: IsGrassmannian, RandDominant

5.2.47 RandPerm — return a random permutation

Calling Sequence:

RandPerm(n)

SG[RandPerm](n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The RandPerm function generates a random permutation belonging to the symmetric group of degree n.
- Whenever there is a conflict between the function name RandPerm and another name used in the same session, use the long form SG['RandPerm'].

Examples:

```
> with(SG):
> RandPerm(7);

                    [7, 4, 2, 3, 5, 1, 6]
```

SEE ALSO: RandDominant, RandGrassmannian

5.2.48 `Rd2Perm` — calculate a permutation from a decomposition

Calling Sequence:

`Rd2Perm`(rd)

`Rd2Perm`(rd, n)

`SG[Rd2Perm]`(rd)

`SG[Rd2Perm]`(rd, n)

Parameters:

rd — a list denoting a decomposition

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `Rd2Perm` function calculates the permutation corresponding to the list rd, the entries of rd being interpreted as simple transpositions. Let $rd = [i, j, k, \ldots]$, then `Rd2Perm`(rd)= `MultPerm`($s_i, s_j, s_k, \ldots$) where s_i stands for `Transpo`(i, n).
- When the second argument is not present, n is the smallest possible order.
- The decomposition needs not to be reduced.
- Whenever there is a conflict between the function name `Rd2Perm` and another name used in the same session, use the long form `SG['Rd2Perm']`.

Examples:

```
> with(SG):
> Rd2Perm([1,2,1], 5);

                                    [3, 2, 1, 4, 5]

> Rd2Perm([1,2,1]);

                                       [3, 2, 1]
```

SEE ALSO: `Perm2Rd`

5.2.49 `Rothe2Perm` — calculate the permutation from its Rothe diagram

Calling Sequence:

`Rothe2Perm`(m)

`SG[Rothe2Perm]`(m)

Parameters:

m — a square matrix

Synopsis:

- The `Rothe2Perm` function calculates a permutation from its Rothe diagram.
- The matrix of the `x`'s is the matrix representing the permutation.

- Reading the entries from right to left and top to bottom, one obtains a canonical reduced decomposition of the permutation (i stands for the simple transposition exchanging i and $i+1$).
- Whenever there is a conflict between the function name `Rothe2Perm` and another name used in the same session, use the long form `SG['Rothe2Perm']`.

Examples:

```
> with(SG):
> Perm2Rothe([3,5,1,4,2]);

                    [ 1  2  x  .  . ]
                    [               ]
                    [ 2  3  .  4  x ]
                    [               ]
                    [ x  .  .  .  . ]
                    [               ]
                    [ .  4  .  x  . ]
                    [               ]
                    [ .  x  .  .  . ]

> Rothe2Perm(");

                     [3, 5, 1, 4, 2]
```

SEE ALSO: `Perm2Rothe`

5.2.50 `Perm2Sign` — return the signature of a permutation

Calling Sequence:

`Perm2Sign`(*perm*)

`SG[Perm2Sign]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Perm2Sign` function calculates the signature of a permutation. This value is equal to $(-1)^{(\text{the length of the permutation})}$.
- Whenever there is a conflict between the function name `Perm2Sign` and another name used in the same session, use the long form `SG['Perm2Sign']`.

Examples:

```
> with(SG):
> Perm2Sign([1,4,2,3]);

                           1
```

SEE ALSO: `Perm2Length, Transpo`

✓

5.2.51 `Transpo` — return a transposition

Calling Sequence:

`Transpo`(i)

`Transpo`(i,n)

`Transpo`(i,j,n)

`SG[Transpo]`(i)

`SG[Transpo]`(i,n)

`SG[Transpo]`(i,j,n)

Parameters:

n — the degree of a symmetric group

i — any positive integer

j — any positive integer

Synopsis:

- The `Transpo` function returns a transposition, that is a permutation interchanging two values and fixing all others.
- `Transpo`(i)=$[1,\ldots,i+1,i]$.
- `Transpo`(i,n)=$[1,\ldots,i+1,i,\ldots,n]$.
- `Transpo`(i,j,n) =$[1,\ldots,j,\ldots,i,\ldots,n]$.
- Whenever there is a conflict between the function name `Transpo` and another name used in the same session, use the long form `SG['Transpo']`.

Examples:

```
> with(SG):
> Transpo(2, 6);

                    [1, 3, 2, 4, 5, 6]

> Transpo(3, 5, 7);

                    [1, 2, 5, 4, 3, 6, 7]
```

SEE ALSO:

5.2.52 `Vect2Perm` — compute a canonical permutation from a vector

Calling Sequence:

`Vect2Perm`(l)

`SG[Vect2Perm]`(l)

Parameters:

l — any list of positive integers

Synopsis:

- The Vect2Perm function calculates a canonical permutation from the list l, having as cycle type, the reordering of l.
- Let $l = [3, 2, 4]$, Vect2Perm(l) gives a permutation in $\mathfrak{S}_3 \times \mathfrak{S}_2 \times \mathfrak{S}_4$ which is a direct product of cycles of length 3, 2, 4.
- Whenever there is a conflict between the function name Vect2Perm and another name used in the same session, use the long form SG['Vect2Perm'].

Examples:

```
> with(SG):
> Vect2Perm([3,1,4,1]);

                        [3, 1, 2, 4, 8, 5, 6, 7, 9]
```

SEE ALSO: Cycle2Perm, Perm2Cycle, Perm2CycleType

5.3 Using SGA to work with the symmetric group algebra

Calling Sequence:

SGA[<function>]($args$)

<function>($args$)

Synopsis:

- This package provides functions to work with the symmetric group algebra, together with its realization as an algebra of operators on the ring of polynomials.
- To get more information on permutations do either ?TYP,Permutation or ?TYP[Permutation].
- Whenever there is a conflict between a function name in SGA and another name used in the same session, use the long form SGA[<function>].
- The available functions are:

Carre	CentralIdemp	Jucis	Nabla
SgaAdd	SgaEmbed	SgaMinus	SgaMult
SgaOnPol	SgaOnX	SgaOnXfix	SgaYang

- For help with a particular function do ?SGA[<function>] or ?SGA,<function> where <function> is one from the above list.
- Instead of SgaAdd, SgaMinus, SgaMult, one can use in short &!+, &!-, &!*.

Examples:

```
> with(SGA):
> Carre([3,1,2]) &!* Nabla([3,1,2]);

      1/2 [3, 1, 2] - [1, 2, 3] - 1/2 [1, 3, 2] - 1/2 [3, 2, 1]

                    + 1/2 [2, 3, 1] + [2, 1, 3]
```

5.3.1 `Carre` — a special element of the symmetric group algebra

Calling Sequence:

`Carre`(*perm*)

`SGA[Carre]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Carre` function calculates a special element of the symmetric group algebra.
- The `Carre` together with the `Nabla` functions are the building blocks of idempotents in the symmetric group algebra.
- Given a permutation *perm*, `Carre`(*perm*) is a certain product of factors of the type (`Transpo`(i, n) $+ 1/k$) where the shifts $1/k$ are given by Yang-Baxter conditions. Special cases provide idempotents. For example, `Carre`$([n, n-1, ..., 2, 1])$ = sum of all permutations of $\mathfrak{S}_n$.
- They are defined by the recursion: for a simple transposition s_i and a permutation *perm*, such that $\text{length}(perm\, s_i) > \text{length}(perm)$, then `Carre`$(perm\, s_i)$ = `Carre`$(perm)$ `&!*` $(s_i + 1/k)$, with $k = perm[i+1] - perm[i]$.
- Whenever there is a conflict between the function name `Carre` and another name used in the same session, use the long form `SGA['Carre']`.

Examples:

```
> with(SGA):
> Carre([2,3,1]);

          1/2 [1, 3, 2] + 1/2 [1, 2, 3] + [2, 1, 3] + [2, 3, 1]
```

SEE ALSO: `Nabla`

5.3.2 CentralIdemp — compute a central idempotent of the symmetric group algebra

Calling Sequence:

`CentralIdemp`(*part*)

`SGA[CentralIdemp]`(*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The `CentralIdemp` function gives the central idempotent corresponding to the partition *part*, as an element of the symmetric group algebra.
- Whenever there is a conflict between the function name `CentralIdemp` and another name used in the same session, use the long form `SGA['CentralIdemp']`.

Examples:

```
> with(SGA):
> CentralIdemp([2,1,1]);

                2/3 [1, 2, 3] - 1/3 [3, 1, 2] - 1/3 [2, 3, 1]
```

SEE ALSO:

5.3.3 SgaEmbed — reduce each permutation inside an expression to the minimum symmetric group containing it

Calling Sequence:

`SgaEmbed`(e)

`SgaEmbed`(e, m)

`SGA[SgaEmbed]`(e)

`SGA[SgaEmbed]`(e, m)

Parameters:

e — any element of the symmetric group algebra

m — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `SgaEmbed` function reduces each permutation inside an expression to the minimum symmetric group containing it ($\mathfrak{S}_n$ is contained in $\mathfrak{S}_{n+1}$ by addition of the fixed point $n + 1$).
- If a parameter m has been specified, the minimum is choosen to be m.
- Whenever there is a conflict between the function name `SgaEmbed` and another name used in the same session, use the long form `SGA['SgaEmbed']`.

Examples:

```
> with(SGA):
> SgaEmbed(a*[3,1,2] - b^2*[2,1,3,4]);

                              2
                           - b  [2, 1] + a [3, 1, 2]

> SgaEmbed(a*[3,1,2] - b^2*[2,1,3,4], 3);

                              2
                           - b  [2, 1, 3] + a [3, 1, 2]
```

SEE ALSO:

5.3.4 `Jucis` — compute a Jucis-Murphy element

Calling Sequence:

`Jucis`(i, n)

`SGA[Jucis]`(i, n)

Parameters:

i, n — any positive integers

Synopsis:

- The `Jucis` function computes the i-th Jucis-Murphy element inside the algebra of the symmetric group of degree n ($i \leq n$). The algebra generated by the Jucis-Murphy elements is a maximal commutative sub-algebra of the symmetric group algebra.
- More explicitly, denoting $T(i, j)$ the transposition exchanging i and j, we have: `Jucis`$(1) = 0$, `Jucis`$(2) = T(1, 2)$, `Jucis`$(3) = T(1, 3) + T(2, 3)$, ..., `Jucis`$(k) = T(1, k) + T(2, k) + \cdots + T(k - 1, k)$.
- Whenever there is a conflict between the function name `Jucis` and another name used in the same session, use the long form `SGA['Jucis']`.

Examples:

```
> with(SGA):
> Jucis(2,4);

                                  [2, 1, 3, 4]

> Jucis(4,6);

        [4, 2, 3, 1, 5, 6] + [1, 4, 3, 2, 5, 6] + [1, 2, 4, 3, 5, 6]
```

SEE ALSO: `SG[Transpo]`

5.3.5 `Nabla` — a special element of the symmetric group algebra

Calling Sequence:

`Nabla`$(perm)$

`SGA[Nabla]`$(perm)$

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The `Nabla` function calculates a special element of the symmetric group algebra.
- The `Nabla` together with the `Carre` functions are the building blocks of idempotents in the symmetric group algebra.
- `Nabla`$(Identity) = 0$.
- Given a permutation $perm$, `Nabla`$(perm)$ is a certain product of factors of the type (`Transpo`(i, n) - $1/k$) where the shifts $1/k$ are given by Yang-Baxter conditions.
- They are defined by the recursion: for a simple transposition s_i and a permutation $perm$, such that length$(perm\, s_i) >$ length$(perm)$, then `Nabla`$(perm\, s_i) =$ `Nabla`$(perm)$ `&!*` $(s_i - 1/k)$, with $k = perm[i+1] - perm[i]$.
- Whenever there is a conflict between the function name `Nabla` and another name used in the same session, use the long form `SGA['Nabla']`.

Examples:

```
> with(SGA):
> Nabla([2,3,1]);

          1/2 [1, 2, 3] - [2, 1, 3] - 1/2 [1, 3, 2] + [2, 3, 1]
```

SEE ALSO: `Carre`

5.3.6 `SgaAdd` — sum of two elements of the symmetric group algebra

Calling Sequence:

`SgaAdd`(e_1, e_2)

`SGA[SgaAdd]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of the symmetric group algebra

Synopsis:

- The `SgaAdd` function adds two elements of the symmetric group algebra.
- One can use instead the operator `&!+`.
- Whenever there is a conflict between the function name `SgaAdd` and another name used in the same session, use the long form `SGA['SgaAdd']`.

Examples:

```
> with(SGA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> SgaAdd(a, b);

                u [3, 1, 2]
                ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                     t

> a &!+ b;

                u [3, 1, 2]
                ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                     t
```

SEE ALSO:

5.3.7 SgaMinus — difference of two elements of the symmetric group algebra

Calling Sequence:

`SgaMinus`(e_1, e_2)

`SGA[SgaMinus]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of the symmetric group algebra

Synopsis:

- The `SgaMinus` function computes the difference between two elements of the symmetric group algebra.
- One can use instead the operator `&!-`.
- Whenever there is a conflict between the function name `SgaMinus` and another name used in the same session, use the long form `SGA['SgaMinus']`.

Examples:

```
> with(SGA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> SgaMinus(a, b);

                u [3, 1, 2]
                ----------- + [1, 3, 2] + (q - t) [1, 2, 3]
                     t

> a &!- a;

                                  0
```

SEE ALSO: `TYP[Permutation]`

5.3.8 SgaMult — multiplication of two elements of the symmetric group algebra

Calling Sequence:

SgaMult(e_1, e_2)

SGA[SgaMult](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the symmetric group algebra

Synopsis:

- The SgaMult function multiplies two elements of the symmetric group algebra.
- One can use instead the operator &!*.
- Whenever there is a conflict between the function name SgaMult and another name used in the same session, use the long form SGA['SgaMult'].

Examples:

```
> with(SGA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> SgaMult(a, b);

                                                    u [3, 2, 1]
          u [3, 1, 2] - q [1, 3, 2] + q t [1, 2, 3] - -----------
                                                         t

> a &!* b;

                                                    u [3, 2, 1]
          u [3, 1, 2] - q [1, 3, 2] + q t [1, 2, 3] - -----------
                                                         t
```

SEE ALSO:

5.3.9 SgaOnPol — action of an element of the algebra on an expression

Calling Sequence:

SgaOnPol(e_1, exp)

SgaOnPol(e_1, exp, v)

SGA[SgaOnPol](e_1, exp)

SGA[SgaOnPol](e_1, exp, v)

Parameters:

e_1 — any element of the symmetric group algebra

exp — any expression

v — any (extra) string

Synopsis:

- The `SgaOnPol` function realizes the action of an element of the symmetric algebra, say e_1, on an expression *exp*. The permutation action is to permute variable indices.
- By default the algebra acts on the variables $x_1, x_2, x_3, \ldots$
- A simple transposition, say $[1, 2, 4, 3, 5, 6]$, exchanges the variables x_3 and x_4.
- When called with a third argument v, being, say y the `SgaOnPol` function acts on the variables $y_1, y_2, y_3, \cdots$
- Whenever there is a conflict between the function name `SgaOnPol` and another name used in the same session, use the long form `SGA['SgaOnPol']`.

Examples:

```
> with(SGA):
> SgaOnPol([2,1], x1+x3);

                              x2 + x3

> SgaOnPol([2,1], x1+x3, y);

                              x1 + x3

> SgaOnPol([2,1] - x3*[3,1,2], x1+x3);

                    - x3 (x2 + x3) + x2 + x3
```

SEE ALSO:

5.3.10 `SgaOnX` — action of an element of the symmetric group algebra on a linear combination of X_{perm}

✓

Calling Sequence:

`SgaOnX`(e_1, exp)

`SGA[SgaOnX]`(e_1, exp)

Parameters:

e_1 — any element of the symmetric group algebra

exp — any expression

Synopsis:

- The `SgaOnX` function realizes the action of an element of the symmetric group algebra, say e_1, on an expression *exp* expressed on the $\mathbf{X}$ Schubert basis.
- The expression *exp* is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument *exp* to choose not to expand the expression *exp* before treating it.

- One may collect the result by adding a third argument: `SgaOnX`(e_1, exp, `'collect'`). Moreover, one can use both `noexpand` and `collect` options: for instance,

  `SgaOnX`(e_1, exp, `'noexpand', 'collect'`)

- The action of the element `Transpo`(i, n) on the Schubert polynomial $\mathbf{X}_{perm}$ is:
  let ν =`MultPerm`($perm$, `Transpo`(i, n)), then

$$\begin{array}{lcll} \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_{perm} & \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_{perm} + (x_{i+1} - x_i)\mathbf{X}_{\nu} & \text{otherwise.} \end{array}$$

- The result is expressed on the **X** Schubert basis and is not collected.

- Whenever there is a conflict between the function name `SgaOnX` and another name used in the same session, use the long form `SGA['SgaOnX']`.

Examples:

```
> with(SGA):
> SgaOnX(q^4*[1,3,2] - q^3*[2,1], z*X[3,2,1] - X[2,1,3]);

                       4                    4                  3
   - X[3, 2, 1] z q  + X[3, 1, 4, 2] z q  + X[3, 2, 1] z q

                         3    4                    3               3
  - X[2, 4, 1, 3] z q  - q  X[2, 1] - X[2, 1] q  + X[1, 3, 2] q
```

SEE ALSO: `SP`

5.3.11 `SgaOnXfix` — action of an element of the symmetric group algebra on the free module

✘

Calling Sequence:

`SgaOnXfix`(e_1, exp)

`SGA[SgaOnXfix]`(e_1, exp)

Parameters:

e_1 — any element of the symmetric group algebra

exp — any expression

Synopsis:

- The `SgaOnXfix` function realizes the action of an element of the symmetric group algebra, say e_1, on an expression exp expressed on the **X** Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.

- The expression exp is expanded and the result is not collected.

- One may add `'noexpand'` just after the argument exp to choose not to expand the expression exp before treating it.

- One may collect the result by adding a third argument: `SgaOnXfix`(e_1, exp, `'collect'`). Moreover, one can use both `noexpand` and `collect` options:

$$\texttt{SgaOnXfix}(e_1, exp, \texttt{'noexpand', 'collect'})$$

- The action of the element `Transpo`(i, n) on the Schubert polynomial $\mathbf{X}_{perm}$ is:
  let $\nu =$ `MultPerm`($perm$, `Transpo`(i, n)), then:

$$\begin{array}{lcll} \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_{perm} & \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_{perm} + (x_{i+1} - x_i)\mathbf{X}_{\nu} & \text{otherwise.} \end{array}$$

- The result is expressed on the **X** Schubert basis corresponding to a given symmetric group and is not collected.
- Whenever there is a conflict between the function name `SgaOnXfix` and another name used in the same session, use the long form `SGA['SgaOnXfix']`.

Examples:

```
> with(SGA):
> _FMn;

                                        4

> SgaOnXfix(q*[3,2,1], s[1,1]*X[2,4,1,3], collect);

                               3                  3
   (s[2, 1] + s[1, 1, 1]) q  X[1, 3, 4, 2] - q  X[2, 3, 4, 1] s[1, 1]

                                                  3
    + (s[2, 2] + s[2, 1, 1] + s[1, 1, 1, 1]) q  X[2, 1, 3, 4]

                                                          3
    - (s[2, 2, 1] + s[2, 1, 1, 1] + s[1, 1, 1, 1, 1]) q  X[1, 2, 3, 4]

                                  3                 3
    - (s[2, 1] + s[1, 1, 1]) q  X[2, 1, 4, 3] + q  X[4, 1, 2, 3] s[1, 1]
```

SEE ALSO: `FM`, `SP`

5.3.12 SgaYang — compute a special element of the symmetric group algebra

Calling Sequence:

`SgaYang`($perm$)

`SGA[SgaYang]`($perm$)

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The SgaYang function calculates a special element of the symmetric group algebra.
- When called with a second parameter, say $'y'$, one specifies that coefficients are in the y_i's.
- When this second parameter is 'num' then $x_1, x_2, x_3, \ldots$ are specialized to $1, 2, 3, \ldots$
- { SgaYang(*perm*), $perm \in \mathfrak{S}_n$ } is a linear basis of the symmetric group algebra, as a free module with coefficients in the x_i's.
- This basis is defined by the recursion: for a simple transposition s_k and a permutation *perm*, such that $length(perm\, s_k) > length(perm)$, then SgaYang$(perm\, s_k)$ = SgaYang$(perm)$ &!* $(1 + (x_j - x_i)s_k)$ where $i = perm[k]$ and $j = perm[k+1]$.
- Whenever there is a conflict between the function name SgaYang and another name used in the same session, use the long form SGA['SgaYang'].

Examples:

```
> with(SGA):
> SgaYang([3,1,2]);

         (x3 - x2) [1, 3, 2] + [1, 2, 3] + (x3 - x1) [2, 1, 3]

              - (- x3 + x2) (x3 - x1) [3, 1, 2]

> SgaYang([3,1,2], 'y');

         (y3 - y2) [1, 3, 2] + [1, 2, 3] + (y3 - y1) [2, 1, 3]

              - (- y3 + y2) (y3 - y1) [3, 1, 2]
```

SEE ALSO:

5.4 NCA to compute in the nilCoxeter algebra

✓

Calling Sequence:

NCA[<function>]$(args)$

<function>$(args)$

Synopsis:

- This package provides functions to work with the nilCoxeter algebra corresponding to the symmetric group. This algebra is realized as the algebra of divided differences acting on polynomials.
- This algebra is generated by simple divided differences ∂_i satisfying the braid relations, together with $\partial_i^2 = 0$ for all i.
- Whenever there is a conflict between a function name in NCA and another name used in the same session, use the long form NCA[<function>].
- The available functions are:

NcaAdd	NcaEmbed	NcaMinus	NcaMult
NcaOnPol	NcaOnX	NcaOnXfix	NcaOnXX
NcaOnY	NcaOnYY	NcaYang	Rd2Nca

- For help with a particular function do `?NCA[<function>]` or `?NCA,<function>` where `<function>` is one from the above list.
- Instead of `NcaAdd`, `NcaMinus`, `NcaMult`, one can use in short `&@+`, `&@-`, `&@*`.

Examples:

```
> with(NCA):
> a:=q + u/t*[3,1,2]:
> b:=t*[2,1,3] - [1,3,2]:
> NcaMult(a, b);

                    u [3, 2, 1]
                  - ----------- - q [1, 3, 2] + q t [2, 1, 3]
                         t
```

5.4.1 `NcaEmbed` — reduce each permutation inside an expression to the minimum symmetric group containing it

Calling Sequence:

`NcaEmbed`(e)

`NcaEmbed`(e, m)

`NCA[NcaEmbed]`(e)

`NCA[NcaEmbed]`(e, m)

Parameters:

e — any element of the nilCoxeter algebra

m — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `NcaEmbed` function reduces each permutation inside an expression to the minimum symmetric group containing it ($\mathfrak{S}_n$ is contained in $\mathfrak{S}_{n+1}$ by addition of the fixed point $n+1$).
- If a parameter m has been specified, the minimum is choosen to be m.
- Whenever there is a conflict between the function name `NcaEmbed` and another name used in the same session, use the long form `NCA['NcaEmbed']`.

Examples:

```
> with(NCA):
> NcaEmbed(a*[3,1,2] - b^2*[2,1,3,4]);

                                  2
                               - b  [2, 1] + a [3, 1, 2]

> NcaEmbed(a*[3,1,2] - b^2*[2,1,3,4], 3);

                                 2
                              - b  [2, 1, 3] + a [3, 1, 2]
```

SEE ALSO:

5.4.2 NcaAdd — sum of two elements of the nilCoxeter algebra

Calling Sequence:

NcaAdd(e_1, e_2)

NCA[NcaAdd](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the nilCoxeter algebra

Synopsis:

- The NcaAdd function adds two elements of the nilCoxeter algebra.
- One can use instead the operator &@+.
- Whenever there is a conflict between the function name NcaAdd and another name used in the same session, use the long form NCA['NcaAdd'].

Examples:

```
> with(NCA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> NcaAdd(a, b);

                  u [3, 1, 2]
                  ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                       t

> a &@+ b;

                  u [3, 1, 2]
                  ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                       t
```

SEE ALSO:

5.4.3 NcaMinus — difference of two elements of the nilCoxeter algebra

Calling Sequence:

NcaMinus(e_1, e_2)

NCA[NcaMinus](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the nilCoxeter algebra

Synopsis:

- The NcaMinus function computes the difference between two elements of the nilCoxeter algebra.
- One can use instead the operator &@-.
- Whenever there is a conflict between the function name NcaMinus and another name used in the same session, use the long form NCA['NcaMinus'].

Examples:

```
> with(NCA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> NcaMinus(a, b);

                    u [3, 1, 2]
                    ----------- + [1, 3, 2] + (q - t) [1, 2, 3]
                         t

> a &@- a;
                                        0
```

SEE ALSO:

5.4.4 NcaMult — multiplication of two elements of the nilCoxeter algebra

Calling Sequence:

NcaMult(e_1, e_2)

NCA[NcaMult](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the nilCoxeter algebra

Synopsis:

- The NcaMult function multiplies two elements of the nilCoxeter algebra.
- One can use instead the operator &@*.
- Whenever there is a conflict between the function name NcaMult and another name used in the same session, use the long form NCA['NcaMult'].

Examples:

```
> with(NCA):
> a:=q+s/t*[3,1,2]:
> b:=t*[2,1,3]-[1,3,2]:
> NcaMult(a,b);

                   s [3, 2, 1]
                 - ----------- - q [1, 3, 2] + q t [2, 1, 3]
                        t

> a &@* a;

                      s q [3, 1, 2]     2
                    2 ------------- + q  [1, 2, 3]
                            t
```

SEE ALSO:

5.4.5 NcaOnPol — action of an element of the algebra on a polynomial

Calling Sequence:

NcaOnPol(e_1, exp)

NcaOnPol(e_1, exp, v)

NCA[NcaOnPol](e_1, exp)

NCA[NcaOnPol](e_1, exp, v)

Parameters:

e_1 — any element of the nilCoxeter algebra

exp — any expression

v — any (extra) string

Synopsis:

- The `NcaOnPol` function realizes the action of an element of the nilCoxeter algebra, say e_1, on an expression exp.
- Simple divided differences ∂_i are defined to be the operators:

$$f \longrightarrow \frac{f(\ldots, x_i, x_{i+1}, \ldots) - f(\ldots, x_{i+1}, x_i, \ldots)}{x_i - x_{i+1}} \quad .$$

- By default the algebra acts on the variables $x_1, x_2, x_3, \ldots$
- When called with a third argument v, being, say y the `NcaOnPol` function acts on the variables $y_1, y_2, y_3, \ldots$
- Whenever there is a conflict between the function name `NcaOnPol` and another name used in the same session, use the long form `NCA['NcaOnPol']`.

Examples:

```
> with(NCA):
> NcaOnPol(q^2*[3,1,2], x1^2*x2*y1^2*y2^3);

                                2      2   3
                               q  x1 y1  y2

> NcaOnPol(q^2*[3,1,2], x1^2*x2*y1^2*y2^3, y);

                            2             2   2
                         - q  (y2 + y3) y1  x1  x2
```

SEE ALSO:

5.4.6 NcaOnX — action of an element of the nilCoxeter algebra on a linear combination of $\mathbf{X}_{perm}$

✓

Calling Sequence:

`NcaOnX`(e_1, exp)

`NCA[NcaOnX]`(e_1, exp)

Parameters:

e_1 — any element of the nilCoxeter algebra

exp — any expression

Synopsis:

- The `NcaOnX` function realizes the action of an element of the nilCoxeter algebra, say e_1, on an expression exp expressed on the $\mathbf{X}$ Schubert basis.
- The expression exp is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument exp to choose not to expand the expression exp before treating it.
- One may collect the result by adding a third argument: `NcaOnX`$(e_1, exp,$ `'collect'`). Moreover, one can use both `noexpand` and `collect` options:

  `NcaOnX`$(e_1, exp,$ `'noexpand'`, `'collect'`)

- A simple divided difference ∂_i acts on a Schubert polynomial $\mathbf{X}_{perm}$ by sending it to 0 if $perm[i] < perm[i+1]$, or to $\mathbf{X}_\nu$ where ν is obtained by transposing $perm[i]$ and $perm[i+1]$, if $perm[i] > perm[i+1]$.
- The result is expressed on the $\mathbf{X}$ Schubert basis and is not collected.
- Whenever there is a conflict between the function name `NcaOnX` and another name used in the same session, use the long form `NCA['NcaOnX']`.

Examples:

```
> with(NCA):
> NcaOnX(q^4*[1,3,2] - q^3*[2,1], z*X[3,2,1] - X[2,1,3]);

                   4                 3                3
              z q  X[3, 1, 2] - z q  X[2, 3, 1] + q  X[1, 2]
```

SEE ALSO: NcaOnY, SP

5.4.7 NcaOnXX — action of an element of the nilCoxeter algebra on a linear combination of XX[*perm*]

✓

Calling Sequence:

NcaOnXX(e_1, exp)

NCA[NcaOnXX](e_1, exp)

Parameters:

e_1 — any element of the nilCoxeter algebra

exp — any expression

Synopsis:

- The NcaOnXX function realizes the action of an element of the nilCoxeter algebra, say e_1, on an expression *exp* expressed on the $\mathbb{X}$ Schubert basis (double Schubert polynomials).
- One may add 'noexpand' just after the argument *exp* to choose not to expand the expression *exp* before treating it.
- One may collect the result by adding a third argument: this is done by NcaOnXX(e_1, exp, 'collect'). Moreover, one can use both noexpand and collect options:

  NcaOnXX(e_1, exp, 'noexpand', 'collect')

- A simple divided difference ∂_i acts on a Schubert polynomial $\mathbb{X}_{perm}$ by sending it to 0 if $perm[i] < perm[i+1]$, or to $\mathbb{X}_\nu$ where ν is obtained by transposing $perm[i]$ and $perm[i+1]$, if $perm[i] > perm[i+1]$.
- The result is expressed on the $\mathbb{X}$ Schubert basis and is not collected.
- Whenever there is a conflict between the function name NcaOnXX and another name used in the same session, use the long form NCA['NcaOnXX'].

Examples:

```
> with(NCA):
> NcaOnXX(q^4*[1,3,2] - q^3*[2,1], z*XX[3,2,1] - XX[2,1,3]);

                  4                  3                 3
             z q  XX[3, 1, 2] - z q  XX[2, 3, 1] + q  XX[1, 2]
```

SEE ALSO: NcaOnYY, SP

5.4.8 `NcaOnXfix` — action of an element of the nilCoxeter algebra on the free module

Calling Sequence:

`NcaOnXfix`(e_1, exp)

`NCA[NcaOnXfix]`(e_1, exp)

Parameters:

e_1 — any element of the nilCoxeter algebra

exp — any expression

Synopsis:

- The `NcaOnXfix` function realizes the action of an element of the nilCoxeter, say e_1, on an expression exp expressed on the $\mathbf{X}$ Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.

- The expression exp is expanded and the result is not collected.

- One may add `'noexpand'` just after the argument exp to choose not to expand the expression exp before treating it.

- One may collect the result by adding a third argument: `NcaOnXfix`$(e_1, exp,$ `'collect'`). Moreover, one can use both `noexpand` and `collect` options:

  `NcaOnXfix`$(e_1, exp,$ `'noexpand', 'collect'`)

- A simple divided difference ∂_i acts on a Schubert polynomial $\mathbf{X}_{perm}$ by sending it to 0 if $perm[i] < perm[i+1]$, or to $\mathbf{X}_\nu$ where ν is obtained by transposing $perm[i]$ and $perm[i+1]$, if $perm[i] > perm[i+1]$.

- The result is expressed on the $\mathbf{X}$ Schubert basis corresponding to a given symmetric group and is not collected.

- Whenever there is a conflict between the function name `NcaOnXfix` and another name used in the same session, use the long form `NCA['NcaOnXfix']`.

Examples:

```
> with(NCA):
> _FMn;

                                 4

> NcaOnXfix(q*[1,3,2], s[1,1]*X[2,4,1,3]);

                      s[1, 1] q X[2, 1, 4, 3]
```

SEE ALSO: `FM`, `SP`

5.4.9 NcaOnY — action of an element of the nilCoxeter algebra on a linear combination of Y[*code*]

✓

Calling Sequence:

NcaOnY(e_1, *exp*)

NCA[NcaOnY](e_1, *exp*)

Parameters:

e_1 — any element of the nilCoxeter algebra

exp — any expression

Synopsis:

- The NcaOnY function realizes the action of an element of the nilCoxeter algebra, say e_1, on an expression *exp* expressed on the **Y** Schubert basis.
- One may add 'noexpand' just after the argument *exp* to choose not to expand the expression *exp* before treating it.
- One may collect the result by adding a third argument: NcaOnY(e_1, *exp*, 'collect'). Moreover, one can use both noexpand and collect options:

  NcaOnY(e_1, *exp*, 'noexpand', 'collect')

- A simple divided difference ∂_i acts on a Schubert polynomial $\mathbf{Y}_{code}$ by sending it to 0 if $code[i] \leq code[i+1]$, or to $\mathbf{Y}_{new\_code}$ where *new_code* is obtained by applying $new\_code[i] = code[i+1]$ and $new\_code[i+1] = code[i] - 1$, if $code[i] > code[i+1]$.
- The result is expressed on the **Y** Schubert basis and is not collected.
- Whenever there is a conflict between the function name NcaOnY and another name used in the same session, use the long form NCA['NcaOnY'].

Examples:

```
> with(NCA):
> NcaOnY(q^4*[1,3,2] - q^3*[2,1],  z*Y[2,1] - Y[1,0,0]);

                    4                 3             3
                 z q  Y[2, 0, 0] - z q  Y[1, 1] + q  Y[0, 0]
```

SEE ALSO: NcaOnX, SP

5.4.10 NcaOnYY — action of an element of the nilCoxeter algebra on a linear combination of YY[*code*]

✓

Calling Sequence:

NcaOnYY(e_1, *exp*)

NCA[NcaOnYY](e_1, *exp*)

Parameters:

e_1 — any element of the nilCoxeter algebra

exp — any expression

Synopsis:

- The `NcaOnYY` function realizes the action of an element of the nilCoxeter algebra, say e_1, on an expression *exp* expressed on the $\mathbb{Y}$ Schubert basis (double Schubert polynomials indexed by codes).
- One may add `'noexpand'` just after the argument *exp* to choose not to expand the expression *exp* before treating it.
- One may collect the result by adding a third argument: this is done by `NcaOnYY`(e_1, *exp*, `'collect'`). Moreover, one can use both `noexpand` and `collect` options:

`NcaOnYY`(e_1, *exp*, `'noexpand', 'collect'`)

- A simple divided difference ∂_i acts on a Schubert polynomial $\mathbb{Y}_{code}$ by sending it to 0 if $code[i] \leq code[i+1]$, or to $\mathbb{Y}_{new\_code}$ where *new_code* is obtained by applying $new\_code[i] = code[i+1]$ and $new\_code[i+1] = code[i] - 1$, if $code[i] > code[i+1]$.
- The result is expressed on the $\mathbb{Y}$ Schubert basis and is not collected.
- Whenever there is a conflict between the function name `NcaOnYY` and another name used in the same session, use the long form `NCA['NcaOnYY']`.

Examples:

```
> with(NCA):
> NcaOnYY(q^4*[1,3,2] - q^3*[2,1],  z*YY[2,1] - YY[1,0,0]);

                  4                    3              3
             z q   YY[2, 0, 0] - z q   YY[1, 1] + q   YY[0, 0]
```

SEE ALSO: `NcaOnX, SP`

5.4.11 `NcaYang` — compute a special element of the nilCoxeter algebra

✓

Calling Sequence:

`NcaYang`(*perm*)

`NCA[NcaYang]`(*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `NcaYang` function calculates a special element of the nilCoxeter algebra.
- { `NcaYang`(*perm*), $perm \in \mathfrak{S}_n$ } is a linear basis of the nilCoxeter algebra, as a free module with coefficients in the x_i's.

- When called with a second parameter, say $'y'$, one specifies that coefficients are in the y_i's.
- When this second parameter is `'num'` then $x_1, x_2, x_3, \ldots$ are specialized to $1, 2, 3, \ldots$
- This basis is defined by the recursion : for a simple transposition s_k and a permutation $perm$, such that $length(perm\, s_k) > length(perm)$, then `NcaYang`$(perm\, s_k)$ = `NcaYang`$(perm)$ `&@*` $(1 + (x_j - x_i)\partial_k)$ where $i = perm[k]$ and $j = perm[k+1]$.
- Coefficients in the expansion are specializations of double Schubert polynomials.
- Whenever there is a conflict between the function name `NcaYang` and another name used in the same session, use the long form `NCA['NcaYang']`.

Examples:

```
> with(NCA):
> NcaYang([3,1,2]);

          (x3 - x2) [1, 3, 2] + [1, 2, 3] + (x3 - x1) [2, 1, 3]

                - (- x3 + x2) (x3 - x1) [3, 1, 2]
```

SEE ALSO: `SP`

5.4.12 `Rd2Nca` — calculate a product of generators

✘

Calling Sequence:

`Rd2Nca`(rd)

`Rd2Nca`(rd, n)

`NCA[Rd2Nca]`(rd)

`NCA[Rd2Nca]`(rd, n)

Parameters:

rd — any list denoting a decomposition

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `Rd2Nca` function calculates the product of the generators given by the reduced decomposition rd. The product is taken in the nilCoxeter algebra.
- If rd is reduced then `Rd2Nca`(rd)=`SG['Rd2Perm']`(rd), and 0 otherwise.
- When the second argument is not present, n is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2Nca` and another name used in the same session, use the long form `NCA['Rd2Nca']`.

Examples:

```
> with(NCA):
> Rd2Nca([1,2,1], 5);
                         [3, 2, 1, 4, 5]
> Rd2Nca([1,2,1]);
                            [3, 2, 1]
> Rd2Nca([1,2,1,2]);
                                0
```

SEE ALSO: `SG[Rd2Perm]`

5.5 `IDCA` to compute in the idCoxeter algebra

✓

Calling Sequence:

`IDCA[<function>]`$(args)$

`<function>`$(args)$

Synopsis:

- This package provides functions to work with the idCoxeter algebra. This is the algebra generated by the simple π_i operators satisfying the braid relations together with the extra relations: $\pi_i^2 = -\pi_i$.
- Whenever there is a conflict between a function name in `IDCA` and another name used in the same session, use the long form `IDCA[<function>]`.
- The available functions are:

Demazure	IdcaAdd	IdcaEmbed	IdcaMinus
IdcaMult	IdcaOnPol	IdcaOnX	IdcaOnXfix
IdcaYang	Rd2Idca		

- For help with a particular function do `?IDCA[<function>]` or `?IDCA,<function>` where `<function>` is one from the above list.
- Instead of `IdcaAdd`, `IdcaMinus`, `IdcaMult`, one can use in short `&$+`, `&$-`, `&$*`.

Examples:

```
> with(IDCA):
> a:=q + u/t*[3,1,2]:
> b:=t*[2,1,3] - [1,3,2]:
> IdcaMult(a, b);
```

```
  u [3, 2, 1]
- ----------- - u [3, 1, 2] + q t [2, 1, 3] - q [1, 3, 2]
       t
```

5.5.1 Demazure — compute a special element of the idCoxeter algebra

Calling Sequence:

`Demazure`$(perm)$

`IDCA[Demazure]`$(perm)$

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The `Demazure` function calculates a special element of the idCoxeter algebra.
- Given a permutation $perm$, with reduced decomposition $s_i s_j s_h \ldots$ where s_i is the transposition exchanging i and $i+1$, then `Demazure`$(perm)$ is equal to the product $(1+\pi_i)$ `&$*` $(1+\pi_j)$ `&$*` $(1+\pi_k)$ `&$*` $\cdots$
- `Demazure`$(perm)$ is the sum of all permutations below than $perm$ (for the Bruhat order).
- Whenever there is a conflict between the function name `Demazure` and another name used in the same session, use the long form `IDCA['Demazure']`.

Examples:

```
> with(IDCA):
> Demazure([2,3,1]);

              [2, 3, 1] + [2, 1, 3] + [1, 3, 2] + [1, 2, 3]
```

SEE ALSO:

5.5.2 IdcaEmbed — reduce each permutation inside an expression to the minimum symmetric group containing it

Calling Sequence:

`IdcaEmbed`(e)

`IdcaEmbed`(e, m)

`IDCA[IdcaEmbed]`(e)

`IDCA[IdcaEmbed]`(e, m)

Parameters:

e — any element of the idCoxeter algebra

m — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `IdcaEmbed` function reduces each permutation inside an expression to the minimum symmetric group containing it ($\mathfrak{S}_n$ is contained in $\mathfrak{S}_{n+1}$ by addition of the fixed point $n+1$).
- If a parameter m has been specified, the minimum is choosen to be m.
- Whenever there is a conflict between the function name `IdcaEmbed` and another name used in the same session, use the long form `IDCA['IdcaEmbed']`.

Examples:

```
> with(IDCA):
> IdcaEmbed(a*[3,1,2] - b^2*[2,1,3,4]);

                              2
                           - b  [2, 1] + a [3, 1, 2]

> IdcaEmbed(a*[3,1,2] - b^2*[2,1,3,4], 3);

                              2
                           - b  [2, 1, 3] + a [3, 1, 2]
```

SEE ALSO:

5.5.3 `IdcaAdd` — sum of two elements of the idCoxeter algebra

Calling Sequence:

`IdcaAdd`(e_1, e_2)

`IDCA[IdcaAdd]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of the idCoxeter algebra

Synopsis:

- The `IdcaAdd` function adds two elements of the idCoxeter algebra.
- One can use instead the operator `&$+`.
- Whenever there is a conflict between the function name `IdcaAdd` and another name used in the same session, use the long form `IDCA['IdcaAdd']`.

Examples:

```
> with(IDCA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> IdcaAdd(a, b);
```

```
                 u [3, 1, 2]
                 ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                      t

> a &$+ b;

                 u [3, 1, 2]
                 ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                      t
```

SEE ALSO:

5.5.4 IdcaMinus — difference of two elements of the idCoxeter algebra

Calling Sequence:

IdcaMinus(e_1, e_2)

IDCA[IdcaMinus](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the idCoxeter algebra

Synopsis:

- The IdcaMinus function computes the difference between two elements of the idCoxeter algebra.
- One can use instead the operator &$-.
- Whenever there is a conflict between the function name IdcaMinus and another name used in the same session, use the long form IDCA['IdcaMinus'].

Examples:

```
> with(IDCA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> IdcaMinus(a, b);

                 u [3, 1, 2]
                 ----------- + [1, 3, 2] + (q - t) [1, 2, 3]
                      t

> a &$- a;
                                    0
```

SEE ALSO:

5.5.5 IdcaMult — multiplication of two elements of the idCoxeter algebra

Calling Sequence:

IdcaMult(e_1, e_2)

IDCA[IdcaMult](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the idCoxeter algebra

Synopsis:

- The `IdcaMult` function multiplies two elements of the idCoxeter algebra.
- One can use instead the operator `&$*`.
- Whenever there is a conflict between the function name `IdcaMult` and another name used in the same session, use the long form `IDCA['IdcaMult']`.

Examples:

```
> with(IDCA):
> a:=q + u/t*[3,1,2]:
> b:=t*[2,1,3] - [1,3,2]:
> IdcaMult(a, b);

            u [3, 2, 1]
          - ----------- - u [3, 1, 2] + q t [2, 1, 3] - q [1, 3, 2]
                 t

> a &$* b;

            u [3, 2, 1]
          - ----------- - u [3, 1, 2] + q t [2, 1, 3] - q [1, 3, 2]
                 t
```

SEE ALSO:

5.5.6 `IdcaOnPol` — action of an element of the algebra on a polynomial

Calling Sequence:

`IdcaOnPol`(e_1, exp)

`IdcaOnPol`(e_1, exp, v)

`IDCA[IdcaOnPol]`(e_1, exp)

`IDCA[IdcaOnPol]`(e_1, exp, v)

Parameters:

e_1 — any element of the idCoxeter algebra

exp — any expression

v — any (extra) string

Synopsis:

- The `IdcaOnPol` function realizes the action of an element of the idCoxeter algebra, say e_1, on an expression exp.
- The action of the element π_i is:

$$exp \longrightarrow \frac{(exp - exp')}{x_i - x_{i+1}}\, x_{i+1} \quad ,$$

where exp' is the image of exp under the exchange of x_i and x_{i+1}.

- By default the algebra acts on the variables $x_1, x_2, x_3, \ldots$
- When called with a third argument v, being, say y the `IdcaOnPol` function acts on the variables $y_1, y_2, y_3, \ldots$
- Whenever there is a conflict between the function name `IdcaOnPol` and another name used in the same session, use the long form `IDCA['IdcaOnPol']`.

Examples:

```
> with(IDCA):
> IdcaOnPol(q^2*[3,1,2], x1^2*x2*y1^2*y2^3);

                         2             3   2
                        q  (x2 + x3) y2  y1  x1 x3

> IdcaOnPol(q^2*[3,1,2], x1^2*x2*y1^2*y2^3, y);

                      2    2              2    2       2
                   - q  (y2  + y3 y2 + y3 ) y1  x2 x1  y3
```

See Also:

5.5.7 IdcaOnX — action of an element of the idCoxeter algebra on a linear combination of $\mathbf{X}_{perm}$

✓

Calling Sequence:

`IdcaOnX`(e_1, exp)

`IDCA[IdcaOnX]`(e_1, exp)

Parameters:

e_1 — any element of the symmetric group algebra

exp — any expression

Synopsis:

- The `IdcaOnX` function realizes the action of an element of the idCoxeter algebra, say e_1, on an expression exp expressed on the $\mathbf{X}$ Schubert basis.
- The expression exp is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument exp to choose not to expand the expression exp before treating it.
- One may collect the result by adding a third argument: this is done by `IdcaOnX`$(e_1, exp,$ `'collect'`$)$. Moreover, one can use both `noexpand` and `collect` options:

`IdcaOnX`$(e_1, exp,$ `'noexpand', 'collect'`$)$

- The action of the element π_i on the Schubert polynomial $\mathbf{X}_{perm}$ is : let $\nu = perm\, s_i$ where s_i=Transpo(i,n), then

$$\begin{array}{llll} \mathbf{X}_{perm} & \longrightarrow & 0 & \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_\nu\, x_{i+1} & \text{otherwise.} \end{array}$$

- The result is expressed on the $\mathbf{X}$ Schubert basis and is not collected.
- Whenever there is a conflict between the function name `IdcaOnX` and another name used in the same session, use the long form `IDCA['IdcaOnX']`.

Examples:

```
> with(IDCA):
> IdcaOnX(q^4 * [1,3,2] - q^3 * [2,1], z * X[3,2,1] - X[2,1,3]);

                      4                     4                 3
       - X[3, 2, 1] z q  + X[3, 1, 4, 2] z q  + X[3, 2, 1] z q

                            3             3                3
          - X[2, 4, 1, 3] z q  - X[2, 1] q  + X[1, 3, 2] q
```

SEE ALSO: SP

5.5.8 `IdcaOnXfix` — action of an element of the idCoxeter algebra on the free module

Calling Sequence:

`IdcaOnXfix`(e_1, exp)

`IDCA[IdcaOnXfix]`(e_1, exp)

Parameters:

e_1 — any element of the symmetric group algebra

exp — any expression

Synopsis:

- The `IdcaOnXfix` function realizes the action of an element of the idCoxeter algebra, say e_1, on an expression exp expressed on the $\mathbf{X}$ Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.
- The expression exp is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument exp to choose not to expand the expression exp before treating it.
- One may collect the result by adding a third argument: `IdcaOnXfix`$(e_1, exp,$ `'collect'`). Moreover, one can use both `noexpand` and `collect` options: `IdcaOnXfix`$(e_1, exp,$ `'noexpand'`, `'collect'`).

- The action of the element π_i on the Schubert polynomial $\mathbf{X}_{perm}$ is : let $\nu = perm\, s_i$ where s_i=Transpo(i, n), then

$$\begin{array}{lll} \mathbf{X}_{perm} & \longrightarrow 0 & \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow \mathbf{X}_{\nu}\, x_{i+1} & \text{otherwise.} \end{array}$$

- The result is expressed on the **X** Schubert basis corresponding to a given symmetric group and is not collected.
- Whenever there is a conflict between the function name IdcaOnXfix and another name used in the same session, use the long form IDCA['IdcaOnXfix'].

Examples:

```
> with(IDCA):
> _FMn;

                                    4

> IdcaOnXfix(q*[1,3,2], s[1,1]*X[2,4,1,3], collect);

  - X[2, 4, 1, 3] s[1, 1] q + (s[2, 1] + s[1, 1, 1]) q X[2, 1, 4, 3]

  - X[4, 1, 2, 3] s[1, 1] q

 - (s[2, 2] + s[2, 1, 1] + s[1, 1, 1, 1]) q X[2, 1, 3, 4]
```

SEE ALSO: FM, SP

5.5.9 IdcaYang — compute a special element of the idCoxeter algebra

✓

Calling Sequence:

IdcaYang($perm$)

IDCA[IdcaYang]($perm$)

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The IdcaYang function calculates a special element of the idCoxeter algebra.
- When called with a second parameter, say $'y'$, one specifies that coefficients are in the y_i's.
- When this second parameter is 'num' then $x_1, x_2, x_3, \ldots$ are specialized to $1, 2, 3, \ldots$
- { IdcaYang($perm$), $perm \in \mathfrak{S}_n$ } is a linear basis of the idCoxeter algebra, as a free module with coefficients in the x_i's.
- This basis is defined by the recursion : for a simple transposition s_k and a permutation $perm$, such that $length(perm\, s_k) > length(perm)$, then IdcaYang($perm\, s_k$) = IdcaYang($perm$) &$* $(1 + (1 - x_j/x_i)\pi_k)$ where $i = perm[k]$ and $j = perm[k+1]$.

- Coefficients in the expansion are specializations of double Grothendieck polynomials.
- Whenever there is a conflict between the function name IdcaYang and another name used in the same session, use the long form IDCA['IdcaYang'].

Examples:

```
> with(IDCA):
> IdcaYang([3,1,2]);

       (x2 - x3) [1, 3, 2]                (- x1 + x3) [2, 1, 3]
       ------------------- + [1, 2, 3] - ---------------------
               x2                                 x1

            (x2 - x3) (- x1 + x3) [3, 1, 2]
          - -------------------------------
                         x2 x1
```

SEE ALSO:

5.5.10 Rd2Idca — calculate a product of generators

Calling Sequence:

Rd2Idca(rd)

Rd2Idca(rd, n)

IDCA[Rd2Idca](rd)

IDCA[Rd2Idca](rd, n)

Parameters:

rd — any list denoting a decomposition

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The Rd2Idca function calculates the product of the generators given by the reduced decomposition rd. The product is taken in the idCoxeter algebra.
- If rd is reduced then Rd2Idca(rd)=SG['Rd2Perm'](rd), and otherwise there is an extra sign.
- When the second argument is not present, n is the smallest possible degree.
- Whenever there is a conflict between the function name Rd2Idca and another name used in the same session, use the long form IDCA['Rd2Idca'].

Examples:

```
> with(IDCA):
> Rd2Idca([1,2,1], 5);

                              [3, 2, 1, 4, 5]
```

```
> Rd2Idca([1,2,1]);

                                [3, 2, 1]

> Rd2Idca([1,2,1,2]);

                              - [3, 2, 1]
```

SEE ALSO:

5.6 Working in the generic Hecke algebra with HEKA

✓

Calling Sequence:

HEKA[<function>]($args$)

<function>($args$)

Synopsis:

- This package provides functions to work with the Hecke algebra of the symmetric group. This algebra is generated by simple operators τ_i's satisfying braid relations and the Hecke relation:

$$\tau_i^2 = (q_1 + q_2)\tau_i - q_1 q_2.$$

- The usual Hecke algebra is obtained by setting $q_1 = q$ and $q_2 = -1$.
- A general element is expressed as a linear combination of permutations with arbitrary coefficients.
- Whenever there is a conflict between a function name in HEKA and another name used in the same session, use the long form HEKA[<function>].
- The available functions are:

HekaAdd	HekaEmbed	HekaMinus	HekaMult
HekaOnPol	HekaOnX	HekaOnXfix	HekaYang

- For help with a particular function do ?HEKA[<function>] or ?HEKA,<function> where <function> is one from the above list.
- Instead of HekaAdd, HekaMinus, HekaMult, one can use in short &%+, &%-, &%*.

Examples:

```
> with(HEKA):
> a:=q + u/t*[3,1,2]:
> b:=t*[2,1,3] - [1,3,2]:
> HekaMult(a, b);

  (q1 u + q2 u) [3, 1, 2] + q t [2, 1, 3] + (- q2 q1 u - q) [1, 3, 2]

          u [3, 2, 1]
        - -----------
               t
```

5.6.1 `HekaEmbed` — reduce each permutation inside an expression to the minimum symmetric group containing it

Calling Sequence:

`HekaEmbed`(e)

`HekaEmbed`(e, m)

`HEKA[HekaEmbed]`(e)

`HEKA[HekaEmbed]`(e, m)

Parameters:

e — any element of the Hecke algebra

m — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `HekaEmbed` function reduces each permutation inside an expression to the minimum symmetric group containing it ($\mathfrak{S}_n$ is contained in $\mathfrak{S}_{n+1}$ by addition of the fixed point $n+1$).
- If a parameter m has been specified, the minimum is choosen to be m.
- Whenever there is a conflict between the function name `HekaEmbed` and another name used in the same session, use the long form `HEKA['HekaEmbed']`.

Examples:

```
> with(HEKA):
> HekaEmbed(a*[3,1,2] - b^2*[2,1,3,4]);

                                   2
                                - b  [2, 1] + a [3, 1, 2]

> HekaEmbed(a*[3,1,2] - b^2*[2,1,3,4], 3);

                                  2
                               - b  [2, 1, 3] + a [3, 1, 2]
```

SEE ALSO:

5.6.2 `HekaAdd` — sum of two elements of the Hecke algebra

Calling Sequence:

`HekaAdd`(e_1, e_2)

`HEKA[HekaAdd]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of the Hecke algebra

Synopsis:

- The `HekaAdd` function adds two elements of the Hecke algebra.
- One can use instead the operator `&%+`.
- Whenever there is a conflict between the function name `HekaAdd` and another name used in the same session, use the long form `HEKA['HekaAdd']`.

Examples:

```
> with(HEKA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> HekaAdd(a, b);

                  u [3, 1, 2]
                  ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                       t

> a &%+ b;

                  u [3, 1, 2]
                  ----------- - [1, 3, 2] + (q + t) [1, 2, 3]
                       t
```

SEE ALSO:

5.6.3 HekaMinus — difference of two elements of the Hecke algebra

Calling Sequence:

`HekaMinus`(e_1, e_2)

`HEKA[HekaMinus]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of the Hecke algebra

Synopsis:

- The `HekaMinus` function computes the difference between two elements of the Hecke algebra.
- One can use instead the operator `&%-`.
- Whenever there is a conflict between the function name `HekaMinus` and another name used in the same session, use the long form `HEKA['HekaMinus']`.

Examples:

```
> with(HEKA):
> a:=q + u/t*[3,1,2]:
> b:=t - [1,3,2]:
> HekaMinus(a, b);
```

```
                 u [3, 1, 2]
                 ----------- + [1, 3, 2] + (q - t) [1, 2, 3]
                      t

> a &%- a;
                                     0
```

SEE ALSO:

5.6.4 HekaMult — multiplication of two elements of the Hecke algebra

Calling Sequence:

`HekaMult`(e_1, e_2)

`HEKA[HekaMult]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of the Hecke algebra

Synopsis:

- The `HekaMult` function multiplies two elements of the Hecke algebra.
- One can use instead the operator `&%*`.
- Whenever there is a conflict between the function name `HekaMult` and another name used in the same session, use the long form `HEKA['HekaMult']`.

Examples:

```
> with(HEKA):
> a:=q + u/t*[3,1,2]:
> b:=t*[2,1,3] - [1,3,2]:
> HekaMult(a, b);

    (q1 u + q2 u) [3, 1, 2] + q t [2, 1, 3] + (- q2 q1 u - q) [1, 3, 2]

            u [3, 2, 1]
          - -----------
                 t

> a &%* b;

    (q1 u + q2 u) [3, 1, 2] + q t [2, 1, 3] + (- q2 q1 u - q) [1, 3, 2]

            u [3, 2, 1]
          - -----------
                 t
```

SEE ALSO:

5.6.5 `HekaOnPol` — action of an element of the algebra on a polynomial

Calling Sequence:

`HekaOnPol`(e_1, exp)

`HekaOnPol`(e_1, exp, v)

`HEKA[HekaOnPol]`(e_1, exp)

`HEKA[HekaOnPol]`(e_1, exp, v)

Parameters:

e_1 — any element of the Hecke algebra of the symmetric group

exp — any expression

v — any (extra) string

Synopsis:

- The `HekaOnPol` function realizes the action of an element of the Hecke algebra of the symmetric group, say e_1, on an expression exp.
- The action of the element `Transpo`(i, n) is:

$$exp \longrightarrow \frac{(exp - exp')(q_2 x_i + q_1 x_{i+1})}{(x_i - x_{i+1})} + q_1\, exp \quad ,$$

  where exp' is the image of exp under `Transpo`(i, n).
- By default the algebra acts on the variables $x_1, x_2, x_3, \ldots$
- When called with a third argument v, being, say y the `HekaOnPol` function acts on the variables $y_1, y_2, y_3, \cdots$
- Whenever there is a conflict between the function name `HekaOnPol` and another name used in the same session, use the long form `HEKA['HekaOnPol']`.

Examples:

```
> with(HEKA):
> HekaOnPol([2,1], 1);
                                            q1

> HekaOnPol([2,1], x1);

                              x2 q1 + x1 q2 + q1 x1

> HekaOnPol(a^2*[3,1,2], x1^2*x2*y1*y2^3);

    2    2            2   2                            2        2
   a   (x2  q1 q2 + x2  q1  + 2 x2 q1 x1 q2 + x2 x1 q2  + x2 q1  x1

                                2                       2         2   2
 + x2 q2 x3 q1 + x2 x3 q1  + x1 q2 x3 q1 + x3 q1  x1 + x3  q1 ) x1 y1 y2
```

```
> HekaOnPol(a^2*[3,1,2], x1^2*x2*y1^2*y2^3, y);

     2    2            2                             2           2      3
  - a   (y2  q2 + q1 y2  + y2 y3 q2 + y2 y3 q1 + y3  q1) q2 x1  x2 y1
```

....

SEE ALSO:

5.6.6 HekaOnX — action of an element of the Hecke algebra of the symmetric group on a linear combination of X_{perm}

✓

Calling Sequence:

`HekaOnX`(e_1, exp)

`HEKA[HekaOnX]`(e_1, exp)

Parameters:

e_1 — any element of the Hecke algebra of the symmetric group

exp — any expression

Synopsis:

- The `HekaOnX` function realizes the action of an element of the Hecke algebra of the symmetric group, say e_1, on an expression exp expressed on the $\mathbf{X}$ Schubert basis.

- The expression exp is expanded and the result is not collected.

- One may add `'noexpand'` just after the argument exp to choose not to expand the expression exp before treating it.

- One may collect the result by adding a third argument: `HekaOnX`$(e_1, exp,$ `'collect'`). Moreover, one can use both `noexpand` and `collect` options:

  `HekaOnX`$(e_1, exp,$ `'noexpand', 'collect'`)

- The action of the element `Transpo`(i,n) on the Schubert polynomial $\mathbf{X}_{perm}$ is : let $\nu = perm\, s_i$ in which s_i stands for `Transpo`(i,n)), then:

$$\begin{array}{lcll} \mathbf{X}_{perm} & \longrightarrow & q_1\mathbf{X}_{perm} & \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & q_1\mathbf{X}_{perm} + (q_2x_i + q_1x_{i+1})\mathbf{X}_\nu & \text{otherwise.} \end{array}$$

- The result is expressed on the $\mathbf{X}$ Schubert basis and is not collected.

- Whenever there is a conflict between the function name `HekaOnX` and another name used in the same session, use the long form `HEKA['HekaOnX']`.

Examples:

```
> with(HEKA):
> HekaOnX(a^4 * [1,3,2] - a^3 * [2,1], z * X[3,2,1] - X[2,1,3]);

                      4                        4                      3
    q2 X[3, 2, 1] z a  + q1 X[3, 1, 4, 2] z a  - q2 X[3, 2, 1] z a

                          3    4                  3                  3
  - q1 X[2, 4, 1, 3] z a  - a  q1 + q2 X[2, 1] a  + q1 X[1, 3, 2] a
```

SEE ALSO: SP

5.6.7 HekaOnXfix — action of an element of the Hecke algebra of the symmetric group on the free module

Calling Sequence:

HekaOnXfix(e_1, exp)

HEKA[HekaOnXfix](e_1, exp)

Parameters:

e_1 — any element of the Hecke algebra of the symmetric group

exp — any expression

Synopsis:

- The HekaOnXfix function realizes the action of an element of the Hecke algebra of the symmetric group, say e_1, on an expression exp expressed on the **X** Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.
- The expression exp is expanded and the result is not collected.
- One may add 'noexpand' just after the argument exp to choose not to expand the expression exp before treating it.
- One may collect the result by adding a third argument: HekaOnXfix(e_1, exp, 'collect'). Moreover, one can use both noexpand and collect options: HekaOnXfix(e_1, exp, 'noexpand', 'collect').
- The action of the element π_i on the Schubert polynomial $\mathbf{X}_{perm}$ is : let $\nu = perm\, s_i$ where s_i=Transpo(i, n), then:

$$\begin{array}{lll} \mathbf{X}_{perm} & \longrightarrow & q_1\mathbf{X}_{perm} \qquad \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & q_1\mathbf{X}_{perm} + (q_2x_i + q_1x_{i+1})\mathbf{X}_\nu \quad \text{otherwise.} \end{array}$$

- The result is expressed on the **X** Schubert basis corresponding to a given symmetric group and is not collected.
- Whenever there is a conflict between the function name HekaOnXfix and another name used in the same session, use the long form HEKA['HekaOnXfix'].

Examples:

```
> with(HEKA):
> _FMn;

                                       4

> HekaOnXfix(q*[1,3,2], s[1,1]*X[2,4,1,3], collect);

  q2 X[2, 4, 1, 3] s[1, 1] q + (s[2, 1] + s[1, 1, 1]) q1 q X[2, 1, 4, 3]

     + q2 X[2, 3, 4, 1] s[1, 1] q - q1 X[4, 1, 2, 3] s[1, 1] q

     - (s[2, 2] + s[2, 1, 1] + s[1, 1, 1, 1]) q1 q X[2, 1, 3, 4]
```

SEE ALSO: FM, SP

5.6.8 HekaYang — compute a special element of the Hecke algebra

✓

Calling Sequence:

HekaYang(*perm*)

HEKA[HekaYang](*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The HekaYang function calculates a special element of the Hecke algebra of the symmetric group.
- When called with a second parameter, say $'y'$, one specifies that coefficients are in the y_i's.
- When this second parameter is 'num' then $x_1, x_2, x_3, \ldots$ are specialized to $1, 2, 3, \ldots$
- { HekaYang(*perm*), $perm \in \mathfrak{S}_n$ } is a linear basis of the Hecke algebra of the symmetric group, as a free module with coefficients in the x_i's, q_1, and q_2.
- This basis is defined by the recursion : for a simple transposition s_k and a permutation *perm*, such that length$(perm\, s_k) >$ length$(perm)$, then HekaYang$(perm\, s_k)$ = HekaYang$(perm)$ &%* $(1 + (x_j - x_i - 1)/(q_1 + q_2)\, \tau_k)$ where $i = perm[k]$ and $j = perm[k+1]$.
- Whenever there is a conflict between the function name HekaYang and another name used in the same session, use the long form HEKA['HekaYang'].

Examples:

```
> with(HEKA):
> HekaYang([3,1,2]);

          (- x3 + x2) [1, 3, 2]                 (x3 - x1) [2, 1, 3]
        - --------------------- + [1, 2, 3] + -------------------
               x2 (q1 + q2)                       x1 (q1 + q2)
```

```
                2
    (- x3  + x3 x1 + x3 x2 - x1 x2) [3, 1, 2]
  - -----------------------------------------
                                 2
               x1 (q1 + q2)  x2
```

SEE ALSO:

5.7 CG: a package about classical Lie groups

Calling Sequence:

CG[<function>]$(args)$

<function>$(args)$

Synopsis:

- This package provides functions to work with classical Lie groups.
- The available functions are:

DimRepO	DimRepSp	Freudenthal	Frob2Part
LRrO	LRrSp	O2h	O2s
OJtMatrix	Part2Frob	PositiveRoots	Rho
Sp2h	Sp2s	SpJtMatrix	Sseries
ToO	ToSp	s2O	s2Sp

- For help with a particular function do either ?CG[<function>] or ?CG,<function> where <function> is one from the above list.

Examples:

```
> with(CG):
> s2O(s[4, 3, 2]);

  O[4, 3, 2] + O[3, 2, 2] + O[4, 2, 1] + O[3, 3, 1] + O[4, 3] + O[4, 1]

  + O[2, 2, 1] + 2 O[3, 2] + O[3, 1, 1] + 2 O[2, 1] + O[3] + O[1]
```

5.7.1 DimRepO — calculate the dimension of an irreducible representation of the orthogonal group

Calling Sequence:

DimRepO$(rk, lambda)$

CG[DimRepO]$(rk, lambda)$

Parameters:

rk — the rank of O

lambda — any partition

Synopsis:

- The `DimRepO` function calculates the dimension of the irreducible representation of the orthogonal group, with highest weight represented by a partition λ.
- Whenever there is a conflict between the function name `DimRepO` and another name used in the same session, use the long form `CG['DimRepO']`.

Examples:

```
> with(CG):
> DimRepO(6, [4,2,2]);
```

720

SEE ALSO: `DimRepSp`

5.7.2 `DimRepSp` — calculate the dimension of an irreducible representation of the symplectic group

Calling Sequence:

`DimRepSp`(*lambda*)

`CG[DimRepSp]`(*lambda*)

Parameters:

lambda — any list denoting a partition

Synopsis:

- The `DimRepSp` function calculates the dimension of the irreducible representation of the symplectic group, with highest weight represented by a partition λ.
- Whenever there is a conflict between the function name `DimRepSp` and another name used in the same session, use the long form `CG['DimRepSp']`.

Examples:

```
> with(CG):
> DimRepSp([4, 2, 2]);
```

1078

SEE ALSO: `DimRepO`

5.7.3 Frob2Part — from Frobenius notation to standard notation for partitions

Calling Sequence:

Frob2Part($frob$)

CG[Frob2Part]($frob$)

Parameters:

$frob$ — any list of lists denoting a partition in Frobenius

Synopsis:

- The Frob2Part function computes a partition from its Frobenius symbol.
- The result is a list denoting a partition.
- Whenever there is a conflict between the function name Frob2Part and another name used in the same session, use the long form CG['Frob2Part'].

Examples:

```
> with(CG):
> Frob2Part([[6,3,0], [4,2,1]]);

                    [7, 5, 3, 3, 1]
```

SEE ALSO: Part2Frob

5.7.4 Freudenthal — calculate weight multiplicities

Calling Sequence:

Freudenthal($typ, lambda, mu$)

CG[Freudenthal]($typ, lambda, mu$)

Parameters:

typ — one of the classical group A, B, C or D

$lambda, mu$ — any lists denoting partitions

Synopsis:

- The Freudenthal function calculates the multiplicity $K(\lambda, \mu)$ of the weight μ in the irreducible module (of the simple Lie algebra of type typ) with highest weight λ, using the Freudenthal formula.
- For type A, $K(\lambda, \mu)$ is the corresponding Kostka number.
- Whenever there is a conflict between the function name Freudenthal and another name used in the same session,use the long form CG['Freudenthal'].

Examples:

```
> with(CG):
> Freudenthal(A, [9,1], [5,3,2]);

                         2
```

SEE ALSO: DimRepO, DimRepSp

5.7.5 LRrO — Littlewood-Richardson rule for orthogonal Schur functions

Calling Sequence:

`LRrO`(O_1, O_2)

`CG[LRrO]`(O_1, O_2)

Parameters:

O_1, O_2 — any orthogonal Schur functions

Synopsis:

- The `LRrO` function computes the product of the two orthogonal Schur functions O_1 and O_2 using the Newell-Littlewood rule.
- The result is expressed on the basis of orthogonal Schur functions.
- Whenever there is a conflict between the function name `LRrO` and another name used in the same session, use the long form `CG['LRrO']`.

Examples:

```
> with(CG):
> LRrO(O[3,2,1], O[1,1]);

    O[3, 2, 1, 1, 1] + O[3, 2, 2, 1] + O[3, 3, 1, 1] + O[3, 3, 2]

  + O[4, 2, 1, 1] + O[4, 2, 2] + o[4, 3, 1] + O[4, 1, 1] + 3 O[3, 2, 1]

  + O[3, 1, 1, 1] + O[3, 3] + O[4, 2] + O[2, 2, 1, 1] + O[2, 2, 2]

  + O[3, 1] + O[2, 2] + O[2, 1, 1]
```

SEE ALSO: LRrSp, LRrSf

5.7.6 LRrSp — Littlewood-Richardson rule for symplectic Schur functions

Calling Sequence:

`LRrSp`(Sp_1, Sp_2)

`CG[LRrSp]`(Sp_1, Sp_2)

Parameters:

Sp_1, Sp_2 — any symplectic Schur functions

Synopsis:

- The `LRrSp` function computes the product of the two symplectic Schur functions Sp_1 and Sp_2 using the Newell-Littlewood rule.
- The result is expressed on the basis of symplectic Schur functions.
- Whenever there is a conflict between the function name `LRrSp` and another name used in the same session, use the long form `CG['LRrSp']`.

Examples:

```
> with(CG):
> LRrSp(Sp[3,2,1], Sp[1,1]);

    Sp[3, 2, 1, 1, 1] + Sp[3, 2, 2, 1] + Sp[3, 3, 1, 1] + Sp[3, 3, 2]

  + Sp[4, 2, 1, 1] + Sp[4, 2, 2] + Sp[4, 3, 1] + Sp[4, 1, 1]

  + 3 Sp[3, 2, 1] + Sp[3, 1, 1, 1] + Sp[3, 3] + Sp[4, 2]

  + Sp[2, 2, 1, 1] + Sp[2, 2, 2] + Sp[3, 1] + Sp[2, 2] + Sp[2, 1, 1]
```

SEE ALSO: LRrO, LRrSf

5.7.7 O2h — convert a monomial in the orthogonal Schur functions to the h-basis

Calling Sequence:

O2h(sf)

CG[O2h](sf)

Parameters:

sf — any product of orthogonal Schur functions

Synopsis:

- The O2h function expresses any product of orthogonal Schur functions in the h-basis using a Jacobi-Trudi-type matrix.
- Whenever there is a conflict between the function name O2h and another name used in the same session, use the long form CG['O2h'].

Examples:

```
> with(CG):
> O2h(O[3, 3, 1]);

   2                    2
 h3  h1 - h3 h4 - h3 h1  + 2 h4 h1 + h1 h2 - h1 - h2 h4 h1 + h5 h2 - h5
```

SEE ALSO: O2s, OJtMatrix

5.7.8 O2s — convert a monomial in the orthogonal Schur basis to the Schur basis

Calling Sequence:

O2s(sf)

CG[O2s](sf)

Parameters:

sf — any product of orthogonal Schur functions

Synopsis:

- The `O2s` function converts a monomial in the orthogonal Schur functions to an element expressed in the Schur basis using a Jacobi-Trudi-type matrix.
- The result is a linear combination of Schur functions.
- Whenever there is a conflict between the function name `O2s` and another name used in the same session, use the long form `CG['O2s']`.

Examples:

```
> with(CG):
> O2s(O[7,3]);

 s[7, 3] - s[5, 3] - s[6, 2] - s[7, 1] + s[4, 2] + s[5, 1] + s[6] - s[4]
```

SEE ALSO: `s2O, OJtMatrix`

5.7.9 `OJtMatrix` — Jacobi-Trudi-type matrix

Calling Sequence:

`OJtMatrix`(*part*)

`CG[OJtMatrix]`(*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The `OJtMatrix` function computes the Jacobi-Trudi-type matrix which corresponds to the partition *part*. Its determinant gives the expression of the orthogonal Schur function O_{part} in the e-basis.
- Whenever there is a conflict between the function name `OJtMatrix` and another name used in the same session, use the long form `CG['OJtMatrix']`.

Examples:

```
> with(CG):
> OJtMatrix([7, 4, 3]);

                    [ h7 - h5  h8 - h4  h9 - h7 + h5 - h3 ]
                    [                                     ]
                    [ h3 - h1   h4 - 1     h5 - h3 + h1   ]
                    [                                     ]
                    [    h1       h2          h3 - h1     ]
```

SEE ALSO: `O2s, s2O`

5.7.10 `Part2Frob` — from standard partition notation to Frobenius notation

Calling Sequence:

`Part2Frob`(*part*)

`CG[Part2Frob]`(*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The `Part2Frob` function converts any partition into Frobenius notation.
- The result is a list of two lists.
- Whenever there is a conflict between the function name `Part2Frob` and another name used in the same session, use the long form `CG['Part2Frob']`.

Examples:

```
> with(CG):
> Part2Frob([7, 5, 3, 3, 1]);

                    [[6, 3, 0], [4, 2, 1]]
```

SEE ALSO: `Frob2Part`

5.7.11 `PositiveRoots` — positive roots of a classical group

Calling Sequence:

`PositiveRoots`($typ, rank$)

`CG[PositiveRoots]`($typ, rank$)

Parameters:

typ — one of the classical group A, B, C or D

rank — the rank of the group

Synopsis:

- The `PositiveRoots` function computes the positive roots of any classical group.
- The result is a list of lists.
- Whenever there is a conflict between the function name `PositiveRoots` and another name used in the same session, use the long form `CG['PositiveRoots']`.

Examples:

```
> with(CG):
> PositiveRoots(D, 4);

 [[1, -1, 0, 0], [1, 1, 0, 0], [1, 0, -1, 0], [1, 0, 1, 0], [1, 0, 0, -1],

  [1, 0, 0, 1],  [0, 1, -1, 0], [0, 1, 1, 0], [0, 1, 0, -1], [0, 1, 0, 1],

  [0, 0, 1, -1], [0, 0, 1, 1]]
```

SEE ALSO: `Freudenthal`

5.7.12 Rho — half the sum of positive roots

Calling Sequence:

Rho$(typ, rank)$

CG[Rho]$(typ, rank)$

Parameters:

typ — one of the classical group A, B, C or D

rank — the rank of the group

Synopsis:

- The Rho function computes half the sum of positive roots.
- The result is a list.
- Whenever there is a conflict between the function name Rho and another name used in the same session, use the long form CG['Rho'].

Examples:

```
> with(CG):
> Rho(B, 4);

                    [1/2, 3/2, 5/2, 7/2]
```

SEE ALSO: `Freudenthal, PositiveRoots`

5.7.13 Sp2h — convert a monomial in the symplectic Schur functions to the h-basis

Calling Sequence:

Sp2h(sf)

CG[Sp2h](sf)

Parameters:

sf — any product of symplectic Schur functions

Synopsis:

- The Sp2h function expresses any product of symplectic Schur functions in the h-basis using a Jacobi-Trudi-type matrix.
- Whenever there is a conflict between the function name Sp2h and another name used in the same session, use the long form CG['Sp2h'].

Examples:

```
> with(CG):
> Sp2h(Sp[3, 3, 1]);

        2        2                                 2
    h1 h3  + h3 h1  - h3 h4 - h3 - h2 h1 h4 - h1 h2  + h2 h5 + h1 h2
```

SEE ALSO: Sp2s, SpJtMatrix

5.7.14 Sp2s — convert a monomial in the symplectic Schur functions to the Schur basis

Calling Sequence:

Sp2s(sf)

CG[Sp2s](sf)

Parameters:

sf — any product of symplectic Schur functions

Synopsis:

- The Sp2s function expresses any product of symplectic Schur functions in the Schur basis using a Jacobi-Trudi-type matrix.
- The result is a linear combination of Schur functions.
- Whenever there is a conflict between the function name Sp2s and another name used in the same session, use the long form CG['Sp2s'].

Examples:

```
> with(CG):
> Sp2s(Sp[7, 4, 3]);

  - s[7, 3, 2] + s[6, 2, 2] - s[6, 3, 3] - s[6, 4, 2] - s[5, 2, 1]

  + s[6, 3, 1] + s[7, 4, 3] + s[5, 3, 2]
```

SEE ALSO: s2Sp, SpJtMatrix

5.7.15 SpJtMatrix — Jacobi-Trudi-type matrix

Calling Sequence:

`SpJtMatrix`(*part*)

`CG[SpJtMatrix]`(*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The `SpJtMatrix` function computes the Jacobi-Trudi-type matrix which corresponds to the partition *part*. Its determinant gives the expansion of the symplectic Schur function Sp_{part} in the e-basis.

- Whenever there is a conflict between the function name `SpJtMatrix` and another name used in the same session, use the long form `CG['SpJtMatrix']`.

Examples:

```
> with(CG):
> SpJtMatrix([7, 4, 3]);

                        [ h7  h8 + h6  h9 + h5 ]
                        [                      ]
                        [ h3  h4 + h2  h5 + h1 ]
                        [                      ]
                        [ h1   h2 + 1     h3   ]
```

SEE ALSO: `Sp2s, s2Sp`

5.7.16 Sseries — compute the truncated Schur function series

Calling Sequence:

`Sseries`(*typ*, *rank*)

`CG[Sseries]`(*typ*, *rank*)

Parameters:

typ — the type of the Schur function series

rank — the rank of truncature

Synopsis:

- The `Sseries` function computes the Schur function series of a given type trucated to the given *rank*. The available types are A, B, C and D, according to the notations of King and Wybourne.

- The result is a linear combination of Schur functions.

- Whenever there is a conflict between the function name `Sseries` and another name used in the same session, use the long form `CG['Sseries']`.

Examples:

```
> with(CG):
> Sseries(A, 10);

    1 - s[1, 1] + s[2, 1, 1] - s[3, 1, 1, 1] + s[4, 1, 1, 1, 1]

    - s[3, 3, 2, 2] - s[5, 1, 1, 1, 1, 1] - s[2, 2, 2] + s[3, 2, 2, 1]

    - s[4, 2, 2, 1, 1]
```

SEE ALSO: `Part2Frob`

5.7.17 `ToO` — express any symmetric function in the basis of orthogonal Schur functions ✘

Calling Sequence:

`ToO`(sf)

`ToO`(sf, b)

`CG[ToO]`(sf)

`CG[ToO]`(sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The `ToO` function expresses any symmetric function in the basis of orthogonal Schur functions, *i.e.* as a linear combination of orthogonal Schur functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- One may add `'noexpand'` just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: `ToO`(sf, b, `'collect'`). For instance, `ToO`(sf, `'O'`, `'collect'`) may be used to collect the argument sf.
- Whenever there is a conflict between the function name `ToO` and another name used in the same session, use the long form `CG['ToO']`.

Examples:

```
> with(CG):
> ToO(h4+2*e2*s[3,1]);

     3 O[4] + 7 O[2] + 3 O[] + 2 O[3, 1, 1, 1] + 2 O[1, 1, 1, 1]

   + 6 O[2, 1, 1] + 2 O[3, 2, 1] + 4 O[2, 2] + 6 O[3, 1] + 4 O[1, 1]

   + 2 O[4, 2] + 2 O[4, 1, 1]
```

SEE ALSO: `ToSp`

5.7.18 `ToSp` — express any symmetric function in the basis of symplectic Schur functions

✘

Calling Sequence:

`ToSp`(sf)

`ToSp`(sf, b)

`CG[ToSp]`(sf)

`CG[ToSp]`(sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The `ToSp` function expresses any symmetric function in the basis of symplectic Schur functions, *i.e.* as a linear combination of symplectic Schur functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- One may add `'noexpand'` just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: `ToSp`($sf, b,$ `'collect'`). For instance, `ToSp`(sf, `'Sp', 'collect'`) may be used to collect the argument sf.
- Whenever there is a conflict between the function name `ToSp` and another name used in the same session, use the long form `CG['ToSp']`.

Examples:

```
> with(CG):
> ToSp(e3*s[3,2]);

   2 Sp[2, 2, 1, 1] + Sp[2, 1, 1, 1, 1] + 2 Sp[3, 1, 1, 1] + 4 Sp[3, 2, 1]

 + 5 Sp[2, 1, 1] + 4 Sp[2, 2] + 4 Sp[3, 1] + 2 Sp[1, 1, 1, 1] + 4 Sp[1, 1]

 + 3 Sp[2] + Sp[3, 2, 1, 1, 1] + 2 Sp[3, 3] + Sp[] + Sp[3, 3, 1, 1]

 + Sp[4, 1, 1] + 2 Sp[4, 2] + Sp[4, 2, 1, 1] + Sp[4, 3, 1]
```

SEE ALSO: `ToO`

5.7.19 s2O — convert a monomial in the Schur functions to the orthogonal Schur basis ✘

Calling Sequence:

`s2O`(sf)

`CG[s2O]`(sf)

Parameters:

sf — any product of Schur functions

Synopsis:

- The `s2O` function converts any product of Schur functions into the orthogonal Schur basis.
- The result is a linear combination of orthogonal Schur functions.
- Whenever there is a conflict between the function name `s2O` and another name used in the same session, use the long form `CG['s2O']`.

Examples:

```
> with(CG):
> s2O(s[3,2,1]);

      O[3, 2, 1] + O[2, 1, 1] + O[3, 1] + O[2, 2] + O[1, 1] + O[2]
```

SEE ALSO: `O2s`

5.7.20 s2Sp — convert a monomial in the Schur functions to the symplectic Schur basis ✘

Calling Sequence:

`s2Sp`(sf)

`CG[s2Sp]`(sf)

Parameters:

sf — any product of Schur functions

Synopsis:

- The `s2Sp` function converts any product of Schur functions into the symplectic Schur basis.
- The result is a linear combination of symplectic Schur functions.
- Whenever there is a conflict between the function name `s2Sp` and another name used in the same session, use the long form `CG['s2Sp']`.

Examples:

```
> with(CG):
> s2Sp(s[4,3,1]);

   Sp[4, 3, 1] + Sp[1, 1] + Sp[2, 2] + Sp[3, 1] + 2 Sp[2] + Sp[3, 2, 1]

    + Sp[4, 2] + Sp[3, 3] + Sp[2, 1, 1]
```

SEE ALSO: `Sp2s`

5.8 `BN` and the hyperoctahedral groups

Calling Sequence:

`BN[<function>]`(*args*)

`<function>`(*args*)

Synopsis:

- This package provides functions to work with the hyperoctahedral group, in particular with its Bruhat order. It gives functions to handle the elements of this group, represented by codes. It also provides functions to make conversions between a code and a signed permutation (`Bar`) or a palindromic permutation (`Palin`). This last coding comes from the embedding of B_n into $\mathfrak{S}_{2n}$.
- Whenever there is a conflict between a function name in `BN` and another name used in the same session, use the long form `BN[<function>]`.
- The available functions are:

BnBar2Code	BnBar2Mat	BnCode2Bar	BnCode2Length
BnCode2ListRd	BnCode2NbRd	BnCode2Palin	BnCode2Rd
BnFirstCode	BnInterval	BnInvCode	BnLastCode
BnListCode	BnMat2Bar	BnMultBar	BnMultCode
BnNextCode	BnPalin2Code	BnPrevCode	BnRd2Code
BnTranspoPalin			

- For help with a particular function do either `?BN[<function>]` or `?BN,<function>` where `<function>` is one from the above list.

Examples:

```
> with(BN):
> BnInterval([1,1,3]);
```

```
  [[1, 1, 3], [1, 1, 2], [1, 2, 1], [1, 1, 1], [1, 0, 3], [1, 0, 2],
   [1, 2, 0], [1, 1, 0], [0, 1, 3], [0, 1, 2], [0, 2, 1], [0, 1, 1],
   [0, 0, 3], [0, 0, 2], [1, 0, 1], [0, 0, 1], [0, 2, 0], [0, 1, 0],
   [1, 0, 0], [0, 0, 0]]
```

5.8.1 BnBar2Code — code of a signed permutation ✘

Calling Sequence:

`BnBar2Code`(*bar*)

`BN[BnBar2Code]`(*bar*)

Parameters:

bar — any list denoting a signed permutation

Synopsis:

- The `BnBar2Code` function calculates the code of a signed permutation *bar*.
- Whenever there is a conflict between the function name `BnBar2Code` and another name used in the same session, use the long form `BN['BnBar2Code']`.

Examples:

```
> with(BN):
> BnBar2Code([-3, 2, -1]);
```

```
                    [1, 1, 3]
```

See Also: `BnCode2Bar`, `TYP[IsBnCode]`

5.8.2 BnBar2Mat — signed permutation matrix from a signed permutation ✘

Calling Sequence:

`BnBar2Mat`(*bar*)

`BN[BnBar2Mat]`(*bar*)

Parameters:

bar — any list denoting a signed permutation

Synopsis:

- The `BnBar2Mat` function calculates the matrix corresponding to the signed permutation *bar*. This matrix has entries either 1 or -1 in position $(i, |\texttt{bar}[i]|)$ and 0 otherwise. The sign of the entry in the i-th row, is the sign of $\texttt{bar}[i]$.

- Whenever there is a conflict between the function name `BnBar2Mat` and another name used in the same session, use the long form `BN['BnBar2Mat']`.

Examples:

```
> with(BN):
> BnBar2Mat([-3, 2, -1]);

                              [  0  0  -1 ]
                              [           ]
                              [  0  1   0 ]
                              [           ]
                              [ -1  0   0 ]
```

SEE ALSO: `BnMat2Bar`

5.8.3 `BnCode2Length` — length of a code

Calling Sequence:

`BnCode2Length`(*code*)

`BN[BnCode2Length]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The `BnCode2Length` function calculates the length of an element of the group given by its code. It is equal to the minimal length of an expression of the group element as a product of simple reflections. It is also the sum of the components of the code.

- Whenever there is a conflict between the function name `BnCode2Length` and another name used in the same session, use the long form `BN['BnCode2Length']`.

Examples:

```
> with(BN):
> BnCode2Length([0,1,0,5,4]);

                                   10
```

SEE ALSO: `BnCode2Rd`

5.8.4 BnCode2Bar — signed permutation corresponding to a code

Calling Sequence:

BnCode2Bar(*code*)

BN[BnCode2Bar](*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The BnCode2Bar function calculates the signed permutation that corresponds to a code.
- Whenever there is a conflict between the function name BnCode2Bar and another name used in the same session, use the long form BN['BnCode2Bar'].

Examples:

```
> with(BN):
> BnCode2Bar([1, 1, 3]);
```

```
[-3, 2, -1]
```

SEE ALSO: BnBar2Code, TYP[IsBnBar]

5.8.5 BnCode2ListRd — list of all reduced decompositions of a code

Calling Sequence:

BnCode2ListRd(*code*)

BN[BnCode2ListRd](*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The BnCode2ListRd function computes the list of all reduced decompositions of a given code.
- Whenever there is a conflict between the function name BnCode2ListRd and another name used in the same session, use the long form BN['BnCode2ListRd'].

Examples:

```
> with(BN):
> BnCode2ListRd([1, 1, 5]);
```

```
[[2, 1, 0, 1, 2, 0, 1], [2, 1, 0, 1, 0, 2, 1], [2, 0, 1, 0, 1, 2, 1],
 [0, 2, 1, 0, 1, 2, 1], [0, 1, 2, 1, 0, 1, 2], [2, 0, 1, 2, 0, 1, 2],
 [0, 2, 1, 2, 0, 1, 2], [2, 0, 1, 0, 2, 1, 2], [0, 2, 1, 0, 2, 1, 2]]
```

SEE ALSO: BnCode2NbRd, BnCode2Rd, BnRd2Code

5.8.6 BnCode2NbRd — number of all reduced decompositions of a code

Calling Sequence:

`BnCode2NbRd`(*code*)

`BN[BnCode2NbRd]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The `BnCode2NbRd` function computes the number of all reduced decompositions of a given code. It does not generate all reduced decompositions but only counts their number.
- Whenever there is a conflict between the function name `BnCode2NbRd` and another name used in the same session, use the long form `BN['BnCode2NbRd']`.

Examples:

```
> with(BN):
> BnCode2NbRd([1, 1, 5]);

                              9
```

SEE ALSO: `BnCode2ListRd, BnCode2Rd, BnRd2Code`

5.8.7 BnCode2Palin — palindromic permutation corresponding to a code

Calling Sequence:

`BnCode2Palin`(*code*)

`BN[BnCode2Palin]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The `BnCode2Palin` function calculates the palindromic permutation w corresponding to a code. This permutation belongs to the symmetric group of degree $2n$, where n is the degree of the hyperoctahedral group B_n, and satisfies the symmetries:

$$w[i] + w[2n - i + 1] = 2n + 1, \qquad i = 1, \ldots, n \quad .$$

- Whenever there is a conflict between the function name `BnCode2Palin` and another name used in the same session, use the long form `BN['BnCode2Palin']`.

Examples:

```
> with(BN):
> BnCode2Palin([1, 1, 3]);

                    [4, 2, 6, 1, 5, 3]
```

SEE ALSO: `BnPalin2Code, TYP[IsBnPalin]`

5.8.8 BnCode2Rd — canonical reduced decomposition of a code

✘

Calling Sequence:

BnCode2Rd(*code*)

BN[BnCode2Rd](*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The BnCode2Rd function computes the canonical reduced decompositions of a given code.
- Whenever there is a conflict between the function name BnCode2Rd and another name used in the same session, use the long form BN['BnCodeRd'].

Examples:

```
> with(BN):
> BnCode2Rd([1, 1, 5]);
```

```
[0, 1, 2, 1, 0, 1, 2]
```

SEE ALSO: BnCode2ListRd, BnRd2Code

5.8.9 BnFirstCode — the first code [0, 0, ..., 0]

✘

Calling Sequence:

BnFirstCode(n)

BN[BnFirstCode](n)

Parameters:

n — any positive integer

Synopsis:

- The BnFirstCode function gives the first code according to the lexicographic order (or the Bruhat order).
- Whenever there is a conflict between the function name BnFirstCode and another name used in the same session, use the long form BN['BnFirstCode'].

Examples:

```
> with(BN):
> BnFirstCode(6);
```

```
[0, 0, 0, 0, 0, 0]
```

SEE ALSO: BnLastCode, BnNextCode, BnPrevCode

5.8.10 `BnInterval` — list of codes of all group elements below a given one

Calling Sequence:

`BnInterval`(*code*)

`BN[BnInterval]`(*code*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `BnInterval` function computes the list of all codes of group elements that are smaller than the given code, according to the Bruhat order.
- Whenever there is a conflict between the function name `BnInterval` and another name used in the same session, use the long form `BN['BnInterval']`.

Examples:

```
> with(BN):
> BnInterval([1, 0, 3]);

  [[1, 2, 1], [1, 1, 1], [1, 2, 0], [1, 1, 0], [0, 2, 1], [0, 1, 1],

   [0, 2, 0], [0, 1, 0], [1, 0, 1], [0, 0, 1], [1, 0, 0], [0, 0, 0]]
```

SEE ALSO:

5.8.11 `BnInvCode` — code of the inverse of a group element

Calling Sequence:

`BnInvCode`(*code*)

`BN[BnInvCode]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The `BnInvCode` function returns the code of the inverse of the group element given by its code.
- Whenever there is a conflict between the function name `BnInvCode` and another name used in the same session, use the long form `BN['BnInvCode']`.

Examples:

```
> with(BN):
> BnInvCode([1, 0, 3, 3, 1, 5]);

                               [0, 1, 3, 4, 1, 4]
> BnInvCode(");

                               [1, 0, 3, 3, 1, 5]
```

SEE ALSO:

5.8.12 BnLastCode — the last code [1, 3, 5, ...]

✘

Calling Sequence:

BnLastCode(n)

BN[BnLastCode](n)

Parameters:

n — any positive integer

Synopsis:

- The BnLastCode function gives the last code according to the lexicographic order (or the Bruhat order).
- Whenever there is a conflict between the function name BnLastCode and another name used in the same session, use the long form BN['BnLastCode'].

Examples:

```
> with(BN):
> BnLastCode(6);
```

```
[1, 3, 5, 7, 9, 11]
```

SEE ALSO: BnFirstCode, BnNextCode, BnPrevCode

5.8.13 BnListCode — generate all codes

✘

Calling Sequence:

BnListCode(n)

BN[BnListCode](n)

Parameters:

n — any positive integer

Synopsis:

- The BnListCode function generates all the $n!\,2^n$ codes of the hyperoctahedral group B_n.
- Whenever there is a conflict between the function name BnListCode and another name used in the same session, use the long form BN['BnListCode'].

Examples:

```
> with(BN):
> BnListCode(2);
```

```
[[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 1], [1, 2], [1, 3]]
```

SEE ALSO: TYP[IsBnCode]

5.8.14 BnMat2Bar — calculate the signed permutation from a given permutation matrix

Calling Sequence:

BnMat2Bar(mat)

BN[BnMat2Bar](mat)

Parameters:

mat — any signed permutation matrix

Synopsis:

- The BnMat2Bar function calculates the signed permutation corresponding to a given permutation matrix. This matrix has entries either 1 or -1 in position $(i, |\texttt{bar}[i]|)$ and 0 otherwise. The sign of the entry in the i-th row, is the sign of $\texttt{bar}[i]$.
- Whenever there is a conflict between the function name BnMat2Bar and another name used in the same session, use the long form BN['BnMat2Bar'].

Examples:

```
> with(BN):
> BnBar2Mat([-3, 2, -1]);

                    [  0  0  -1 ]
                    [           ]
                    [  0  1   0 ]
                    [           ]
                    [ -1  0   0 ]

> BnMat2Bar(");

                    [-3, 2, -1]
```

SEE ALSO: BnBar2Mat

5.8.15 BnMultBar — multiply signed permutations

Calling Sequence:

BnMultBar(bar_1, bar_2)

BN[BnMultBar](bar_1, bar_2)

Parameters:

bar_1, bar_2 — any lists denoting signed permutations

Synopsis:

- The BnMultBar function calculates the product of a sequence (of arbitrary length) of signed permutations.
- BnMultBar(bar_1, bar_2) computes $bar_1(bar_2)$.
- Signed permutations need not to belong to the same hyperoctahedral group.

- Whenever there is a conflict between the function name `BnMultBar` and another name used in the same session, use the long form `BN['BnMultBar']`.

Examples:

```
> with(BN):
> BnMultBar([-1, 3, -2], [-2, 3, -1], [1, 3, 2, -4]);

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

See Also:

5.8.16 `BnMultCode` — multiply codes

✘

Calling Sequence:

`BnMultCode`($code_1, code_2$)

`BN[BnMultCode]`($code_1, code_2$)

Parameters:

$code_1, code_2$ — any lists denoting codes

Synopsis:

- The `BnMultCode` function computes the product of a sequence (of arbitrary length) of codes.
- `BnMultCode`($code_1, code_2$) computes $code_1(code_2)$.
- Codes need not to belong to the same group.
- Whenever there is a conflict between the function name `BnMultCode` and another name used in the same session, use the long form `BN['BnMultCode']`.

Examples:

```
> with(BN):
> BnMultCode([1, 3, 1], [1, 2, 1], [0, 0, 1, 7]);

                              [0, 3, 3, 7]
```

See Also:

5.8.17 `BnNextCode` — next code in lexicographic ordering

✘

Calling Sequence:

`BnNextCode`(*code*)

`BN[BnNextCode]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The `BnNextCode` function gives the next code according to the lexicographic order. The function returns the `NULL` value when the argument is the last code $[1, 3, 5, \ldots]$.
- Whenever there is a conflict between the function name `BnNextCode` and another name used in the same session, use the long form `BN['BnNextCode']`.

Examples:

```
> with(BN):
> BnNextCode([1, 0, 4, 7, 9]);

                    [1, 0, 5, 0, 0]
```

SEE ALSO: `BnFirstCode`, `BnLastCode`, `BnPrevCode`

✘ 5.8.18 `BnPalin2Code` — code corresponding to a palindromic permutation

Calling Sequence:

`BnPalin2Code`(*palin*)

`BN[BnPalin2Code]`(*palin*)

Parameters:

palin — any list denoting a palindromic permutation

Synopsis:

- The `BnPalin2Code` function calculates the code corresponding to a palindromic permutation. The code belongs to B_n if the permutation belongs to the symmetric group of degree $2n$.
- Whenever there is a conflict between the function name `BnPalin2Code` and another name used in the same session, use the long form `BN['BnPalin2Code']`.

Examples:

```
> with(BN):
> BnPalin2Code([4, 2, 6, 1, 5, 3]);

                    [1, 1, 3]
```

SEE ALSO: `BnCode2Palin`, `TYP[IsBnCode]`

✘ 5.8.19 `BnPrevCode` — previous code in lexicographic ordering

Calling Sequence:

`BnPrevCode`(*code*)

`BN[BnPrevCode]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The `BnPrevCode` function gives the previous code according to the lexicographic order. The function returns the `NULL` value when the argument is the first code $[0, 0, 0, \ldots]$.
- Whenever there is a conflict between the function name `BnPrevCode` and another name used in the same session, use the long form `BN['BnPrevCode']`.

Examples:

```
> with(BN):
> BnPrevCode([1, 0, 5, 0, 0]);

                              [1, 0, 4, 7, 9]
```

SEE ALSO: `BnFirstCode`, `BnLastCode`, `BnPrevCode`

5.8.20 `BnRd2Code` — calculate a code from a decomposition into simple reflections

✘

Calling Sequence:

`BnRd2Code`(rd)

`BnRd2Code`(rd, n)

`BN[BnRd2Code]`(rd)

`BN[BnRd2Code]`(rd, n)

Parameters:

rd — any list denoting a decomposition

n — any positive integer

Synopsis:

- The `BnRd2Code` function computes the code corresponding to the list rd, the entries of rd being interpreted as simple reflections.
- Let $rd = [i, j, k, \ldots]$, then `BnRd2Code`(rd) = `SG[MultPerm]`$(s_i, s_j, s_k, \ldots)$ where s_i stands for `BnTranspoPalin`(i).
- When the second argument is not present, n is taken to be the smallest possible one.
- The decomposition needs not to be reduced.
- Whenever there is a conflict between the function name `BnRd2Code` and another name used in the same session, use the long form `BN['BnCodeRd']`.

Examples:

```
> with(BN):
> BnRd2Code([0, 1, 2, 1, 0, 1, 2]);

                              [1, 1, 5]
```

SEE ALSO: `BnCode2Rd`

5.8.21 BnTranspoPalin — image of a simple reflection in $\mathfrak{S}_{2n}$

Calling Sequence:

BnTranspoPalin(i, n)

BN[BnTranspoPalin](i, n)

Parameters:

n — any positive integer

i — any non negative integer

Synopsis:

- The **BnTranspoPalin** function returns a generator of B_n. The result is an element of B_n embedded into $\mathfrak{S}_{2n}$, as a palindromic permutation.
- BnTranspoPalin$(0, n) = [1, \ldots, n+1, n, \ldots, 2n]$.
- BnTranspoPalin$(i, n) = [1, \ldots, n-i+1, n-i, \ldots, n+i+1, n+i, \ldots, 2n]$.
- Whenever there is a conflict between the function name **BnTranspoPalin** and another name used in the same session, use the long form **BN['BnTranspoPalin']**.

Examples:

```
> with(BN):
> BnTranspoPalin(1, 3);

                    [1, 3, 2, 5, 4, 6]
```

SEE ALSO:

5.9 Algebras related to hyperoctahedral groups: package BNA

Calling Sequence:

BNA[<function>]$(args)$

<function>$(args)$

Synopsis:

- This package provides functions to work with several algebras associated to the hyperoctahedral groups B_n, together with their realizations as algebras of operators on the ring of polynomials. It includes the B_n-group algebra together with the B_n-nilCoxeter and B_n-idCoxeter algebra. Bases of these algebras are indexed by codes of B_n.
- Whenever there is a conflict between a function name in **BNA** and another name used in the same session, use the long form **BNA[<function>]**.
- The available functions are:

BnGaMult	BnGaOnPol	BnGaYang	BnIdcaMult
BnIdcaOnPol	BnNcaMult	BnNcaOnPol	BnNcaYang
BnaAdd	BnaCode2Bar	BnaCode2Palin	BnaEmbed
BnaMinus	Rd2BnIdca	Rd2BnNca	

- For help with a particular function do either `?BNA[<function>]` or `?BNA,<function>` where `<function>` is one from the above list.
- Instead of `BnaAdd`, `BnaMinus`, `BnGaMult`, `BnNcaMult`, `BnIdcaMult`, one can use in short `&!!+`, `&!!-`, `&!!*`, `&!@*`, `&!$*`.

Examples:

```
> with(BNA):
> BnaCode2Palin(q*[0,1,0,2] - q^2*[1,1,0,3]);

                                          2
        q [2, 4, 1, 3, 6, 8, 5, 7] - q  [2, 5, 3, 1, 8, 6, 4, 7]
```

5.9.1 BnaAdd — sum of two elements of any B_n-algebra

Calling Sequence:

`BnaAdd`(e_1, e_2)

`BNA[BnaAdd]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of any B_n-algebra

Synopsis:

- The `BnaAdd` function adds two elements of either the B_n-group algebra, B_n-nilCoxeter algebra or the B_n-idCoxeter algebra.
- One can use instead the operator `&!!+`.
- Whenever there is a conflict between the function name `BnaAdd` and another name used in the same session, use the long form `BNA['BnaAdd']`.

Examples:

```
> with(BNA):
> a:=q + u/t*[0,1,2]:
> b:=t - [1,2,1]:
> BnaAdd(a, b);

               u [0, 1, 2]
               ----------- + (q + t) [0, 0, 0] - [1, 2, 1]
```

```
                  t

> a &!!+ b;

              u [0, 1, 2]
              ----------- + (q + t) [0, 0, 0] - [1, 2, 1]
                   t
```

SEE ALSO:

5.9.2 BnaMinus — difference of two elements of any B_n-algebra

Calling Sequence:

BnaMinus(e_1, e_2)

BNA[BnaMinus](e_1, e_2)

Parameters:

e_1, e_2 — any elements of any B_n-algebra

Synopsis:

- The BnaMinus function computes the difference between two elements of either the B_n-group algebra, B_n-nilCoxeter algebra or the B_n-idCoxeter algebra.
- One can use instead the operator &!!-.
- Whenever there is a conflict between the function name BnaMinus and another name used in the same session, use the long form BNA['BnaMinus'].

Examples:

```
> with(BNA):
> a:=q + u/t*[0,1,2]:
> b:=t - [1,2,1]:
> BnaMinus(a, b);

              u [0, 1, 2]
              ----------- + (q - t) [0, 0, 0] + [1, 2, 1]
                   t

> a &!!- a;

                                0
```

SEE ALSO:

5.9.3 BnaCode2Bar — convert codes into signed permutations

Calling Sequence:

BnaCode2Bar(e)

BNA[BnaCode2Bar](e)

Parameters:

e — any element of any B_n-algebra

Synopsis:

- The `BnaCode2Bar` function converts all codes that appear in the element e into signed permutations. The element e can belong to any B_n-algebra, but the result is no longer considered as an element of a B_n-algebra, the bases in this package being indexed by codes.

- Whenever there is a conflict between the function name `BnaCode2Bar` and another name used in the same session, use the long form `BNA['BnaCode2Bar']`.

Examples:

```
> with(BNA):
> BnaCode2Bar(q*[0,1,0,2] - q^2*[1,1,0,3]);

                                         2
                    q [2, 4, 1, 3] - q  [4, 2, -1, 3]
```

SEE ALSO:

5.9.4 `BnaCode2Palin` — convert codes into palindromic permutations

Calling Sequence:

`BnaCode2Palin`(e)

`BNA[BnaCode2Palin]`(e)

Parameters:

e — any element of any B_n-algebra

Synopsis:

- The `BnaCode2Palin` function converts all codes that appear in the element e into palindromic permutations. The element e can belong to any B_n-algebra but the result is no longer considered as an element of a B_n-algebra, the bases in this package being indexed by codes.

- Whenever there is a conflict between the function name `BnaCode2Palin` and another name used in the same session, use the long form `BNA['BnaCode2Palin']`.

Examples:

```
> with(BNA):
> BnaCode2Palin(q*[0,1,0,2] - q^2*[1,1,0,3]);

                                                 2
         q [2, 4, 1, 3, 6, 8, 5, 7] - q  [2, 5, 3, 1, 8, 6, 4, 7]
```

SEE ALSO:

5.9.5 `BnaEmbed` — reduce each code inside an expression to the minimum group containing it

✘

Calling Sequence:

`BnaEmbed`(e)

`BnaEmbed`(e, m)

`BNA[BnaEmbed]`(e)

`BNA[BnaEmbed]`(e, m)

Parameters:

e — any element of the B_n-group algebra

m — any positive integer denoting the degree of a B_n-group

Synopsis:

- The `BnaEmbed` function reduces each code inside an expression to the minimum B_n-group containing it (B_n is contained in B_{n+1} by addition of one 0 to the right).
- If a parameter m has been specified, the minimum is choosen to be m.
- Whenever there is a conflict between the function name `BnaEmbed` and another name used in the same session, use the long form `BNA['BnaEmbed']`.

✄ . . . .

Examples:

```
> with(BNA):
> BnaEmbed(a*[0,1,2] - b^2*[1,0,4,0]);

                          2
                       - b  [1, 0, 4] + a [0, 1, 2]

> BnaEmbed(a*[0,1,2] - b^2*[1,0,4,0], 3);

                          2
                       - b  [1, 0, 4] + a [0, 1, 2]
```

. . . . ✄

SEE ALSO:

5.9.6 `BnGaMult` — multiplication of two elements of the B_n-group algebra

✘

Calling Sequence:

`BnGaMult`(e_1, e_2)

`BNA[BnGaMult]`(e_1, e_2)

Parameters:

e_1, e_2 — any elements of the B_n-group algebra

Synopsis:

- The `BnGaMult` function multiplies two elements of the B_n-group algebra.

- One can use instead the operator `&!!*`.
- Whenever there is a conflict between the function name `BnGaMult` and another name used in the same session, use the long form `BNA['BnGaMult']`.

Examples:

```
> with(BNA):
> a:=q + u/t*[0,1,2]:
> b:=t - [1,2,1]:
> BnGaMult(a, b);

                                            u [0, 2, 5]
          u [0, 1, 2] + q t [0, 0, 0] - ----------- - q [1, 2, 1]
                                                 t

> a &!!* b;

                                            u [0, 2, 5]
          u [0, 1, 2] + q t [0, 0, 0] - ----------- - q [1, 2, 1]
                                                 t
```

SEE ALSO: `BnIdcaMult, BnNcaMult`

5.9.7 `BnGaOnPol` — action of an element of the B_n-algebra

Calling Sequence:

`BnGaOnPol`(e_1, exp)

`BnGaOnPol`(e_1, exp, v)

`BNA[BnGaOnPol]`(e_1, exp)

`BNA[BnGaOnPol]`(e_1, exp, v)

Parameters:

e_1 — any element of the B_n-algebra

exp — any expression

v — any (extra) string

Synopsis:

- The `BnGaOnPol` function realizes the action of an element of the B_n-algebra, say e_1, on an expression exp. The signed permutation action is to permute variable indices and to change the sign as indicated by the signed permutation.
- By default the algebra acts on the variables $x_1, x_2, x_3, \ldots$
- When called with a third argument v, being, say y the `BnGaOnPol` function acts on the variables $y_1, y_2, y_3, \ldots$
- Whenever there is a conflict between the function name `BnGaOnPol` and another name used in the same session, use the long form `BNA['BnGaOnPol']`.

Examples:

```
> with(BNA):
> BnGaOnPol([1,0,3], x1^3*x2^2*x3);

                                        3    2
                                  - x3  x1   x2

> BnGaOnPol([1,0,3], x1^3*x2^2*x3, y);

                                        3    2
                                    x1  x2   x3
```

SEE ALSO: BnIdcaOnPol, BnNcaOnPol

5.9.8 BnGaYang — Yang-Baxter element of the B_n-group algebra

Calling Sequence:

`BnGaYang`(*code*)

`BNA[BnGaYang]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The **BnGaYang** function calculates a Yang-Baxter element of the B_n-group algebra.
- { `BnGaYang`(*code*), $code \in B_n$ } is a linear basis of the B_n-group algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say $'y'$, one specifies that coefficients are in the y_i's.
- When this second parameter is **'num'** then $x_1, x_2, x_3, \ldots$ are specialized to $1, 2, 3, \ldots$
- This basis is defined by the recursion: for a simple reflection s_k, $k > 0$ and a code, such that $length(code\, s_k) > length(code)$, then one has:

$$\texttt{BnGaYang}(code\, s_k) = \texttt{BnGaYang}(code)\&!!{*}(1 + (\mathrm{sign}(perm[k])x_j - \mathrm{sign}(perm[k+1])x_i)s_k) \quad ,$$

where $i = |perm[k]|$, $j = |perm[k+1]|$ and $perm$ is the corresponding signed permutation. When $k = 0$, one has: $\texttt{BnGaYang}(code\, s_0) = \texttt{BnGaYang}(code)\&!!{*}(1 - 2(\mathrm{sign}(perm[1])x_i)s_0)$ in which $i = |perm[1]|$.

- Whenever there is a conflict between the function name **BnGaYang** and another name used in the same session, use the long form **BNA['BnGaYang']**.

Examples:

```
> with(BNA):
> BnGaYang([0,2]);

     - 2 x2 [1, 0] - 2 (x2 - x1) x2 [0, 2] + [0, 0] + (x2 - x1) [0, 1]

> BnGaYang([0,2], 'num');

                  - 4 [1, 0] - 4 [0, 2] + [0, 0] + [0, 1]
```

SEE ALSO: BnNcaYang

5.9.9 BnIdcaMult — multiplication of two elements of the B_n-idCoxeter algebra

Calling Sequence:

BnIdcaMult(e_1, e_2)

BNA[BnIdcaMult](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the B_n-idCoxeter algebra

Synopsis:

- The BnIdcaMult function multiplies two elements of the B_n-idCoxeter algebra.
- One can use instead the operator &!$*.
- Whenever there is a conflict between the function name BnIdcaMult and another name used in the same session, use the long form BNA['BnIdcaMult'].

Examples:

```
> with(BNA):
> a:=t - [1,2,1]:
> b:=q + u/t*[0,1,2]:
> BnIdcaMult(a, b);

           u [1, 3, 2]
           ----------- + u [0, 1, 2] - q [1, 2, 1] + t q [0, 0, 0]
                t

> a &!$* b;

           u [1, 3, 2]
           ----------- + u [0, 1, 2] - q [1, 2, 1] + t q [0, 0, 0]
                t
```

SEE ALSO: BnGaMult, BnNcaMult

5.9.10 `BnIdcaOnPol` — action of an element of the B_n-idCoxeter algebra

✘

Calling Sequence:

`BnIdcaOnPol`(e_1, exp)

`BnIdcaOnPol`(e_1, exp, v)

`BNA[BnIdcaOnPol]`(e_1, exp)

`BNA[BnIdcaOnPol]`(e_1, exp, v)

Parameters:

e_1 — any element of the B_n-idCoxeter algebra

exp — any expression

v — any (extra) string

Synopsis:

- The `BnIdcaOnPol` function realizes the action of an element of the B_n-idCoxeter algebra, say e_1, on an expression exp.
- The action of the i-th elementary generator is:

$$exp \longrightarrow \frac{exp - exp'}{x_i - x_{i+1}} x_{i+1} \quad , \qquad i \neq 0 \quad ,$$

$$exp \longrightarrow \frac{x_1 exp - \frac{1}{x_1} exp''}{1/x_1 - x1} \quad , \qquad i = 0 \quad ,$$

  where exp' is the image of exp under the exchange of x_i and x_{i+1} and exp'' is the image of exp under the exchange of x_1 and $1/x_1$.

- By default the algebra acts on the variables $x_1, x_2, x_3, \ldots$
- When called with a third argument v, being, say y the `BnIdcaOnPol` function acts on the variables $y_1, y_2, y_3, \ldots$
- Whenever there is a conflict between the function name `BnIdcaOnPol` and another name used in the same session, use the long form `BNA['BnIdcaOnPol']`.

Examples:

```
> with(BNA):
> BnIdcaOnPol([1,0,1], x1^3*x2^2*y3^2);

                6     4     2                     2
              (x1  + x1  + x1  + 1) (x2 + x3) y3   x3
            - -------------------------------------
                                  3
                                x1

> BnIdcaOnPol([1,0,1], x1^3*x2^2*y3^2, y);

                                  2    3
                      (y3 + y2) x2  x1  y3
```

See Also: `BnGaOnPol`, `BnNcaOnPol`

5.9.11 BnNcaMult — multiplication of two elements of the B_n-nilCoxeter algebra

Calling Sequence:

BnNcaMult(e_1, e_2)

BNA[BnNcaMult](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the B_n-nilCoxeter algebra

Synopsis:

- The BnNcaMult function multiplies two elements of the B_n-nilCoxeter algebra.
- One can use instead the operator &!@*.
- Whenever there is a conflict between the function name BnNcaMult and another name used in the same session, use the long form BNA['BnNcaMult'].

Examples:

```
> with(BNA):
> a:=t - [1,2,1]:
> b:=q + u/t*[0,1,2]:
> BnNcaMult(a, b);

                   u [0, 1, 2] + q t [0, 0, 0] - q [1, 2, 1]
> a &!@* b;

                   u [0, 1, 2] + q t [0, 0, 0] - q [1, 2, 1]
```

SEE ALSO: BnGaMult, BnIdcaMult

5.9.12 BnNcaOnPol — action of an element of the B_n-nilCoxeter algebra

Calling Sequence:

BnNcaOnPol(e_1, exp)

BnNcaOnPol(e_1, exp, v)

BNA[BnNcaOnPol](e_1, exp)

BNA[BnNcaOnPol](e_1, exp, v)

Parameters:

e_1 — any element of the B_n-nilCoxeter algebra

exp — any expression

v — any (extra) string

Synopsis:

- The BnNcaOnPol function realizes the action of an element of the B_n-nilCoxeter algebra, say e_1, on an expression exp.

- Simple divided differences ∂_i are the operators:

$$f \longrightarrow \frac{f(\ldots,x_i,x_{i+1},\ldots)-f(\ldots,x_{i+1},x_i,\ldots)}{x_i-x_{i+1}} \quad , \qquad i \neq 0 \quad ,$$

$$f \longrightarrow \frac{f(\ldots,x_i,\ldots)-f(\ldots,-x_i,\ldots)}{2x_i} \quad , \qquad i = 0 \quad .$$

- By default the algebra acts on the variables $x_1, x_2, x_3, \ldots$
- When called with a third argument v, being, say y the `BnNcaOnPol` function acts on the variables $y_1, y_2, y_3, \ldots$
- Whenever there is a conflict between the function name `BnNcaOnPol` and another name used in the same session, use the long form `BNA['BnNcaOnPol']`.

Examples:

```
> with(BNA):
> BnNcaOnPol([1,0,1], x1^3*x2^2*y3^2);

                                   2   2
                       (x3 + x2) x1  y3

> BnNcaOnPol([1,0,1], x1^3*x2^2*y3^2, y);

                                0
```

SEE ALSO: `BnGaOnPol`, `BnIdcaOnPol`

5.9.13 `BnNcaYang` — Yang-Baxter element of the B_n-nilCoxeter algebra

Calling Sequence:

`BnNcaYang`(*code*)

`BNA[BnNcaYang]`(*code*)

Parameters:

code — any list denoting a code

Synopsis:

- The `BnNcaYang` function calculates a Yang-Baxter element of the B_n-nilCoxeter algebra.
- { `BnNcaYang`(*code*), $code \in B_n$ } is a linear basis of the B_n-group algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say $'y'$, one specifies that coefficients are in the y_i's.
- When this second parameter is `'num'` then $x_1, x_2, x_3, \ldots$ are specialized to $1, 2, 3, \ldots$

- This basis is defined by the recursion: for a simple reflection s_k, $k > 0$ and a code, such that $length(code\, s_k) > length(code)$, then one has `BnNcaYang`$(code\, s_k) =$ `BnNcaYang`$(code)$&!@$*(1 + (\mathrm{sign}(perm[k]) * x_j - \mathrm{sign}(perm[k+1]) * x_i)\partial_k)$ where $i = |perm[k]|$, $j = |perm[k+1]|$ and $perm$ is the corresponding signed permutation. When $k = 0$, one has: `BnNcaYang`$(codes0) =$ `BnNcaYang`$(code)$&!@$*(1 - 2 * (\mathrm{sign}(perm[1]) * x_i)\partial_0)$ in which $i = |perm[1]|$.

- Whenever there is a conflict between the function name `BnNcaYang` and another name used in the same session, use the long form `BNA['BnNcaYang']`.

Examples:

```
> with(BNA):
> BnNcaYang([0,2]);

      (x2 - x1) [0, 1] + [0, 0] - 2 (x2 - x1) x2 [0, 2] - 2 x2 [1, 0]

> BnNcaYang([0,2], 'num');

                     [0, 0] + [0, 1] - 4 [1, 0] - 4 [0, 2]
```

SEE ALSO: `BnGaYang`

5.9.14 `Rd2BnIdca` — calculate a product of generators

Calling Sequence:

`Rd2BnIdca`(rd)

`Rd2BnIdca`(rd, n)

`BNA[Rd2BnIdca]`(rd)

`BNA[Rd2BnIdca]`(rd, n)

Parameters:

rd — any list denoting a decomposition

n — any positive integer

Synopsis:

- The `Rd2BnIdca` function calculates the product of the generators given by the reduced decomposition rd. The product is taken in the B_n-idCoxeter algebra.

- If rd is reduced then `Rd2BnIdca`$(rd) =$ `BN['BnRd2Code']`(rd), and otherwise there is a possible extra sign.

- When the second argument is not present, n is the smallest possible degree.

- Whenever there is a conflict between the function name `Rd2BnIdca` and another name used in the same session, use the long form `BNA['Rd2BnIdca']`.

Examples:

```
> with(BNA):
> Rd2BnIdca([0,1,0,1], 5);
```

```
                    [1, 3, 0, 0, 0]
```

```
> Rd2BnIdca([0,1,0,1]);
```

```
                    [1, 3]
```

```
> Rd2BnIdca([1,0,1,0,1]);
```

```
                    - [1, 3]
```

```
> Rd2BnIdca([0,0]);
```

```
                    [1]
```

SEE ALSO: `BN[BnRd2Code]`

5.9.15 `Rd2BnNca` — calculate a product of generators

Calling Sequence:

`Rd2BnNca`(rd)

`Rd2BnNca`(rd, n)

`BNA[Rd2BnNca]`(rd)

`BNA[Rd2BnNca]`(rd, n)

Parameters:

rd — any list denoting a decomposition

n — any positive integer

Synopsis:

- The `Rd2BnNca` function calculates the product of the generators given by the reduced decomposition rd. The product is taken in the B_n-nilCoxeter algebra.
- If rd is reduced then `Rd2BnNca`(rd) = `BN['BnRd2Code']`(rd), and 0 otherwise.
- When the second argument is not present, n is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2BnNca` and another name used in the same session, use the long form `BNA['Rd2BnNca']`.

Examples:

```
> with(BNA):
> Rd2BnNca([0,1,0,1], 5);
```

```
                    [1, 3, 0, 0, 0]
```

```
> Rd2BnNca([0,1,0,1]);

                                   [1, 3]

> Rd2BnNca([1,0,1,0,1]);

                                     0
```

SEE ALSO: BN[BnRd2Code]

5.10 Fock space representations of the quantum affine algebras $U_q(\widehat{sl}_n)$

Calling Sequence:

<function>$(args)$

FOCK[<function>]$(args)$

Synopsis:

- This package provides functions to compute in the Fock space representations of the quantum affine algebras $\widehat{sl}_n$ and $\widehat{gl}_n$, and in particular, to compute canonical bases.
- Whenever there is a conflict between a function name in FOCK and another name used in the same session, use the long form FOCK[<function>].
- The available functions are:

ABasis	AOp	BOp	CB
CBMat	DivPowE	DivPowF	EE
Etilda	FF	Ftilda	Invol
InvolMat	IsReg	ListRegPart	MK
Reflect	Shapovalov	TableG	TableGup
UOp	VOp	nWeight	

- For help with a particular function do either ?FOCK[<function>] or ?FOCK,<function> where <function> is one from the above list.

Examples:

```
> with(FOCK):
> CBMat(2, 5);

                        [ 1  0   0  0  0  0  0 ]
                        [                      ]
                        [ 0  1   0  0  0  0  0 ]
                        [                      ]
                        [ 0  0   1  0  0  0  0 ]
                        [                      ]
```

```
[ q  0   q  1  0  0  0 ]
[                      ]
[        2             ]
[ 0  0  q   q  1  0  0 ]
[                      ]
[ 0  q   0  0  0  1  0 ]
[                      ]
[ 2                    ]
[q   0   0  q  0  0  1 ]
```

SEE ALSO:

5.10.1 ABasis — intermediate monomial basis of the basic representation

Calling Sequence:

ABasis($n, lambda$)

FOCK[ABasis]($n, lambda$)

Parameters:

n — an integer greater than 1

$lambda$ — any partition

Synopsis:

- Computes the intermediate invariant basis A of the basic representation of $U_q(\widehat{sl}_n)$. $A(\lambda)$ is obtained by applying to the highest weight vector a product of lowering operators read on the ladder decomposition of λ.
- Whenever there is a conflict between the function name ABasis and another name used in the same session, use the long form FOCK['ABasis'].

Examples:

```
> with(FOCK):
> ABasis(2, [5,3,1]);

      3                  4                      3
     q  s[3, 2, 2, 2] + q  s[3, 2, 2, 1, 1] + q  s[3, 3, 1, 1, 1]

        2
      + q  s[3, 3, 3] + q s[4, 4, 1] + s[5, 3, 1] + s[5, 2, 2] q

        2                  2
      + q  s[5, 2, 1, 1] + q  s[4, 2, 1, 1, 1]
```

SEE ALSO:

5.10.2 `AOp` — q-analogue of derivative with respect to a power sum

Calling Sequence:

`AOp`(sf, k, n)

`FOCK[AOp]`(sf, k, n)

Parameters:

sf — any linear combination of Schur functions

k — any non negative integer less than n

n — an integer greater than 1

Synopsis:

- Kashiwara-Miwa-Stern operators.
- Computes the image of a linear combination of Schur functions sf by the q-analogue of the differential operator of the power-sum p_{nk}.
- This operator commutes with $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `AOp` and another name used in the same session, use the long form `FOCK['AOp']`.

Examples:

```
> with(FOCK):
> AOp(s[4,3,2,1], 2, 3):
> Tos(", collect);

                    2                          2
           (- 1 + q ) s[1, 1, 1, 1]   (- 1 + q ) s[4]
           ------------------------ + ---------------
                       3                      3
                      q                      q
```

SEE ALSO: `BOp`, `UOp`, `VOp`

5.10.3 `BOp` — q-analogue of multiplication by a power sum

Calling Sequence:

`BOp`(sf, k, n)

`FOCK[BOp]`(sf, k, n)

Parameters:

sf — any linear combination of Schur functions

k — any non negative integer less than n

n — an integer greater than 1

Synopsis:

- Kashiwara-Miwa-Stern operators.

- Computes the image of a linear combination of Schur functions sf by the q-analogue of the multiplication operator by the power-sum p_{nk}.
- This operator commutes with $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `BOp` and another name used in the same session, use the long form `FOCK['BOp']`.

Examples:

```
> with(FOCK):
> BOp(s[2,1], 2, 2):
> Tos(", collect);

          s[2, 2, 2, 1]              s[2, 1, 1, 1, 1, 1]
          ------------- + s[6, 1] - ------------------- - s[4, 3]
                2                            2
               q                            q
```

SEE ALSO: `AOp`, `UOp`, `VOp`

5.10.4 `CB` — canonical basis

Calling Sequence:

`CB`$(n, m, lambda)$

`FOCK[CB]`$(n, m, lambda)$

Parameters:

n	—	an integer greater than 1
m	—	any positive integer
$lambda$	—	any partition

Synopsis:

- Computes the vector of the $U_q(\widehat{sl}_n)$-canonical basis of the Fock space indexed by a partition λ of weight m.
- Whenever there is a conflict between the function name `CB` and another name used in the same session, use the long form `FOCK['CB']`.

Examples:

```
> with(FOCK):
> CB(2, 8, [3,2,2,1]);

                                              2
          s[3, 2, 2, 1] + q s[2, 2, 2, 2] + q  s[2, 2, 2, 1, 1]
```

SEE ALSO: `CBMat`, `TableG`, `TableGup`

5.10.5 `CBMat` — matrix of canonical basis

Calling Sequence:

`CBMat`(n, m)

`FOCK[CBMat]`(n, m)

Parameters:

n — an integer greater than 1

m — any positive integer

Synopsis:

- Computes the matrix of the $U_q(\widehat{sl}_n)$-canonical basis of the Fock space in the basis of partitions of weight m, represented by Schur functions.
- Rows and columns are labelled by partitions in the order given by the function `SYMF[ListPart]`.
- Whenever there is a conflict between the function name `CBMat` and another name used in the same session, use the long form `FOCK['CBMat']`.

Examples:

```
> with(FOCK):
> CBMat(2, 4);
```

```
                    [ 1   0  0  0  0 ]
                    [                ]
                    [ q   1  0  0  0 ]
                    [                ]
                    [ 0   q  1  0  0 ]
                    [                ]
                    [      2         ]
                    [ q  q   q  1  0 ]
                    [                ]
                    [ 2              ]
                    [q    0  0  q  1 ]
```

See Also: `CB, TableG, TableGup`

5.10.6 `DivPowE` — divided powers of e-operators of $U_q(\widehat{sl}_n)$

Calling Sequence:

`DivPowE`(n, k, r, sf)

`FOCK[DivPowE]`(n, k, r, sf)

Parameters:

n — an integer greater than 1

k — a non negative integer less than n

r — any positive integer

sf — any linear combination of Schur functions

Synopsis:

- Computes the image of a linear combination of Schur functions sf by the r-th divided power of the Chevalley generator e_k of $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `DivPowE` and another name used in the same session, use the long form `FOCK['DivPowE']`.

Examples:

```
> with(FOCK):
> DivPowE(3, 2, 2, s[3,2,2]);

                                   s[2, 2, 1]

> EE(s[3,2,2], 3, 2): EE(", 3, 2):
> Tos(", collect);

                                  2
                                (q  + 1) s[2, 2, 1]
                                -------------------
                                         q
```

SEE ALSO: `DivePowF, EE, Etilda, FF, Ftilda`

5.10.7 `DivPowF` — divided powers of f-operators of $U_q(\widehat{sl}_n)$

Calling Sequence:

`DivPowF`(n, k, r, sf)

`FOCK[DivPowF]`(n, k, r, sf)

Parameters:

n	—	an integer greater than 1
k	—	a non negative integer less than n
r	—	any positive integer
sf	—	any linear combination of Schur functions

Synopsis:

- Computes the image of a linear combination of Schur functions sf by the r-th divided power of the Chevalley generator f_k of $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `DivPowF` and another name used in the same session, use the long form `FOCK['DivPowF']`.

Examples:

```
> with(FOCK):
> DivPowF(2, 1, 2, s[1]);

                                   s[2, 1]
```

```
> FF(s[1], 2, 1): FF(", 2, 1):
> Tos(", collect);
                                2
                              (q  + 1) s[2, 1]
                              ----------------
                                     q
```

SEE ALSO: `DivPowE, EE, Etilda, FF, Ftilda`

5.10.8 `EE` — raising operators of quantum affine $U_q(\widehat{sl}_n)$

Calling Sequence:

`EE`(sf, n, k)

`FOCK[EE]`(sf, n, k)

Parameters:

sf — any linear combination of Schur functions

n — an integer greater than 1

k — a positive integer less than n

Synopsis:

- Computes the image of a linear combination of Schur functions sf by the Chevalley generator e_k of $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `EE` and another name used in the same session, use the long form `FOCK['EE']`.

Examples:

```
> with(FOCK):
> EE(2*s[3,2]-(q+q^2)*s[4,1], 2, 0);
                                         s[3, 1]
                        2 s[2, 2] + 2 -------
                                            q
```

SEE ALSO: `DivPowE, DivPowF, Etilda, FF, Ftilda`

5.10.9 `Etilda` — crystal graph raising operators of quantum $U_q(\widehat{sl}_n)$

Calling Sequence:

`Etilda`$(n, k, lambda)$

`FOCK[Etilda]`$(n, k, lambda)$

Parameters:

n — an integer greater than 1

k — a non negative integer less than n

$lambda$ — any partition

Synopsis:

- Computes the predecessor of a partition λ on its string of colour k in the crystal graph of the basic representation of $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `Etilda` and another name used in the same session, use the long form `FOCK['Etilda']`.

Examples:

```
> with(FOCK):
> Etilda(3, 0, [3,2,1,1]);
                                [3, 2, 1]
> Etilda(3, 1, [3,2,1,1]);
                                    0
> Etilda(3, 2, [3,2,1,1]);
                               [2, 2, 1, 1]
```

SEE ALSO: `DivPowE, DivPowF, EE, FF, Ftilda`

5.10.10 FF — lowering operators of quantum $U_q(\widehat{sl}_n)$

Calling Sequence:

`FF`(sf, n, k)

`FOCK[FF]`(sf, n, k)

Parameters:

sf — any linear combination of Schur functions

n — an integer greater than 1

k — a non negative integer less than n

Synopsis:

- Computes the image of a linear combination of Schur functions sf by the Chevalley generator f_k of $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `FF` and another name used in the same session, use the long form `FOCK['FF']`.

Examples:

```
> with(FOCK):
> FF(2*s[3,2]-(q+q^2)*s[4,1], 2, 0);

            s[3, 2, 1]          2                             2
          2 ---------- - (q + q ) (s[5, 1] + q s[4, 2] + q  s[4, 1, 1])
                 2
                q
```

SEE ALSO: `DivPowE, DivPowF, EE, Etilda, Ftilda`

5.10.11 `Ftilda` — crystal graph lowering operators of quantum $U_q(\widehat{sl}_n)$

Calling Sequence:

`Ftilda`$(n, k, lambda)$

`FOCK[Ftilda]`$(n, k, lambda)$

Parameters:

n — an integer greater than 1

k — a non negative integer less than n

$lambda$ — any partition

Synopsis:

- Computes the successor of a partition λ on its string of colour k in the crystal graph of the basic representation of $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `Ftilda` and another name used in the same session, use the long form `FOCK['Ftilda']`.

Examples:

```
> with(FOCK):
> Ftilda(3, 0, [3,2,1,1]);

                                       0

> Ftilda(3, 1, [3,2,1,1]);

                                  [3, 3, 1, 1]

> Ftilda(3, 2, [3,2,1,1]);

                                  [3, 2, 2, 1]
```

SEE ALSO: `DivPowE, DivPowF, EE, Etilda, FF`

5.10.12 `Invol` — involution of the Fock space

Calling Sequence:

`Invol`(n, m, sf)

`FOCK[Invol]`(n, m, sf)

Parameters:

n — an integer greater than 1

m — any positive integer

sf — any linear combination of Schur functions

Synopsis:

- Computes the image of a symmetric function sf by the canonical involution of the Fock space, in degree m for $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `Invol` and another name used in the same session, use the long form `FOCK['Invol']`.

Examples:

```
> with(FOCK):
> Invol(3, 7, s[3,2,1,1]);

                           2
                         (q  - 1) s[2, 2, 2, 1]      2
       s[3, 2, 1, 1] + ---------------------- + (q  - 1) s[2, 2, 1, 1, 1]
                                   q
```

SEE ALSO: `InvolMat`

5.10.13 `InvolMat` — matrix of the involution of the Fock space

Calling Sequence:

`InvolMat`(n, m)

`FOCK[InvolMat]`(n, m)

Parameters:

n — an integer greater than 1

m — any positive integer

Synopsis:

- Computes the matrix (as a semi-linear map) of the canonical involution of the Fock space, in degree m for $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `InvolMat` and another name used in the same session, use the long form `FOCK['InvolMat']`.

Examples:

```
> with(FOCK):
> InvolMat(2, 3);

                              [     1     0  0 ]
                              [                ]
                              [     0     1  0 ]
                              [                ]
                              [         2      ]
                              [ - 1 + q        ]
                              [ --------  0  1 ]
                              [     q          ]
```

SEE ALSO: `Invol`

5.10.14 `IsReg` — tests whether a partition is n-regular

Calling Sequence:

`IsReg`$(n, lambda)$

`FOCK[IsReg]`$(n, lambda)$

Parameters:

n — an integer greater than 1

$lambda$ — any partition

Synopsis:

- Returns true if the partition λ is n-regular, and false otherwise.
- Whenever there is a conflict between the function name `IsReg` and another name used in the same session, use the long form `FOCK['IsReg']`.

Examples:

```
> with(FOCK):
> IsReg(2, [4,3,3,1]);

                                   false

> IsReg(2, [4,3,2,1]);

                                    true
```

SEE ALSO: `ListRegPart`

5.10.15 `ListRegPart` — list of n-regular partitions of an integer m

Calling Sequence:

`ListRegPart`(n, m)

`FOCK[ListRegPart]`(n, m)

Parameters:

n — an integer greater than 1

m — any positive integer

Synopsis:

- Returns the list of n-regular partitions of m in reverse lexicographic order.
- Whenever there is a conflict between the function name `ListRegPart` and another name used in the same session, use the long form `FOCK['ListRegPart']`.

Examples:

```
> with(FOCK):
> ListRegPart(2, 6);

                  [[6], [5, 1], [4, 2], [3, 2, 1]]

> ListRegPart(3, 8);

  [[8], [7, 1], [6, 2], [6, 1, 1], [5, 3], [5, 2, 1], [4, 4], [4, 3, 1],

     [4, 2, 2], [4, 2, 1, 1], [3, 3, 2], [3, 3, 1, 1], [3, 2, 2, 1]]
```

SEE ALSO: `IsReg`

5.10.16 `MK` — Mullineux-Kleshchev involution

Calling Sequence:

`MK`$(n, lambda)$

`FOCK[MK]`$(n, lambda)$

Parameters:

n — an integer greater than 1

$lambda$ — any partition

Synopsis:

- Computes the image of a partition λ under the natural symmetry of the crystal graph of the basic representation of $U_q(\widehat{sl}_n)$.
- This symmetry is exactly the Mullineux involution of the modular representation theory of symmetric groups, in the description given by Kleshchev.
- Whenever there is a conflict between the function name `MK` and another name used in the same session, use the long form `FOCK['MK']`.

Examples:

```
> with(FOCK):
> MK(3, [3,2,1,1]);
```

```
                                [5, 2]
```

SEE ALSO:

5.10.17 `Reflect` — action of the affine Weyl group on the crystal graph

✘

Calling Sequence:

`Reflect`$(n, k, lambda)$

`FOCK[Reflect]`$(n, k, lambda)$

Parameters:

n	—	an integer greater than 1
k	—	any non negative integer
$lambda$	—	any partition

Synopsis:

- Computes the image of the vertex λ of the crystal graph of the Fock space representation of $U_q(\widehat{sl}_n)$ by the reflection s_k of the Weyl group of $\widehat{sl}(n)$.
- Whenever there is a conflict between the function name `Reflect` and another name used in the same session, use the long form `FOCK['Reflect']`.

Examples:

```
> with(FOCK):
> Reflect(3, 0, [3,2,1,1]);
```

```
                               [3, 2, 1]
```

```
> Reflect(3, 1, [3,2,1,1]);
```

```
                              [3, 3, 1, 1]
```

```
> Reflect(3, 2, [3,2,1,1]);
```

```
                              [3, 2, 2, 1]
```

SEE ALSO: `Etilda, Ftilda`

5.10.18 `Shapovalov` — contravariant form of the Fock space

Calling Sequence:

`Shapovalov`(f, g, n)

`FOCK[Shapovalov]`(f, g, n)

Parameters:

f, g — any linear combinations of Schur functions

n — an integer greater than 1

Synopsis:

- Evaluates the Shapovalov (or contravariant) form of the Fock space representation of $U_q(\widehat{sl}_n)$ on two linear combination of Schur functions f and g.
- Whenever there is a conflict between the function name `Shapovalov` and another name used in the same session, use the long form `FOCK['Shapovalov']`.

Examples:

```
> with(FOCK):
> Shapovalov(s[6,3,2,1]+2*s[7,5], s[8,3,1]+s[6,3,2,1]-s[7,5], 2);

                                        2
                              1/q - ----
                                        6
                                       q
```

SEE ALSO:

5.10.19 `TableG` — lower crystal basis of the basic representation

Calling Sequence:

`TableG`(n, m)

`FOCK[TableG]`(n, m)

Parameters:

n — an integer greater than 1

m — any positive integer

Synopsis:

- Computes the global lower crystal basis of the basic representation of $U_q(\widehat{sl}_n)$.
- The result is a table indexed by n-regular partitions of m.
- Whenever there is a conflict between the function name `TableG` and another name used in the same session, use the long form `FOCK['TableG']`.

Examples:

```
> with(FOCK):
> TableG(2, 5);

              table([
                                              2
                  5 = s[5] + s[3, 1, 1] q + q  s[1, 1, 1, 1, 1]
                               2
                  (3, 2) = q  s[2, 2, 1] + s[3, 1, 1] q + s[3, 2]
                  (4, 1) = s[4, 1] + q s[2, 1, 1, 1]
              ])
```

SEE ALSO: CB, CBMat, TableGup

5.10.20 TableGup — global upper crystal basis of the basic representation

Calling Sequence:

TableGup(n, m)

FOCK[TableGup](n, m)

Parameters:

n — an integer greater than 1

m — any positive integer

Synopsis:

- Computes the global upper crystal basis of the basic representation of $U_q(\widehat{sl}_n)$.
- The result is a table indexed by n-regular partitions of m.
- Whenever there is a conflict between the function name TableGup and another name used in the same session, use the long form FOCK['TableGup'].

Examples:

```
> with(FOCK):
> TableGup(2, 3);

                    table([
                        (2, 1) = s[2, 1]
                            q (s[3] + q s[1, 1, 1])
                        3 = -----------------------
                                       2
                                      q  + 1
                    ])
```

SEE ALSO: CB, CBMat, TableG

5.10.21 `UOp` — q-analogue of derivation with respect to $p_n[h_\lambda]$

Calling Sequence:

`UOp`$(sf, lambda, n)$

`FOCK[UOp]`$(sf, lambda, n)$

Parameters:

sf — any linear combination of Schur functions

$lambda$ — any partition

n — an integer greater than 1

Synopsis:

- Computes the image of a linear combination of Schur functions sf by the q-analogue of the differential operator associated to the plethysm $p_n[h_\lambda]$ of a product of complete homogeneous functions by a power sum.
- This operator commutes with $U_q(\widehat{sl}_n)$.
- Whenever there is a conflict between the function name `UOp` and another name used in the same session, use the long form `FOCK['UOp']`.

Examples:

```
> with(FOCK):
> UOp(s[4,2,1,1], [2,1], 2):
> Tos(", collect);

                                s[2]     s[1, 1]
                              - ---- + 2 -------
                                 q          2
                                           q
```

SEE ALSO: `AOp, BOp, VOp`

5.10.22 `VOp` — q-analogue of multiplication by $p_n[h_\lambda]$

Calling Sequence:

`VOp`$(sf, lambda, n)$

`FOCK[VOp]`$(sf, lambda, n)$

Parameters:

sf — any linear combination of Schur functions

$lambda$ — any partition

n — an integer greater than 1

Synopsis:

- Computes the image of a linear combination of Schur functions sf by the q-analogue of the multiplication operator by the plethysm $p_n[h_\lambda]$ of a product of complete homogeneous functions by a power sum.

- The images of the empty partition by these operators form a basis of highest weight vectors of the Fock space representation.
- Whenever there is a conflict between the function name `VOp` and another name used in the same session, use the long form `FOCK['VOp']`.

Examples:

```
> with(FOCK):
> VOp(s[1], [1,1], 2):
> Tos(", collect);

        s[1, 1, 1, 1, 1]   s[2, 2, 1]     s[3, 1, 1]
        ---------------- + ---------- - 2 ---------- + s[3, 2] + s[5]
                2                2              q
               q                q
```

SEE ALSO: `AOp`, `BOp`, `UOp`

5.10.23 `nWeight` — n-weight of a partition

Calling Sequence:

`nWeight`$(n, lambda)$

`FOCK[nWeight]`$(n, lambda)$

Parameters:

n	—	an integer greater than 1
$lambda$	—	any partition

Synopsis:

- Computes the n-weight of a partition λ, that is, the number of n-ribbons to be removed to reach the n-core of λ.
- Whenever there is a conflict between the function name `nWeight` and another name used in the same session, use the long form `FOCK['nWeight']`.

Examples:

```
> with(FOCK):
> nWeight(3, [8,5,4,2,1]);

                                    6
```

SEE ALSO:

5.11 SHRI, a package for symmetric group Hecke algebra representations and isomorphisms

✓

Calling Sequence:

SHRI[<function>]($args$)

<function>($args$)

Synopsis:

- This package provides functions to work with the usual Hecke algebra of type A and Hecke algebras of type A with other spectral parameters essentially the generic Hecke algebra and the Hecke algebras with symmetric parameters. The usual Hecke algebra is generated by simple operators T_i's satisfying braid relations and the Hecke relation:

$$T_i^2 = (q-1)\,T_i + q \quad .$$

  In the same way, the operators of the generic Hecke algebra satisfy:

$$T_i^2 = (q_1+q_2)\,T_i - q_1 q_2 \quad .$$

  And for Hecke algebras with symmetric parameters:

$$T_i^2 = (\frac{1}{v} - v)\,T_i + 1 \quad .$$

- An element of Hecke algebras is expressed as a linear combination of permutations with arbitrary coefficients.
- Whenever there is a conflict between a function name in SHRI and another name used in the same session, use the long form SHRI[<function>].
- The available functions are:

ShriCentralIdemp	ShriCharTable	ShriConvert	ShriCpCT
ShriCpT	ShriIdemp	ShriInv	ShriJucis
ShriMpCT	ShriMpT	ShriRepCT	ShriRepT

- Functions as ShriCpCT and ShriCpT (or ShriMpCT and ShriMpT) correspond to representations computed with different bases; they return the same value but for certain shapes, computation times can be different.
- For help with a particular function do either ?SHRI[<function>] or ?SHRI,<function> where <function> is one from the above list.

Examples:

```
> with(SHRI):
> ShriRepT(a*[2,3,1] + b*[3,1,2],[2,1,1]);

             [              3             3               3        ]
             [ a + b    a q           a q  (q - 1) - b q           ]
```

```
[                                                    ]
[              2                        2       2    ]
[   0     - a q      a (- q - (q - 1) q ) + b q      ]
[                                                    ]
[                                                2   ]
[   0    a q - b  a (q - 1) q + b ((q - 1) q - q )   ]
```

5.11.1 ShriCpCT — characteristic polynomial for the contretableaux

✓

Calling Sequence:

ShriCpCT($e, part, x$)

ShriCpCT($e, part, x, l$)

SHRI[ShriCpCT]($e, part, x$)

SHRI[ShriCpCT]($e, part, x, l$)

Parameters:

e	—	any element of the usual, generic or symmetric Hecke algebra
part	—	any list of positive integers in weakly increasing order
x	—	any name or algebraic expression
l	—	any (extra) letter of {G, Q, S}

Synopsis:

- The ShriCpCT function returns the characteristic polynomial in x of the image of the element e of the usual, generic or symmetric Hecke algebra in the irreducible representation indexed by the shape *part* (l is equal to Q, G or S).
- If l is not specified then it means that l is equal to Q.
- Whenever there is a conflict between the function name ShriCpCT and another name used in the same session, use the long form SHRI[ShriCpCT].

Examples:

```
> with(SHRI):
> ShriCpCT(a*[2,1]+b*[3,1,2],[1,2],x);

   2                              2      2              2  2
  x  - x a q + x a + x b q - b q  a - a  q + b q a + b  q

> ShriCpCT([1,4,3,2]+1,[1,1,2],x,G);

                       3    2                3   3
              (x - 1 - q1 ) (x  - 2 x + 1 + q2  q1 )
```

SEE ALSO: ShriCpT, ShriMpCT, ShriMpT

5.11.2 ShriCpT — characteristic polynomial for the tableaux

✓

Calling Sequence:

ShriCpT(e,$part$,x)

ShriCpT(e,$part$,x,l)

SHRI[ShriCpT](e,$part$,x)

SHRI[ShriCpT](e,$part$,x,l)

Parameters:

e	—	any element of the usual, generic or symmetric Hecke algebra
$part$	—	any list of integers in weakly decreasing order
x	—	any name or algebraic expression
l	—	any (extra) letter of {G,Q,S}

Synopsis:

- The ShriCpT function returns the characteristic polynomial in x of the image of the element e of the usual, generic or symmetric Hecke algebra in the irreducible representation indexed by the shape $part$ (l is equal to Q, G or S).
- If l is not specified then it means that l is equal to Q.
- Whenever there is a conflict between the function name ShriCpT and another name used in the same session, use the long form SHRI[ShriCpT].

Examples:

```
> with(SHRI):
> ShriCpT(a*[2,1]+b*[3,1,2],[2,1],x);

      2                             2      2               2   2
     x  - x a q + x a + x b q - b q  a - a  q + b q a + b  q

> ShriCpT([2,1,3]-[3,1,2],[1,1,1],x,S);

                                2    2 1/2      2 1/2     2
                              x v  (v )    + (v )    + v
                             ---------------------------
                                       2    2 1/2
                                      v   (v )
```

SEE ALSO: ShriCpCT, ShriMpCT, ShriMpT

5.11.3 ShriCentralIdemp — central idempotent

✓

Calling Sequence:

ShriCentralIdemp($part$)

SHRI[ShriCentralIdemp]($part$)

Parameters:

part — any list of positive integers in weakly decreasing order

Synopsis:

- The `ShriCentralIdemp` function calculates the central idempotent of the shape *part* for the usual Hecke algebra of type A. The program uses Kilmoyer's formula.
- Whenever there is a conflict between the function name `ShriCentralIdemp` and another name used in the same session, use the long form `SHRI[ShriCentralIdemp]`.

Examples:

```
> with(SHRI):
> ShriCentralIdemp([1,1]);

                            [2, 1]   q [1]
                          - ------ + -----
                             q + 1   q + 1
```

SEE ALSO: `ShriIdemp`

5.11.4 `ShriCharTable` — characters of the generic Hecke algebra

Calling Sequence:

`ShriCharTable`(n)

`ShriCharTable`$(n,$ `'index'`$)$

`ShriCharTable`$(n, part_1, part_2)$

`ShriCharTable`$(e, part)$

`SHRI[ShriCharTable]`(n)

`SHRI[ShriCharTable]`$(n,$ `'index'`$)$

`SHRI[ShriCharTable]`$(n, part_1, part_2)$

`SHRI[ShriCharTable]`$(e, part)$

Parameters:

n — any positive integer denoting the degree of a symmetric group

$part_1, part_2$ — any lists denoting partitions

e — any element of the generic Hecke algebra

part — any list denoting partition

Synopsis:

- The `ShriCharTable` gives the characters table for the generic Hecke algebra of order n. Let t be this matrix, $part_1$ and $part_2$ be two partitions of n of index i, j in `ShriCharTable`$(n,$ `'index'`$)$ the list of partitions then $t[i,j]$ is equal to the character, in any irreducible representation of $part_1$, of the element T_w in which w is the product of cycles:

$$(1, 2, \ldots, k_1)(k_1+1, \ldots, k_1+k_2)(k_1+k_2+1, \ldots, k_1+k_2+k_3)\ldots$$

- One may extract some rows or columns by the `linalg[row]` and `linalg[col]` commands of the `linalg` package.
- The `ShriCharTable` function also returns the character of the element e of the generic Hecke algebra in any representation of shape *part*. The program uses a bivariate deformation of the Murnaghan-Nakayama rule.
- Whenever there is a conflict between the function name `ShriCharTable` and and another name used in the same session, use the long form `SHRI[ShriCharTable]`.

Examples:

```
> with(SHRI):
> ShriCharTable(3);

                    [    2               ]
                    [  q2        q2    1 ]
                    [                    ]
                    [ q1 q2  q1 + q2   2 ]
                    [                    ]
                    [    2               ]
                    [  q1        q1    1 ]

> ShriCharTable(5,[2,2,1],[3,2]);

                        2                2
                  q1 q2  + (q1 + q2) q1

> ShriCharTable(1+2*[2,3,1]-[1,4,3,2],[2,1,1]);

                                          2
        3 + 2 q1 (q1 + q2) - (q1 + q2)  q1 + q1 q2 (2 q1 + q2)

> normal(subs(q1=q,q2=-1,"));

                                 2          3
                          3 + 2 q  - 2 q - q
```

SEE ALSO: `ShriRepT, SG[SnCharTable]`

5.11.5 `ShriIdemp` — orthogonal minimal idempotent

✓

Calling Sequence:

`ShriIdemp`(t)

`SHRI[ShriIdemp]`(t)

Parameters:

t — any Yamanouchi word coding a tableau

Synopsis:

- The `ShriIdemp` function returns the orthogonal minimal idempotent of the usual Hecke algebra of type A for the tableau t within the Jucis-Murphy subalgebra.

- Yamanouchi words are given on the alphabet $1, 2, 3, \ldots$
- Whenever there is a conflict between the function name `ShriIdemp` and another name used in the same session, use the long form `SHRI[ShriIdemp]`.

Examples:

```
> with(SHRI):
> ShriIdemp([1,2,1]);

    q [1]         q [3, 1, 2]             [3, 2, 1]          [2, 1]
   ---------- - -------------------- + -------------------- - ----------
    2             2                     2                    2
   q  + q + 1   (q  + q + 1) (q + 1)   (q  + q + 1) (q + 1)   q  + q + 1

                 2
                q  [1, 3, 2]         q [2, 3, 1]
         + -------------------- - --------------------
              2                      2
            (q  + q + 1) (q + 1)   (q  + q + 1) (q + 1)
```

SEE ALSO: `ShriCentralIdemp`

5.11.6 `ShriInv` — invertible of an element of a Hecke algebra

✓

Calling Sequence:

`ShriInv`(e)

`ShriInv`(e, l)

`SHRI[ShriInv]`(e)

`SHRI[ShriInv]`(e, l)

Parameters:

e — any element of the usual, generic or symmetric Hecke algebra

l — any (extra) letter of {`G`, `Q`, `S`}

Synopsis:

- The `ShriInv` function returns the inverse of e in the usual, generic or symmetric Hecke algebra (l is equal to `Q, G` or `S`).
- If l is not specified then it means that l is equal to `Q`.
- Whenever the element e is not invertible, the function returns 0.
- Whenever there is a conflict between the function name `ShriInv` and another name used in the same session, use the long form `SHRI[ShriInv]`.

Examples:

```
> with(SHRI):
> e:=a*[2,1]:
> ShriInv(e);

                              (q - 1) [1]    [2, 1]
                            - ----------- + ------
                                  a q        a q

> ShriInv([2,1]-[1,3,2]);

                                        0

> ShriInv(1+[2,1],G);

                     (1 + q1 + q2) [1]          [2, 1]
                     ----------------- - -----------------
                     (1 + q1) (1 + q2)   (1 + q1) (1 + q2)
```

SEE ALSO:

5.11.7 ShriMpCT — minimal polynomial for the contretableaux

✓

Calling Sequence:

ShriMpCT(e, x, l)

ShriMpCT($e, part, x, l$)

SHRI[ShriMpCT](e, x, l)

SHRI[ShriMpCT]($e, part, x, l$)

Parameters:

e — any element of the usual, generic or symmetric Hecke algebra

$part$ — any (extra) list of positive integers in weakly increasing order

x — any name or algebraic expression

l — any letter of {G, Q, S}

Synopsis:

- The ShriMpCT function returns the minimal polynomial in x of the representation of the element e in the usual, generic or symmetric Hecke algebra for the shape $part$ (l is equal to Q, G or S).
- If $part$ is not specified then it returns the least common multiple of all minimal polynomials for the shapes of minimal weight.
- Whenever there is a conflict between the function name ShriMpCT and another name used in the same session, use the long form SHRI[ShriMpCT].

Examples:

```
> with(SHRI):
> ShriMpCT(a*[2,1]+b*[3,2,1],[1,2],x,Q);

                2        2      2  3                   2
           - a  q + a q  b - b  q  - a (q - 1) x + x

> ShriMpCT(a*[2,1],x,Q);

                              (a + x) (a q - x)
                              -----------------
                                 a (q + 1)

> ShriMpCT(q+[3,1,2],x,S);

        4          2    4    2            2        2        2
    (- v  - 1 + x v ) (v  - v  + 1 - 2 x v  + x + x ) (- 2 v  + x)
    --------------------------------------------------------------
                            4    2        4
                          (v  + v  + 1) (v  - 1)
```

See Also: `ShriCpCT`, `ShriCpT`, `ShriMpT`

5.11.8 `ShriMpT` — minimal polynomial for the tableaux

✓

Calling Sequence:

`ShriMpT`(e, x, l)

`ShriMpT`$(e, part, x, l)$

`SHRI[ShriMpT]`(e, x, l)

`SHRI[ShriMpT]`$(e, part, x, l)$

Parameters:

e	—	any element of the usual, generic or symmetric Hecke algebra
part	—	any (extra) list of positive integers in weakly decreasing order
x	—	any name or algebraic expression
l	—	any letter of {`G`, `Q`, `S`}

Synopsis:

- The `ShriMpT` function returns the minimal polynomial in x of the representation of the element e in the usual, generic or symmetric Hecke algebra for the shape *part* (l is equal to `Q`, `G` or `S`).
- If *part* is not specified then it returns the least common multiple of all minimal polynomials for the shapes of minimal weight.
- Whenever there is a conflict between the function name `ShriMpT` and another name used in the same session, use the long form `SHRI[ShriMpT]`.

Examples:

```
> with(SHRI):
> ShriMpT(a*[2,1]+b*[3,2,1],[2,1],x,Q);

                  2        2      2  3                   2
               - a  q + a q  b - b  q  - a (q - 1) x + x

> ShriMpT(a*[2,1],x,Q);

                              (a + x) (a q - x)
                              -----------------
                                 a (q + 1)

> ShriMpT(1-[2,1],x,G);

                       (- 1 + q1 + x) (- 1 + q2 + x)
                       -----------------------------
                                 q2 - q1
```

SEE ALSO: `ShriCpCT`, `ShriCpT`, `ShriMpCT`

5.11.9 `ShriRepCT` — representation for the contretableaux

✓

Calling Sequence:

`ShriRepCT`$(e, part)$

`ShriRepCT`$(e, part, l)$

`SHRI[ShriRepCT]`$(e, part)$

`SHRI[ShriRepCT]`$(e, part, l)$

Parameters:

e	—	any element of the usual, generic or symmetric Hecke algebra
$part$	—	any list of integers in weakly increasing order
l	—	any (extra) letter of $\{$`G`,`Q`,`S`$\}$

Synopsis:

- The `ShriRepCT` function returns the representation matrix of the element e in the usual, generic or symmetric Hecke algebra in the irreducible representation indexed by an increasing shape $part$ (l is equal to `Q, G` or `S`).

- If l is not specified then it means that l is equal to `Q`.

- Whenever there is a conflict between the function name `ShriRepCT` and another name used in the same session, use the long form `SHRI[ShriRepCT]`.

Examples:

```
> with(SHRI):
> ShriRepCT(a*[3,2,1]+b*[1],[1,1,2]);

                   [                                2     ]
                   [  b         0              - a q      ]
                   [                                      ]
                   [  0     - a q + b    - a q (- 1 + q) ]
                   [                                      ]
                   [ - a  - (- 1 + q) a   - a + a q + b  ]

> ShriRepCT([1,3,2]-1,[1,2],S);

                      [     2 1/2                    ]
                      [   (v )    + 1        2 3/2   ]
                      [ - -----------    - (v )      ]
                      [       2 1/2                  ]
                      [     (v )                     ]
                      [                              ]
                      [                        2 1/2 ]
                      [       0        - 1 + (v )    ]
```

SEE ALSO: ShriRepT

5.11.10 ShriRepT — representation for the tableaux

✓

Calling Sequence:

ShriRepT($e, part$)

ShriRepT($e, part, l$)

SHRI[ShriRepT]($e, part$)

SHRI[ShriRepT]($e, part, l$)

Parameters:

e — any element of the usual, generic or symmetric Hecke algebra

part — any list of integers in weakly decreasing order

l — any (extra) letter of {G, Q, S}

Synopsis:

- The ShriRepT function returns the representation matrix of the element e in the usual, generic or symmetric Hecke algebra in the irreducible representation indexed by a decreasing shape *part* (l is equal to Q, G or S).

- Whenever there is a conflict between the function name ShriRepT and another name used in the same session, use the long form SHRI[ShriRepT].

Examples:

```
> with(SHRI):
> ShriRepT(a*[2,1,3]+b*[1],[2,1,1]);

                  [                        3  ]
                  [ - a + b     0       a q   ]
                  [                           ]
                  [                         2 ]
                  [    0     - a + b   - a q  ]
                  [                           ]
                  [    0        0     a q + b ]

> ShriRepT([3,2,1]-1,[2,1,1],G);

              [                             3           ]
              [         3          3     q2  (q2 + q1) ]
              [ - 1 + q1       - q2      ------------- ]
              [                                q1       ]
              [                                         ]
              [                       2    2            ]
              [     0      - 1 - q1 q2   q2  (q2 + q1) ]
              [                                         ]
              [                  2                   2 ]
              [     0        - q1  q2     - 1 + q1 q2  ]
```

SEE ALSO: `ShriRepCT`

5.11.11 `ShriConvert` — isomorphisms

Calling Sequence:

`ShriConvert`(e, A, B)

`SHRI[ShriConvert]`(e, A, B)

Parameters:

e — any element of the usual, generic or symmetric Hecke algebra

A — any letter of $\{\mathtt{G}, \mathtt{Q}, \mathtt{S}\}$

B — any letter of $\{\mathtt{G}, \mathtt{Q}, \mathtt{S}\}$ minus $\{A\}$

Synopsis:

- The `ShriConvert` function returns the image of the element e of the usual, generic or symmetric Hecke algebra by isomorphism in another Hecke algebra.
- Whenever there is a conflict between the function name `ShriConvert` and another name used in the same session, use the long form `SHRI[ShriConvert]`.

Examples:

```
> ShriConvert([2,1]-q*[3,1,2],Q,S);
```

```
                    v [2, 1] - v  [3, 1, 2]
```

```
> ShriConvert([2,1]-q*[3,1,2],Q,G);

                      [2, 1]   q2 [3, 1, 2]
                    - ------ + ------------
                        q1            3
                                    q1
```

```
> ShriConvert(",G,S);

                                 4
                    v [2, 1] - v  [3, 1, 2]
```

SEE ALSO:

5.11.12 ShriJucis — computation of a Jucis-Murphy element

✓

Calling Sequence:

Jucis(i)

SHRI[Jucis](i)

Parameters:

i — any positive integer

Synopsis:

- The ShriJucis function computes the i-th Jucis-Murphy element inside the usual Hecke algebra of type A.
- More explicitly, denoting $T(i,j)$ the transposition exchanging i and j, we have: Jucis$(1) = 1$, Jucis$(2) = T(1,2)/q$, Jucis$(3) = T(1,3)/q^2 + T(2,3)/q$, and more generally:

$$\texttt{Jucis}(k) = T(1,k)/q^{(k-1)} + T(2,k)/q^{(k-2)} + \cdots + T(k-1,k)/q \quad .$$

- Whenever there is a conflict between the function name ShriJucis and another name used in the same session, use the long form SHRI[ShriJucis].

Examples:

```
> with(SHRI):
> ShriJucis(4);

              [4, 2, 3, 1]   [1, 4, 3, 2]   [1, 2, 4, 3]
              ------------ + ------------ + ------------
                    3              2              q
                   q              q
```

SEE ALSO: SG[Transpo]

5.12 SYMF to handle symmetric functions

✓

Calling Sequence:

SYMF[<function>]($args$)

<function>($args$)

Synopsis:

- This package provides functions to work with symmetric functions. It is related to the SF package of Stembridge.

- To get more information on symmetric functions do either ?TYP,Sf or ?TYP[Sf].

- All defined bases are stored in the global variable SYMFBases that is equal by default to the set {'e', 'h', 's', 'p', 'm', 'c'}.

- Whenever there is a conflict between a function name in SYMF and another name used in the same session, use the long form SYMF[<function>].

- The available functions are:

CCProd	Char2Sf	Conjugate	DiffSf
EvalSf	Hooks	ITensor	JtMatrix
ListPart	ListPartIn	LRrSf	Omega
MuirSf	PieriSf	Pletee	Pleteh
Plethe	Plethh	Plethysm	RedressSchur
ScalarSf	Sf2Char	Theta	Toc
Toe	Toh	Tom	Top
Tos	c2e	c2h	c2m
c2p	c2s	e2c	e2h
e2m	e2p	e2s	h2c
h2e	h2m	h2p	h2s
m2c	m2e	m2h	m2p
m2s	p2c	p2e	p2h
p2m	p2s	s2c	s2e
s2h	s2p	Zee	

- For help with a particular function do ?SYMF[<function>] or ?SYMF,<function> where <function> is one from the above list.

Examples:

```
> with(SYMF):
> PieriSf(e5, s[4,2,2,1,1]);

 s[4, 2, 2, 1, 1, 1, 1, 1, 1, 1] + s[4, 2, 2, 2, 1, 1, 1, 1, 1]

      + s[4, 2, 2, 2, 2, 1, 1, 1] + s[4, 3, 2, 1, 1, 1, 1, 1, 1]

      + s[4, 3, 2, 2, 1, 1, 1, 1] + s[4, 3, 2, 2, 2, 1, 1]

      + s[4, 3, 3, 1, 1, 1, 1, 1] + s[4, 3, 3, 2, 1, 1, 1]

      + s[4, 3, 3, 2, 2, 1] + s[5, 2, 2, 1, 1, 1, 1, 1, 1]

      + s[5, 2, 2, 2, 1, 1, 1, 1] + s[5, 2, 2, 2, 2, 1, 1]

      + s[5, 3, 2, 1, 1, 1, 1, 1] + s[5, 3, 2, 2, 1, 1, 1]

      + s[5, 3, 2, 2, 2, 1] + s[5, 3, 3, 1, 1, 1, 1]

      + s[5, 3, 3, 2, 2] + s[5, 3, 3, 2, 1, 1]
```

5.12.1 CCProd — compute the conjugacy class product of symmetric functions

Calling Sequence:

CCProd(sf_1, sf_2, b_1, b_2)

SYMF[CCProd](sf_1, sf_2, b_1, b_2)

Parameters:

sf_1, sf_2 — any symmetric functions

b_1, b_2 — any strings denoting names of known bases

Synopsis:

- The CCProd function computes the conjugacy class product of symmetric functions sf_1 and sf_2. It amounts, in the basis of Schur function, to compute the product $s_\lambda s_\mu = 1/f(\lambda)s_\lambda$ in which $f(\lambda)$ stands for the number of standard tableaux of shape λ and 0 otherwise.
- The result is expressed in the basis of Schur functions.
- In the case that sf_1 and sf_2 are solely expressed in the basis b_1 and b_2, one may specify these bases.
- Whenever there is a conflict between the function name CCProd and another name used in the same session, use the long form SYMF['CCProd'].

Examples:

```
> with(SYMF):
> CCProd(s^2*s[2,1,1] + s[3,1], q*s[2,1,1]);

                                      2
                                1/3 s   q s[2, 1, 1]
```

SEE ALSO:

5.12.2 `Char2Sf` — convert a virtual character into a symmetric function

Calling Sequence:

`Char2Sf`(*expr*)

`SYMF[Char2Sf]`(*expr*)

Parameters:

expr — any linear combination of `cl`[*part*] for any partition part

Synopsis:

- The `Char2Sf` function converts a virtual character into a symmetric function expressed on p-basis.
- We denote by `cl`[*part*], where *part* is a partition, the class function on the symmetric group that is evaluated to 1 for permutations with cycle-type *part*, and 0 otherwise.
- The `Char2Sf` function realizes the isomorphism between class functions on the symmetric group of degree n and homogeneous symmetric functions of degree n.
- Whenever there is a conflict between the function name `Char2Sf` and another name used in the same session, use the long form `SYMF['Char2Sf']`.

Examples:

```
> with(SYMF):
> Char2Sf(cl[5,3,1,1]);

                                                 2
                                  1/30 p5 p3 p1
```

SEE ALSO: `ITensor, Plethysm, Sf2Char, Zee`

5.12.3 `Conjugate` — conjugate a partition

Calling Sequence:

`Conjugate`(*part*)

`SYMF[Conjugate]`(*part*)

Parameters:

part — any list denoting a partition in (weakly) decreasing order

Synopsis:

- The `Conjugate` function calculates the conjugate (or transposed) partition of a partition *part*. Geometrically, conjugation amounts to reflect the diagram of the partition with respect to the main diagonal.
- Whenever there is a conflict between the function name `Conjugate` and another name used in the same session, use the long form `SYMF['Conjugate']`.

Examples:

```
> with(SYMF):
> Conjugate([4,2,2]);

                              [3, 3, 1, 1]
```

SEE ALSO:

5.12.4 `DiffSf` — differential operator

✘

Calling Sequence:

`DiffSf`(sf_1, sf_2)

`SYMF[DiffSf]`(sf_1, sf_2)

Parameters:

sf_1, sf_2 — any symmetric functions

Synopsis:

- The `DiffSf` function computes $D_{sf_1}(sf_2)$ in which D_{sf_1} stands for the differential operator corresponding to the symmetric function sf_1. Both symmetric functions sf_1 and sf_2 can be expressed in terms of any known basis since both expressions are converted in terms of Schur functions.
- The result is a linear combination of Schur functions, and not collected.
- Whenever there is a conflict between the function name `DiffSf` and another name used in the same session, use the long form `SYMF['DiffSf']`.

Examples:

```
> with(SYMF):
> DiffSf(s[3,1] + q*s[3], 2*e2*p2);

                           2 s[] + 2 q s[1]

> Tos(2*e2*p2);

                2 s[3, 1] - 2 s[1, 1, 1, 1] - 2 s[2, 2]

> DiffSf(s[4,3,2,1,1], s[4,3,2,1,1]);

                                 s[]
```

SEE ALSO: `Tos`

5.12.5 EvalSf — realize the action of symmetric functions on polynomials

Calling Sequence:

`EvalSf`$(sf, expr)$

`SYMF[EvalSf]`$(sf, expr)$

Parameters:

sf — any symmetric function

$expr$ — any expression depending on some variables

Synopsis:

- Symmetric functions can be considered as operators on polynomials. Let sf be a power-sum p_k, then:

$$\texttt{EvalSf}(p_k, c_u u + c_v v + \cdots)$$

is equal to

$$c_u u^k + c_v v^k + \cdots$$

where $u, v, \ldots$ are monomials and $c_u, c_v, \ldots$ are scalars.

- For a product of power-sums $sf = p_i p_j \ldots$, `EvalSf`$(sf, expr)$ is set to be equal to the product:

$$\texttt{EvalSf}(p_i, expr) \times \texttt{EvalSf}(p_j, expr) \times \cdots$$

The definition is extended by linearity to any symmetric function sf.

- Whenever there is a conflict between the function name `EvalSf` and another name used in the same session, use the long form `SYMF['EvalSf']`.

Examples:

```
> with(SYMF):
> EvalSf(p2, 3*x*y + 2*z);
```

```
                                  2  2      2
                               3 x  y  + 2 z
```

```
> EvalSf(p3*p4 + q*p2, 2*x + 3*y);
```

```
                     3      3     4      4          2      2
               (2 x  + 3 y ) (2 x  + 3 y ) + q (2 x  + 3 y )
```

See Also: Plethysm

5.12.6 Hooks — calculate the list of hook lengths

Calling Sequence:

Hooks(*part*)

SYMF[Hooks](*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The Hooks function computes the list of all hook lengths of a given partition. The result is sorted into non-increasing order.
- The arm and leg length of a cell (i, j) are respectively $part[i] - j$ and $cpart[j] - i$ where *cpart* is the conjugate of *part*.
- The call Hooks$(part, a)$ computes the Jack-hook-product. a may be any expression. This is the product

$$a \times \text{arm\_length}[i, j] + \text{leg\_length}[i, j] + 1$$

over all cells of the diagram of the partition *part*.

- The call Hooks$(part, q, t)$ computes the two-variable hook polynomial. The arguments q and t can be any expressions. This is the product

$$1 - q^{\text{arm\_length}[i,j]} t^{(1+\text{leg\_length}[i,j])}$$

over all cells of the diagram of the partition *part*.

- Whenever there is a conflict between the function name Hooks and another name used in the same session, use the long form SYMF['Hooks'].

Examples:

```
> with(SYMF):
> Hooks([3,1]);

                              [4, 2, 1, 1]

> 4!/convert(", *);

                                   3

> Hooks([5,4,3,2,1]);

           [9, 7, 7, 5, 5, 5, 3, 3, 3, 3, 1, 1, 1, 1, 1]

> Hooks([2,1], z);

                                 z + 2
```

```
> Hooks([2,1], z, t);

                                       2         2
                               (1 - z t ) (1 - t)
```

SEE ALSO: Conjugate, ListPart

5.12.7 ITensor — compute the inner tensor product of symmetric functions

Calling Sequence:

ITensor(sf_1, sf_2, b_1, b_2)

SYMF[ITensor](sf_1, sf_2, b_1, b_2)

Parameters:

sf_1, sf_2 — any symmetric functions

b_1, b_2 — any strings denoting names of known bases

Synopsis:

- The ITensor function computes the inner tensor product of symmetric functions sf_1 and sf_2. This operation is also called the internal product. Since sf_1 and sf_2 may be seen as virtual characters, ITensor computes the symmetric function associated with the product (Kronecker product) of these two virtual characters.
- For the basis of products of power-sums,

$$\texttt{ITensor}(\dots p_2^j p_1^i, \dots p_2^{j'} p_1^{i'})$$

is equal to 0 or Zee$([\dots, 2^j, 1^i]) \dots p_2^j p_1^i$ whether $(\dots, j, i) \neq (\dots, j', i')$ or not.
- The result is expressed in the p-basis.
- In the case that sf_1 and sf_2 are solely expressed in the basis b_1 and b_2, one may specify these bases.
- Whenever there is a conflict between the function name ITensor and another name used in the same session, use the long form SYMF['ITensor'].

Examples:

```
> with(SYMF):
> ITensor(p3*p3*p2*p1*p1, p3*p3*p2*p1*p1);

                                     2       2
                               72 p3   p2 p1

> Zee([3,3,2,1,1]);

                                     72

> ITensor(s[3,2,1], p3*h3, 's');
```

```
                  2             3
       - 2/3 p3   - 1/3 p3 p1
```

SEE ALSO: Char2Sf, Plethysm, Sf2Char

5.12.8 JtMatrix — compute the Jacobi-Trudi matrix

Calling Sequence:

JtMatrix($part_1, part_2$)

SYMF[JtMatrix]($part_1, part_2$)

Parameters:

$part_1, part_2$ — any lists denoting partitions

Synopsis:

- The JtMatrix function computes the Jacobi-Trudi matrix corresponding to the partition $part_1$ or to the skew-partition $part_1/part_2$. The determinant of this matrix is exactly the Schur function indexed by this partition.
- Whenever you use a third argument b, the entries of the matrix are the b_i's instead of the h_i's.
- Whenever there is a conflict between the function name JtMatrix and another name used in the same session, use the long form SYMF['JtMatrix'].

Examples:

```
> with(SYMF):
> JtMatrix([3,2,1,1]);
                              [ h3  h4  h5  h6 ]
                              [                ]
                              [ h1  h2  h3  h4 ]
                              [                ]
                              [  0   1  h1  h2 ]
                              [                ]
                              [  0   0   1  h1 ]

> JtMatrix([2,1], [1]);
                                  [ h1  h3 ]
                                  [        ]
                                  [  0  h1 ]

> JtMatrix([2,2,1], [1,1,1], 'b');

                                [ b1  b2  b3 ]
                                [            ]
                                [  1  b1  b2 ]
                                [            ]
                                [  0   0   1 ]
```

SEE ALSO:

5.12.9 `LRrSf` — Littlewood-Richardson rule

Calling Sequence:

`LRrSf`($schur_1, schur_2$)

`SYMF[LRrSf]`($schur_1, schur_2$)

Parameters:

$schur_1, schur_2$ — any Schur functions

Synopsis:

- The `LRrSf` function computes the product of the two Schur functions $schur_1$ and $schur_2$ using the Littlewood-Richardson rule. The result is expressed in the basis of Schur functions.
- Whenever there is a conflict between the function name `LRrSf` and another name used in the same session, use the long form `SYMF['LRrSf']`.

Examples:

```
> with(SYMF):
> LRrSf(s[4,2,1], s[2,2]);

s[4, 2, 2, 2, 1] + s[4, 3, 2, 1, 1] + s[4, 3, 2, 2] + s[4, 3, 3, 1]

 + s[4, 4, 2, 1] + s[4, 4, 3] + s[5, 2, 2, 1, 1] + s[5, 3, 1, 1, 1]

 + s[5, 2, 2, 2] + 2 s[5, 3, 2, 1] + s[5, 3, 3] + s[5, 4, 1, 1]

 + s[5, 4, 2] + s[6, 2, 2, 1] + s[6, 3, 1, 1] + s[6, 3, 2] + s[6, 4, 1]
```

SEE ALSO: `MuirSf`, `PieriSf`

5.12.10 `ListPartIn` — list of partitions contained into a given partition or between two given partitions

✓

Calling Sequence:

`ListPartIn`($part$)

`ListPartIn`($part, part_{in}$)

`SYMF[ListPart]`($part$)

`SYMF[ListPart]`($part, part_{in}$)

Parameters:

$part, part_{in}$ — any lists denoting partitions

Synopsis:

- The `ListPartIn` function gives all partitions contained into the diagram of a given partition. The list is ordered by increasing length.
- Called with a second argument $part_{in}$, the function returns the list of all partitions contained into the diagram of $part$ and containing the diagram of $part_{in}$.
- Whenever there is a conflict between the function name `ListPartIn` and another name used in the same session, use the long form `SYMF['ListPartIn']`.

Examples:

```
> with(SYMF):
> ListPartIn([3,1]);

                [[], [1], [2], [3], [1, 1], [2, 1], [3, 1]]

> ListPartIn([3,2,1], [2,1,1]);

                [[2, 1, 1], [2, 2, 1], [3, 1, 1], [3, 2, 1]]
```

SEE ALSO:

5.12.11 `ListPart` — list of partitions of a given weight

Calling Sequence:

`ListPart`(n, l)

`ListPart`(n, k, l)

`SYMF[ListPart]`(n, l)

`SYMF[ListPart]`(n, k, l)

Parameters:

n — any non negative integer

l — the maximum number of parts in each partition

k — the maximal part size in each partition

Synopsis:

- The `ListPart` function gives all partitions of n. A partition of n is a weakly decreasing list of positive integers, called parts, with sum n.
- When called with one argument, say n, the function returns the list of all partitions of n.
- One may fix the maximal number of parts of each partitions with a second argument, say l. In that case, it is also possible to fix the maximal part k in each partition. `ListPart`(n, k, n) will be the list of all partitions of n with maximal part k.
- Given a partition *part*, `convert`(*part*, `‘+‘`) gives its weight.
- Whenever there is a conflict between the function name `ListPart` and another name used in the same session, use the long form `SYMF['ListPart']`.

Examples:

```
> with(SYMF):
> ListPart(4);

             [[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

```
> ListPart(5, 3);

               [[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1]]

> ListPart(5, 3, 2);

                                   [[3, 2]]
```

SEE ALSO:

5.12.12 `MuirSf` — Muir's formula

Calling Sequence:

`MuirSf`$(mono, schur)$

`SYMF[MuirSf]`$(mono, schur)$

Parameters:

mono — any monomial symmetric function

schur — any Schur function

Synopsis:

- The `MuirSf` function computes the product of the monomial symmetric function $mono = m_{part}$ by the Schur function *schur*, using Muir formula. The result is expressed on the basis of Schur functions.

- Whenever there is a conflict between the function name `MuirSf` and another name used in the same session, use the long form `SYMF['MuirSf']`.

Examples:

```
> with(SYMF):
> MuirSf(m[2,1], s[4,2,2,1]);

     s[6, 3, 2, 1] - 2 s[4, 2, 2, 1, 1, 1, 1] + s[4, 4, 3, 1]

       - s[4, 2, 2, 2, 1, 1]  - s[5, 3, 3, 1] + s[5, 4, 2, 1]

       + s[6, 2, 2, 1, 1] + s[6, 2, 2, 2] - s[4, 3, 3, 1, 1]

       - s[4, 3, 3, 2] - s[5, 2, 2, 1, 1, 1]

       + s[4, 4, 2, 1, 1] + s[4, 4, 2, 2] + s[4, 2, 2, 2, 2]

       - s[4, 3, 2, 1, 1, 1]
```

SEE ALSO: `PieriSf`

5.12.13 Omega — apply the omega-automorphism

Calling Sequence:

Omega(sf)

SYMF[Omega](sf)

Parameters:

sf — any symmetric function

Synopsis:

- The Omega function applies the omega-automorphism to the symmetric function sf. This involution is defined as: $h_i \to e_i$, $e_i \to h_i$ and $p_i \to -p_i$ if i is even. In terms of Schur functions, the involution conjugates the indexing partition.
- Whenever there is a conflict between the function name Omega and another name used in the same session, use the long form SYMF['Omega'].

Examples:

```
> with(SYMF):
> Omega(h3);

                                    e3

> Omega(p4);

                                   - p4
```

SEE ALSO: Conjugate, Theta

5.12.14 PieriSf — Pieri's formula

Calling Sequence:

PieriSf($sf, schur$)

SYMF[PieriSf]($sf, schur$)

Parameters:

sf — any elementary or complete symmetric function

$schur$ — any Schur function

Synopsis:

- The PieriSf function computes the product of the elementary (resp. complete) symmetric function e_k (resp. h_k) by the Schur function $schur$, using Pieri formula. The result is expressed on the basis of Schur functions.
- Whenever there is a conflict between the function name PieriSf and another name used in the same session, use the long form SYMF['PieriSf'].

Examples:

```
> with(SYMF):
> PieriSf(e2, s[2,1]);

            s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 2] + s[3, 1, 1]

> PieriSf(h3, s[3,2,2,1]);

    s[6, 2, 2, 1] + s[5, 3, 2, 1] + s[5, 2, 2, 2] + s[4, 3, 2, 2]

         + s[5, 2, 2, 1, 1] + s[4, 3, 2, 1, 1] + s[3, 3, 2, 2, 1]

         + s[4, 2, 2, 2, 1]
```

SEE ALSO: MuirSf

5.12.15 Pletee — plethysm of elementary symmetric functions

Calling Sequence:

Pletee(e_1, e_2)

SYMF[Pletee](e_1, e_2)

Parameters:

e_1, e_2 — any elementary symmetric functions

Synopsis:

- The Pletee function computes the plethysm $e_1[e_2]$ where e_1 and e_2 are two elementary symmetric functions. This special plethysm is computed by using Newton's formula and Muir's formula and the result is given in the basis of Schur functions.
- Whenever there is a conflict between the function name Pletee and another name used in the same session, use the long form SYMF['Pletee'].

Examples:

```
> with(SYMF):
> Pletee(e4, e3);

 s[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] + s[2, 2, 1, 1, 1, 1, 1, 1, 1, 1]

    + s[2, 2, 2, 1, 1, 1, 1, 1, 1] + s[3, 3, 1, 1, 1, 1, 1, 1]

    + s[2, 2, 2, 2, 1, 1, 1, 1] + s[3, 3, 2, 1, 1, 1, 1]

    + s[3, 2, 2, 2, 1, 1, 1] + s[3, 3, 2, 2, 1, 1] + s[2, 2, 2, 2, 2, 2]

    + s[4, 4, 1, 1, 1, 1] + s[4, 3, 2, 2, 1] + s[3, 3, 3, 3]
```

SEE ALSO: Pleteh, Plethe, Plethh, Plethysm

5.12.16 Pleteh — plethysm of a complete symmetric function by an elementary symmetric function

✘

Calling Sequence:

Pleteh(e_1, h_2)

SYMF[Pleteh](e_1, h_2)

Parameters:

e_1 — any elementary symmetric function

h_2 — any complete symmetric function

Synopsis:

- The Pleteh function computes the plethysm $e_1[h_2]$ where e_1 is an elementary symmetric function and h_2 is a complete symmetric function. This special plethysm is computed by using Newton's formula and Muir's formula and the result is given in the basis of Schur functions.
- Whenever there is a conflict between the function name Pleteh and another name used in the same session, use the long form SYMF['Pleteh'].

Examples:

```
> with(SYMF):
> Pleteh(e4, h3);

    s[5, 3, 3, 1] + s[7, 3, 1, 1] + s[9, 1, 1, 1] + s[5, 5, 1, 1]

  + s[3, 3, 3, 3] + s[6, 4, 2] + s[6, 6] + s[7, 4, 1] + s[8, 3, 1]

  + s[6, 3, 3]
```

SEE ALSO: Pletee, Plethe, Plethh, Plethysm

5.12.17 Plethe — plethysm of an elementary symmetric function by a complete symmetric function

✘

Calling Sequence:

Plethe(h_1, e_2)

SYMF[Plethe](h_1, e_2)

Parameters:

h_1 — any complete symmetric function

e_2 — any elementary symmetric function

Synopsis:

- The Plethe function computes the plethysm $h_1[e_2]$ where h_1 is a complete symmetric function and e_2 is an elementary symmetric function. This special plethysm is computed by using Newton's formula and Muir's formula and the result is given in the basis of Schur functions.
- Whenever there is a conflict between the function name Plethe and another name used in the same session, use the long form SYMF['Plethe'].

Examples:

```
> with(SYMF):
> Plethe(h4, e3);

   s[4, 1, 1, 1, 1, 1, 1, 1, 1] + s[3, 2, 2, 1, 1, 1, 1, 1]

 + s[4, 2, 2, 1, 1, 1, 1] + s[3, 2, 2, 2, 1, 1, 1] + s[3, 3, 3, 1, 1, 1]

 + s[3, 3, 2, 2, 1, 1] + s[2, 2, 2, 2, 2, 2] + s[4, 3, 3, 1, 1]

 + s[4, 2, 2, 2, 2] + s[4, 4, 4]
```

SEE ALSO: `Pletee, Pleteh, Plethh, Plethysm`

5.12.18 `Plethh` — plethysm of complete symmetric functions

Calling Sequence:

`Plethh`(h_1, h_2)

`SYMF[Plethh]`(h_1, h_2)

Parameters:

h_1, h_2 — any complete symmetric functions

Synopsis:

- The `Plethh` function computes the plethysm $h_1[h_2]$ where h_1 and h_2 are two complete symmetric functions. This special plethysm is computed by using Newton's formula and Muir's formula and the result is given in the basis of Schur functions.
- Whenever there is a conflict betwhhn the function name `Plethh` and another name used in the same session, use the long form `SYMF['Plethh']`.

Examples:

```
> with(SYMF):
> Plethh(h4, h3);

   s[12] + s[10, 2] + s[9, 3] + s[8, 2, 2] + s[8, 4] + s[7, 3, 2]

 + s[7, 4, 1] + s[6, 4, 2] + s[6, 6] + s[6, 2, 2, 2] + s[5, 4, 2, 1]

 + s[4, 4, 4]
```

SEE ALSO: `Pletee, Pleteh, Plethe, Plethysm`

5.12.19 `Plethysm` — plethysm of symmetric functions

Calling Sequence:

`Plethysm`(sf_1, sf_2, b_1, b_2)

`SYMF[Plethysm]`(sf_1, sf_2, b_1, b_2)

Parameters:

sf_1, sf_2 — any symmetric functions

b_1, b_2 — any names of known bases

Synopsis:

- The `Plethysm` function computes the plethysm $sf_1[sf_2]$. One may specify that sf_1 and sf_2 are expressed on the bases b_1 and b_2.
- The plethysm operation is defined as follows: let p_i be the i-th power-sum symmetric function and sf_1, sf_2, sf_3 be any symmetric functions, then,
  - $(sf_1 + sf_2)[sf_3] = sf_1[sf_3] + sf_2[sf_3]$
  - $(sf_1 * sf_2)[sf_3] = sf_1[sf_3] * sf_2[sf_3]$
  - $sf_1[p_i] = p_i[sf_1]$
  - $p_i[p_j] = p_{i\times j}$
- The result is expressed on the p-basis.
- Whenever there is a conflict between the function name `Plethysm` and another name used in the same session, use the long form `SYMF['Plethysm']`.

Examples:

```
> with(SYMF):
> Plethysm(h1, h3, 'h', 'h');

                                                  3
                          1/3 p3 + 1/2 p2 p1 + 1/6 p1

> Plethysm(e1^2, s[2,1,1]);

       2   4              6          3   2             2             8
   1/32 p2  p1  - 1/16 p2 p1  + 1/16 p2  p1  - 1/8 p2 p1  p4 + 1/64 p1

                4              4         2             2
        + 1/16 p1  p4 + 1/64 p2  - 1/16 p2  p4 + 1/16 p4
```

SEE ALSO: `Pletee`, `Pleteh`, `Plethe`, `Plethh`

5.12.20 `RedressSchur` — reorder the index of Schur functions

Calling Sequence:

`RedressSchur`(sf)

`SYMF[RedressSchur]`(sf)

Parameters:

sf — any symmetric function

Synopsis:

- The `RedressSchur` function reorders the indices of all Schur functions appearing in an expression. This expression can involve other symmetric functions. Each single Schur function is redressed into s_{part}, $-s_{part}$ or 0.
- Whenever there is a conflict between the function name `RedressSchur` and another name used in the same session, use the long form `SYMF['RedressSchur']`.

Examples:

```
> with(SYMF):
> RedressSchur(h3*s[4,1]^2 - 2*a*s[4,1,2] + q*s[1,3]*s[0,5]);

                                 2
                     h3 s[4, 1]  + q s[2, 2] s[4, 1]
```

SEE ALSO:

5.12.21 `ScalarSf` — scalar product of symmetric functions

Calling Sequence:

`ScalarSf`(sf_1, sf_2)

`ScalarSf`(sf_1, sf_2, sfp)

`SYMF[ScalarSf]`(sf_1, sf_2)

`SYMF[ScalarSf]`(sf_1, sf_2, sfp)

Parameters:

sf_1, sf_2 — any symmetric functions

sfp — any procedure accepting a partition as input

Synopsis:

- The `ScalarSf` function computes the scalar product between the symmetric functions sf_1 and sf_2.
- This scalar product is such that Schur functions constitute an orthonormal basis.
- The p-basis is an orthogonal basis. Let

$$sf = \dots p_3^i p_2^j p_1^k,$$

  then `ScalarSf`$(sf, sf) =$ `Zee`$([\dots, 3^i, 2^j, 1^k])$.
- One can choose another scalar product, for which the p-basis will still remain orthogonal by replacing `Zee` with any procedure sfp accepting partitions.
- One may specify that sf_1 is expressed on basis b_1 and sf_2 is expressed on basis b_2 by adding two arguments as follows:
  - `ScalarSf`(sf_1, sf_2, b_1, b_2)

- `ScalarSf`$(sf_1, sf_2, b_1, b_2, sfp)$

- Whenever there is a conflict between the function name `ScalarSf` and another name used in the same session, use the long form `SYMF['ScalarSf']`.

Examples:

```
> with(SYMF):
> ScalarSf(p3^2*p1, p3^2*p1, 'p', 'p');

                                   18

> Zee([3,3,1]);

                                   18
```

SEE ALSO: `Zee`

5.12.22 `Sf2Char` — convert a symmetric function into a virtual character

Calling Sequence:

`Sf2Char`(sf)

`SYMF[Sf2Char]`(sf)

Parameters:

sf — any symmetric function

Synopsis:

- The `Sf2Char` function converts a symmetric function into a virtual character. This function is the inverse of `Char2Sf`.

- The `Sf2Char` function realizes the isomorphism between the linear space of homogeneous symmetric functions of degree n and class functions on the symmetric group of degree n (Frobenius isomorphism).

- The result is a linear combination of `cl`$[part]$ where $part$ is a partition.

- Whenever there is a conflict between the function name `Sf2Char` and another name used in the same session, use the long form `SYMF['Sf2Char']`.

Examples:

```
> with(SYMF):
> Sf2Char(s[3,1,1]);

             - 2 cl[2, 2, 1] + 6 cl[1, 1, 1, 1, 1] + cl[5]

> Sf2Char(p3*s[3] - e4);

      3 cl[3, 2, 1] - cl[1, 1, 1, 1] + cl[4] - cl[3, 1] + cl[2, 1, 1]

             - cl[2, 2] + 6 cl[3, 3] + 3 cl[3, 1, 1, 1]
```

SEE ALSO: `Char2Sf`, `ITensor`, `Plethysm`, `Zee`

5.12.23 Theta — apply the θ-automorphism

Calling Sequence:

`Theta`(sf, a)

`Theta`(sf, q, t)

`SYMF[Theta]`(sf, a)

`SYMF[Theta]`(sf, q, t)

Parameters:

sf — any symmetric function

a, q, t — any names or expressions

Synopsis:

- The `Theta` function realizes a certain multiplicative automorphism of the ring of symmetric functions. It is defined on power-sum functions as follows: `Theta`(sf, a) gives the image of sf under the transformation $p_i \to a \times p_i$.
- `Theta`(sf, q, t) is the image of sf under the transformation: $p_i \to \frac{(1-q^i)}{(1-t^i)} p_i$.
- The result is given in the p-basis.
- Whenever there is a conflict between the function name `Theta` and another name used in the same session, use the long form `SYMF['Theta']`.

Examples:

```
> with(SYMF):
> Theta(s[4,1], a);

       3        2          4       3           5    5                      2
  1/6 a  p3 p1  + 1/6 a  p2 p1  + 1/30 a  p1  - 1/5 a p5 - 1/6 a  p3 p2

> Theta(p3, q, t);

                                              3
                                        (1 - q ) p3
                                      - -----------
                                                 3
                                          - 1 + t

> Theta(s[2,1], q, t);

                                 3   3               3
                         (- 1 + q)  p1        (- 1 + q ) p3
                     1/3 -------------- - 1/3 -------------
                                     3                   3
                           (- 1 + t)            - 1 + t
```

SEE ALSO: `Omega`

5.12.24 Toc — express any symmetric function in the basis of cycle indices

Calling Sequence:

Toc(sf)

Toc(sf, b)

SYMF[Toc](sf)

SYMF[Toc](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Toc function expresses any symmetric function in the basis of cycle indices, *i.e.* as a linear combination of cycle indices c[$part$] of conjugacy classes corresponding to the partition $part = [1^{m_1}, 2^{m_2}, \ldots]$:
$$\mathtt{c}[part] = \mathtt{CCCard}(part)\, p_1^{m_1}\, p_2^{m_2} \ldots$$
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Toc($sf,' c'$) does not affect the argument sf.
- One may add $'noexpand'$ just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: Toc($sf, b,' collect'$). For instance, Toc($sf,' c',' collect'$) may be used to collect the argument sf.
- Whenever there is a conflict between the function name Toc and another name used in the same session, use the long form SYMF['Toc'].

Examples:

```
> with(SYMF):
> Toc((1+q)^500*e3*e4):                    # expands the input
> Toc((1+q)^500*e3*e4,noexpand):           # does not expand (1+q)^500
> Toc((1+q)^500*e3*e4,collect):            # collects the result
> Toc((1+q)^500*e3*e4,noexpand,'e'):       # the most efficient
> Toc((1+q)^500*e3*e4,'e',collect):        # specifies a basis
> Toc((1+q)^500*e3*e4,noexpand,'e',collect):
> Toc(c[2,1]*h4 - 2*q*p3);

     1/840 c[4, 2, 1] + 1/420 c[3, 2, 1, 1] + 1/280 c[2, 2, 2, 1]

     + 1/140 c[2, 2, 1, 1, 1] + 1/168 c[2, 1, 1, 1, 1, 1] - c[3] q
```

SEE ALSO: Toe, Toh, Tom, Top, Tos

5.12.25 `Toe` — convert any symmetric function to a e-polynomial

Calling Sequence:

`Toe`(sf)

`Toe`(sf, b)

`SYMF[Toe]`(sf)

`SYMF[Toe]`(sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The `Toe` function converts any symmetric function to a e-polynomial.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call `Toe`(sf, `'e'`) does not affect the argument sf.
- One may add `'noexpand'` just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by `Toe`(sf, b, `'collect'`). For instance, `Toe`(sf, `'e', 'collect'`) may be used to collect the argument sf.
- Whenever there is a conflict between the function name `Toe` and another name used in the same session, use the long form `SYMF['Toe']`.

Examples:

```
> with(SYMF):
> Toe((1+q)^500*h3*h4):                          # expands the input
> Toe((1+q)^500*h3*h4,noexpand):                 # does not expand (1+q)^500
> Toe((1+q)^500*h3*h4,collect):                  # collects the result
> Toe((1+q)^500*h3*h4,noexpand,'h'):             # the most efficient
> Toe((1+q)^500*h3*h4,'h',collect):              # specifies a basis
> Toe((1+q)^500*h3*h4,noexpand,'h',collect):
> Toe(s[2,1]*h4 - 2*q*p3);

                              2          2        3         2          2   3
 - e4 e2 e1 + e4 e3 + 5 e3 e1  e2 - 2 e3  e1 + e2  e1 - e2  e3 - 3 e2  e1

              5        4                                  3
          + e1  e2 - e1  e3 - 6 q e3 + 6 q e2 e1 - 2 q e1
```

SEE ALSO: `Toc, Toh, Top, Tos`

5.12.26 `Toh` — convert any symmetric function to a h-polynomial

Calling Sequence:

`Toh`(sf)

`Toh`(sf, b)

`SYMF[Toh]`(sf)

`SYMF[Toh]`(sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The `Toh` function converts any symmetric function to a h-polynomial.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call `Toh`(sf, `'h'`) does not affect the argument sf.
- One may add `'noexpand'` just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by `Toh`(sf, b, `'collect'`). For instance, `Toh`(sf, `'h', 'collect'`) may be used to collect the argument sf.
- Whenever there is a conflict between the function name `Toh` and another name used in the same session, use the long form `SYMF['Toh']`.

Examples:

```
> with(SYMF):
> Toh((1+q)^500*e3*e4):                      # expands the input
> Toh((1+q)^500*e3*e4,noexpand):             # does not expand (1+q)^500
> Toh((1+q)^500*e3*e4,collect):              # collects the result
> Toh((1+q)^500*e3*e4,noexpand,'e'):         # the most efficient
> Toh((1+q)^500*e3*e4,'e',collect):          # specifies a basis
> Toh((1+q)^500*e3*e4,noexpand,'e',collect):
> Toh(s[2,1]*e4 - 2*q*p3);

                             2          2        3        2          2   3
- h4 h2 h1 + h4 h3 + 5 h3 h1  h2 - 2 h3  h1 + h2  h1 - h2  h3 - 3 h2  h1

                 5       4                                 3
             + h1  h2 - h1  h3 - 6 q h3 + 6 q h2 h1 - 2 q h1
```

SEE ALSO: `Toc, Toe, Top, Tos`

5.12.27 Tom — express any symmetric function in the basis of basis of monomial symmetric functions

✘

Calling Sequence:

Tom(sf)

Tom(sf, b)

SYMF[Tom](sf)

SYMF[Tom](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The `Tom` function expresses any symmetric function in the basis of monomial symmetric functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Tom($sf,'m'$) does not affect the argument sf.
- One may add `'noexpand'` just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: Tom($sf, b,$'collect'). For instance, Tom($sf,'m',$'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name `Tom` and another name used in the same session, use the long form `SYMF['Tom']`.

Examples:

```
> with(SYMF):
> Tom((1+q)^500*e3*e4):                        # expands the input
> Tom((1+q)^500*e3*e4,noexpand):               # does not expand (1+q)^500
> Tom((1+q)^500*e3*e4,collect):                # collects the result
> Tom((1+q)^500*e3*e4,noexpand,'e'):           # the most efficient
> Tom((1+q)^500*e3*e4,'e',collect):            # specifies a basis
> Tom((1+q)^500*e3*e4,noexpand,'e',collect):
> Tom(s[2,1]*h4 - 2*q*p3);

   8 m[3, 2, 2] + m[6, 1] + 2 m[5, 2] + 4 m[5, 1, 1] + 2 m[4, 3]

  + 6 m[4, 2, 1] + 11 m[4, 1, 1, 1] + 6 m[3, 3, 1] + 14 m[3, 2, 1, 1]

  + 24 m[3, 1, 1, 1, 1] + 17 m[2, 2, 2, 1] + 28 m[2, 2, 1, 1, 1]

  + 45 m[2, 1, 1, 1, 1, 1] + 70 m[1, 1, 1, 1, 1, 1, 1] - 2 q m[3]
```

See Also: Toc, Toe, Toh, Top, Tos

5.12.28 Top — convert any symmetric function to a p-polynomial

Calling Sequence:

Top(sf)

Top(sf, b)

SYMF[Top](sf)

SYMF[Top](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Top function converts any symmetric function to a p-polynomial.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Top(sf, 'p') does not affect the argument sf.
- One may add 'noexpand' just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by Top(sf, b, 'collect'). For instance, Top(sf, 'p', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Top and another name used in the same session, use the long form SYMF['Top'].

Examples:

```
> with(SYMF):
> Top((1+q)^500*e3*e4):                     # expands the input
> Top((1+q)^500*e3*e4,noexpand):            # does not expand (1+q)^500
> Top((1+q)^500*e3*e4,collect):             # collects the result
> Top((1+q)^500*e3*e4,noexpand,'p'):        # the most efficient
> Top((1+q)^500*e3*e4,'p',collect):         # specifies a basis
> Top((1+q)^500*e3*e4,noexpand,'p',collect):
> Top(s[2,1]*h3 - 2*q*e3);

          3          2            4                          6           3
 1/18 p3 p1  - 1/9 p3  + 1/6 p2 p1  - 1/6 p2 p1 p3 + 1/18 p1  - 1/3 q p1

                        + q p2 p1 - 2/3 q p3
```

SEE ALSO: Toc, Toe, Toh, Tos

5.12.29 Tos — express any symmetric function in the basis of Schur functions

Calling Sequence:

Tos(sf)

Tos(sf, b)

SYMF[Tos](sf)

SYMF[Tos](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Tos function expresses any symmetric function in the basis of Schur functions, *i.e.* as a linear combination of Schur functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Tos(sf, 's') does not affect the argument sf.
- One may add 'noexpand' just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by Tos(sf, b, 'collect'). For instance, Tos(sf, 's', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Tos and another name used in the same session, use the long form SYMF['Tos'].

Examples:

```
> with(SYMF):
> Tos((1+q)^500*e3*e4):                         # expands the input
> Tos((1+q)^500*e3*e4,noexpand):                # does not expand (1+q)^500
> Tos((1+q)^500*e3*e4,collect):                 # collects the result
> Tos((1+q)^500*e3*e4,noexpand,'e'):            # the most efficient
> Tos((1+q)^500*e3*e4,'e',collect):             # specifies a basis
> Tos((1+q)^500*e3*e4,noexpand,'e',collect):
> Tos(s[2,1]*h4 - 2*q*p3);

             s[4, 2, 1] + s[5, 1, 1] + s[5, 2] + s[6, 1]

                 - 2 (s[3] - s[2, 1] + s[1, 1, 1]) q
```

SEE ALSO: Toc, Toe, Toh, Top

5.12.30 Zee — compute the scalar product of a p-function with itself

Calling Sequence:

Zee(*part*)

SYMF[Zee](*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- Let $part = [\ldots 2^i 1^j]$ and $p = \ldots p_2^i p_1^j$, then Zee(*part*)=ScalarSf(p,p).
- The argument must be a partition, that is a weakly decreasing list of positive integers.
- Whenever there is a conflict between the function name Zee and another name used in the same session, use the long form SYMF['Zee'].

Examples:

```
> with(SYMF):
> Zee([4,3,1,1]);

                                    24
```

SEE ALSO:

5.12.31 c2e — convert a monomial in the cycle indices to the e-basis

Calling Sequence:

c2e(*sf*)

SYMF[c2e](*sf*)

Parameters:

sf — any product of cycle indices

Synopsis:

- The c2e function converts a monomial in the cycle indices to an element expressed in the e-basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name c2e and another name used in the same session, use the long form SYMF['c2e'].

Examples:

```
> with(SYMF):
> c2e(c[2,1]*c[3,1]);

                  2                2   3                5          4            7
       - 144 e2 e1   e3 + 144 e2   e1   - 120 e2 e1   + 72 e1   e3 + 24 e1
```

SEE ALSO: e2c

5.12.32 c2h — convert a monomial in the cycle indices to the h-basis

Calling Sequence:

c2h(sf)

SYMF[c2h](sf)

Parameters:

sf — any product of cycle indices

Synopsis:

- The c2h function converts a monomial in the cycle indices to an element expressed in the h-basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name c2h and another name used in the same session, use the long form SYMF['c2h'].

Examples:

```
> with(SYMF):
> c2h(c[2,1]*c[3,1]);

            2                3   2          5           4           7
      144 h1  h2 h3 - 144 h1  h2  + 120 h1  h2 - 72 h1  h3 - 24 h1
```

See Also: h2c

5.12.33 c2m — convert a monomial in the cycle indices to the monomial symmetric function basis

Calling Sequence:

c2m(sf)

SYMF[c2m](sf)

Parameters:

sf — any product of cycle indices

Synopsis:

- The c2m function converts a monomial in the cycle indices to an element expressed in the monomial symmetric function basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name c2m and another name used in the same session, use the long form SYMF['c2m'].

Examples:

```
> with(SYMF):
> c2m(c[2,1]*c[3,1]);

      24 m[7] + 72 m[4, 3] + 48 m[6, 1] + 96 m[3, 3, 1] + 48 m[5, 2]

   + 48 m[3, 2, 2] + 48 m[5, 1, 1] + 48 m[4, 2, 1] + 48 m[3, 2, 1, 1]
```

SEE ALSO: m2c

5.12.34 c2p — convert a monomial in the cycle indices to the p-basis

Calling Sequence:

c2p(sf)

SYMF[c2p](sf)

Parameters:

sf — any product of cycle indices

Synopsis:

- The c2p function converts a monomial in the cycle indices to an element expressed in the p-basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name c2p and another name used in the same session, use the long form SYMF['c2p'].

Examples:

```
> with(SYMF):
> c2p(c[2,1]*c[3,1]);

                         2
                 24 p2 p1  p3
```

SEE ALSO: p2c

5.12.35 c2s — convert a monomial in the cycle indices to the Schur basis

Calling Sequence:

c2s(sf)

SYMF[c2s](sf)

Parameters:

sf — any product of cycle indices

Synopsis:

- The c2s function converts a monomial in the cycle indices to an element expressed in the Schur basis.

- This function has the option remember set.
- Whenever there is a conflict between the function name c2s and another name used in the same session, use the long form SYMF['c2s'].

Examples:

```
> with(SYMF):
> c2s(c[2,1]*c[3,1]);

   24 s[7] + 24 s[6, 1] + 24 s[4, 3] - 24 s[5, 1, 1] - 24 s[4, 2, 1]

 - 24 s[3, 2, 2] + 24 s[3, 3, 1] + 24 s[3, 2, 1, 1] + 24 s[3, 1, 1, 1, 1]

 - 24 s[2, 2, 2, 1] -  24 s[2, 1, 1, 1, 1, 1] - 24 s[1, 1, 1, 1, 1, 1, 1]
```

SEE ALSO: s2c

5.12.36 e2c — convert an element of the e-basis to the cycle index basis

Calling Sequence:

e2c(sf)

SYMF[e2c](sf)

Parameters:

sf — an element of the e-basis

Synopsis:

- The e2c function converts an element of the e-basis into an element expressed in the cycle index basis.
- The input of such a function is a monomial in the e-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name e2c and another name used in the same session, use the long form SYMF['e2c'].

Examples:

```
> with(SYMF):
> e2c(e2*e1);

                       1/2 c[1, 1, 1] - 1/6 c[2, 1]
```

SEE ALSO: c2e

5.12.37 e2h — convert an element of the e-basis to the h-basis

Calling Sequence:

e2h(sf)

SYMF[e2h](sf)

Parameters:

sf — an element in the e-basis

Synopsis:

- The e2h function converts an element of the e-basis into an element expressed in the h-basis using Wronski formula.
- The input of such a function is a monomial in the e-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name e2h and another name used in the same session, use the long form SYMF['e2h'].

Examples:

```
> with(SYMF):
> e2h(e1*e2*e3);

                                 2   2           4      3         6
               - h1 h2 h3 + 2 h2   h1   - 3 h2 h1   + h1   h3 + h1
```

SEE ALSO: h2e

5.12.38 e2m — convert an element of the e-basis to monomial symmetric function basis

Calling Sequence:

e2m(sf)

SYMF[e2m](sf)

Parameters:

sf — an element of the e-basis

Synopsis:

- The e2m function converts an element of the e-basis into an element expressed in the basis of monomial symmetric functions.
- The input of such a function is a monomial in the e-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name e2m and another name used in the same session, use the long form SYMF['e2m'].

Examples:

```
> with(SYMF):
> e2m(e3*e4);

        m[2, 2, 2, 1] + 3 m[2, 2, 1, 1, 1] + 10 m[2, 1, 1, 1, 1, 1]

             + 35 m[1, 1, 1, 1, 1, 1, 1]
```

SEE ALSO: `m2e`

5.12.39 e2p — convert an element of the e-basis to the p-basis

Calling Sequence:

e2p(sf)

SYMF[e2p](sf)

Parameters:

sf — an element of the e-basis

Synopsis:

- The `e2p` function converts an element of the e-basis into an element expressed in the p-basis using Newton formula.
- The input of such a function is a monomial in the e-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name `e2p` and another name used in the same session, use the long form `SYMF['e2p']`.

Examples:

```
> with(SYMF):
> e2p(e3*e4);

          7           5             4             3   2          3
  1/144 p1  - 1/16 p1  p2 + 5/72 p1  p3 + 7/48 p1  p2  - 1/24 p1  p4

           2            3                            2                2
 - 1/4 p2 p1  p3 - 1/16 p2  p1 + 1/8 p2 p1 p4 + 1/9 p3  p1 + 1/24 p3 p2

 - 1/12 p3 p4
```

SEE ALSO: `p2e`

5.12.40 e2s — convert an element of the e-basis to the Schur basis

Calling Sequence:

e2s(sf)

SYMF[e2s](sf)

Parameters:

sf — an element of the e-basis

Synopsis:

- The e2s function converts an element of the e-basis into an element expressed in the basis of Schur functions using Pieri formula.
- The input of such a function is a monomial in the e-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name e2s and another name used in the same session, use the long form SYMF['e2s'].

Examples:

```
> with(SYMF):
> e2s(e3*e4);

    s[1, 1, 1, 1, 1, 1, 1] + s[2, 1, 1, 1, 1, 1] + s[2, 2, 1, 1, 1]

                             + s[2, 2, 2, 1]
```

SEE ALSO: s2e

5.12.41 h2c — convert an element of the h-basis to the cycle index basis

Calling Sequence:

h2c(sf)

SYMF[h2c](sf)

Parameters:

sf — an element of the h-basis

Synopsis:

- The h2c function converts an element of the h-basis into an element expressed in the cycle index basis.
- The input of such a function is a monomial in the h-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name h2c and another name used in the same session, use the long form SYMF['h2c'].

Examples:

```
> with(SYMF):
> h2c(h2*h1);

                       1/6 c[2, 1] + 1/2 c[1, 1, 1]
```

See Also: `c2h`

5.12.42 `h2e` — convert an element of the h-basis to the e-basis

Calling Sequence:

`h2e`(sf)

`SYMF[h2e]`(sf)

Parameters:

sf — an element of the h-basis

Synopsis:

- The `h2e` function converts an element of the h-basis into an element expressed in the e-basis using Wronski formula.
- The input of such a function is a monomial in the h-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name `h2e` and another name used in the same session, use the long form `SYMF['h2e']`.

Examples:

```
> with(SYMF):
> h2e(h1*h2*h3);

                           2   2           4     3        6
            - e1 e2 e3 + 2 e2  e1  - 3 e2 e1  + e1  e3 + e1
```

See Also: `e2h`

5.12.43 `h2m` — convert an element of the h-basis to monomial symmetric function basis

Calling Sequence:

`h2m`(sf)

`SYMF[h2m]`(sf)

Parameters:

sf — an element of the h-basis

Synopsis:

- The `h2m` function converts an element of the h-basis into an element expressed in the basis of monomial symmetric functions.
- The input of such a function is a monomial in the h-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name `h2m` and another name used in the same session, use the long form `SYMF['h2m']`.

Examples:

```
> with(SYMF):
> h2m(h3*h4);

  m[7] + 2 m[6, 1] + 3 m[5, 2] + 4 m[5, 1, 1] + 4 m[4, 3] + 6 m[4, 2, 1]

       + 8 m[4, 1, 1, 1] + 7 m[3, 3, 1] + 8 m[3, 2, 2] + 11 m[3, 2, 1, 1]

       + 15 m[3, 1, 1, 1, 1] + 13 m[2, 2, 2, 1] + 18 m[2, 2, 1, 1, 1]

       + 25 m[2, 1, 1, 1, 1, 1] + 35 m[1, 1, 1, 1, 1, 1, 1]
```

SEE ALSO: `m2h`

5.12.44 h2p — convert an element of the h-basis to the p-basis

Calling Sequence:

`h2p`(sf)

`SYMF[h2p]`(sf)

Parameters:

sf — an element of the h-basis

Synopsis:

- The `h2p` function converts an element of the h-basis into an element expressed in the p-basis using Crochin formula.
- The input of such a function is a monomial in the h-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name `h2p` and another name used in the same session, use the long form `SYMF['h2p']`.

Examples:

```
> with(SYMF):
> h2p(h2*h3);

                                2              3          2               5
      1/6 p2 p3 + 1/4 p2  p1 + 1/3 p2 p1  + 1/6 p1  p3 + 1/12 p1
```

SEE ALSO: `p2h`

5.12.45 h2s — convert an element of the h-basis to the Schur basis

Calling Sequence:

h2s(sf)

SYMF[h2s](sf)

Parameters:

sf — an element of the h-basis

Synopsis:

- The h2s function converts an element of the h-basis into an element expressed in the basis of Schur functions using Pieri formula.
- The input of such a function is a monomial in the h-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name h2s and another name used in the same session, use the long form SYMF['h2s'].

Examples:

```
> with(SYMF):
> h2s(h2*h4);

                        s[6] + s[5, 1] + s[4, 2]
```

SEE ALSO: s2h

5.12.46 m2c — convert a monomial in the monomial symmetric functions to the cycle index basis

Calling Sequence:

m2c(sf)

SYMF[m2c](sf)

Parameters:

sf — any product of monomial symmetric functions

Synopsis:

- The m2c function converts a monomial in the monomial symmetric functions to an element expressed in the cycle index basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name m2c and another name used in the same session, use the long form SYMF['m2c'].

Examples:

```
> with(SYMF):
> m2c(m[2,1]*m[3,1]);

      1/420 c[3, 2, 1, 1] - 1/630 c[4, 2, 1] - 1/280 c[3, 3, 1] + 1/420 c[4, 3]
```

SEE ALSO: `c2m`

5.12.47 `m2e` — convert a monomial in the monomial symmetric functions to the e-basis

Calling Sequence:

`m2e`(sf)

`SYMF[m2e]`(sf)

Parameters:

sf — any product of monomial symmetric functions

Synopsis:

- The `m2e` function converts a monomial in the monomial symmetric functions to an element expressed in the e-basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name `m2e` and another name used in the same session, use the long form `SYMF['m2e']`.

Examples:

```
> with(SYMF):
> m2e(m[2,1]*m[1,1]);

                                                  2
                                   - 3 e2 e3 + e2  e1
```

SEE ALSO: `e2m`

5.12.48 `m2h` — convert a monomial in the monomial symmetric functions to the h-basis

Calling Sequence:

`m2h`(sf)

`SYMF[m2h]`(sf)

Parameters:

sf — any product of monomial symmetric functions

Synopsis:

- The `m2h` function converts a monomial in the monomial symmetric functions to an element expressed in the h-basis.

- This function has the option remember set.
- Whenever there is a conflict between the function name m2h and another name used in the same session, use the long form SYMF['m2h'].

Examples:

```
> with(SYMF):
> m2h(m[2,1]*m[1,1]);
                          2              3       2          5
          3 h2 h3 - 5 h2  h1 + 7 h2 h1  - 3 h1  h3 - 2 h1
```

SEE ALSO: h2m

5.12.49 m2p — convert a monomial in the monomial symmetric functions to the p-basis

Calling Sequence:

m2p(sf)

SYMF[m2p](sf)

Parameters:

sf — any product of monomial symmetric functions

Synopsis:

- The m2p function converts a monomial in the monomial symmetric functions to an element expressed in the p-basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name m2p and another name used in the same session, use the long form SYMF['m2p'].

Examples:

```
> with(SYMF):
> m2p(m[2,1]*m[3,1]);
                       2                    2
               p2 p1  p3 - p2 p1 p4 - p3  p1 + p3 p4
```

SEE ALSO: p2m

5.12.50 m2s — convert a monomial in the monomial symmetric functions to the Schur basis

Calling Sequence:

m2s(sf)

SYMF[m2s](sf)

Parameters:

sf — any product of monomial symmetric functions

Synopsis:

- The `m2s` function converts a monomial in the monomial symmetric functions to an element expressed in the Schur basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name `m2s` and another name used in the same session, use the long form `SYMF['m2s']`.

Examples:

```
> with(SYMF):
> m2s(m[2,1]*m[1,1]);

  s[3, 2] + s[3, 1, 1] - s[2, 2, 1] - s[2, 1, 1, 1] - 2 s[1, 1, 1, 1, 1]
```

SEE ALSO: `s2m`

5.12.51 p2c — convert an element of the p-basis to the cycle index basis ✘

Calling Sequence:

p2c(sf)

SYMF[p2c](sf)

Parameters:

sf — an element of the p-basis

Synopsis:

- The `p2c` function converts an element of the p-basis into an element expressed in the cycle index basis.
- The input of such a function is a monomial in the p-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name `p2c` and another name used in the same session, use the long form `SYMF['p2c']`.

Examples:

```
> with(SYMF):
> p2c(p2^2*p3);

                              1/210 c[3, 2, 2]
```

SEE ALSO: `c2p`

5.12.52 p2e — convert an element of the p-basis to the e-basis

Calling Sequence:

`p2e`(sf)

`SYMF[p2e]`(sf)

Parameters:

sf — an element of the p-basis

Synopsis:

- The `p2e` function converts an element of the p-basis into an element expressed in the e-basis using Newton formula.
- The input of such a function is a monomial in the p-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name `p2e` and another name used in the same session, use the long form `SYMF['p2e']`.

Examples:

```
> with(SYMF):
> p2e(p2*p3);

                              2               3        2         5
            - 6 e2 e3 + 6 e2  e1 - 5 e2 e1  + 3 e1  e3 + e1
```

SEE ALSO: `e2p`

5.12.53 p2h — convert an element of the p-basis to the h-basis

Calling Sequence:

`p2h`(sf)

`SYMF[p2h]`(sf)

Parameters:

sf — an element of the p-basis

Synopsis:

- The `p2h` function converts an element of the p-basis into an element expressed in the h-basis using Crochin formula.
- The input of such a function is a monomial in the p-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name `p2h` and another name used in the same session, use the long form `SYMF['p2h']`.

Examples:

```
> with(SYMF):
> p2h(p2*p3*p4);

         5         2   3           6         3            4            3   3
 - 4 h4 h1  + 12 h3  h1  + 7 h3 h1  - 12 h2  h3 + 12 h2  h1 - 34 h2  h1

       2   5           7         2                  2   2            4
 + 28 h2  h1  - 9 h2 h1  - 24 h3  h1 h2 + 54 h3 h1  h2  - 38 h3 h1  h2

                             2           2                  3     9
 + 24 h4 h3 h2 - 12 h4 h3 h1  - 24 h4 h2  h1 + 20 h4 h2 h1  + h1
```

SEE ALSO: h2p

5.12.54 p2m — convert an element of the p-basis to monomial symmetric function basis ✘

Calling Sequence:

p2m(sf)

SYMF[p2m](sf)

Parameters:

sf — an element of the p-basis

Synopsis:

- The p2m function converts an element of the p-basis into an element expressed in the basis of monomial symmetric functions.
- The input of such a function is a monomial in the p-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name p2m and another name used in the same session, use the long form SYMF['p2m'].

Examples:

```
> with(SYMF):
> p2m(p1^2*p2);

                  m[4] + 2 m[2, 2] + 2 m[3, 1] + 2 m[2, 1, 1]
```

SEE ALSO: s2p

5.12.55 p2s — convert an element of the p-basis to the Schur basis

Calling Sequence:

p2s(sf)

SYMF[p2s](sf)

Parameters:

sf — an element of the p-basis

Synopsis:

- The p2s function converts an element of the p-basis into an element expressed in the basis of Schur functions using Muir formula.
- The input of such a function is a monomial in the p-generators.
- This function has the option remember set.
- Whenever there is a conflict between the function name p2s and another name used in the same session, use the long form SYMF['p2s'].

Examples:

```
> with(SYMF):
> p2s(p2*p3);

 s[5] - s[2, 2, 1] + s[2, 1, 1, 1] - s[4, 1] + s[3, 2] - s[1, 1, 1, 1, 1]
```

SEE ALSO: s2p

5.12.56 s2c — convert a monomial in the Schur functions to the cycle index basis

✘

Calling Sequence:

s2c(sf)

SYMF[s2c](sf)

Parameters:

sf — any product of Schur functions

Synopsis:

- The s2c function converts a monomial in the Schur functions to an element expressed in the cycle index basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name s2c and another name used in the same session, use the long form SYMF['s2c'].

Examples:

```
> with(SYMF):
> s2c(s[2,1]*s[1,1]);

       1/120 c[3, 2] - 1/120 c[3, 1, 1] - 1/60 c[2, 1, 1, 1]

       + 1/6 c[1, 1, 1, 1, 1]
```

SEE ALSO: c2s

5.12.57 s2e — convert a monomial in the Schur functions to the e-basis

Calling Sequence:

s2e(sf)

SYMF[s2e](sf)

Parameters:

sf — any product of Schur functions

Synopsis:

- The s2e function converts a monomial in the Schur functions to an element expressed in the e-basis using Jacobi-Trudi identity.
- This function has the option remember set.
- Whenever there is a conflict between the function name s2e and another name used in the same session, use the long form SYMF['s2e'].

Examples:

```
> with(SYMF):
> s2e(s[2,1]*s[3,1]);

     3         2   3                        2         2                 2
 - e2  e1 + e2  e1  + e4 e2 e1 - 2 e3 e1  e2 + e2  e3 - e4 e3 + e3  e1
```

SEE ALSO:

5.12.58 s2h — convert a monomial in the Schur functions to the h-basis

Calling Sequence:

s2h(sf)

SYMF[s2h](sf)

Parameters:

sf — any product of Schur functions

Synopsis:

- The `s2h` function converts a monomial in the Schur functions to an element expressed in the h-basis using Jacobi-Trudi identity.

- This function has the option remember set.

- Whenever there is a conflict between the function name `s2h` and another name used in the same session, use the long form `SYMF['s2h']`.

Examples:

```
> with(SYMF):
> s2h(s[2,1]*s[3,1]);
                        2                    2
                  h2 h1  h3 - h2 h1 h4 - h3  h1 + h3 h4
```

SEE ALSO:

5.12.59 s2m — convert a monomial in the Schur functions to the monomial symmetric function basis

Calling Sequence:

`s2m(`sf`)`

`SYMF[s2m](`sf`)`

Parameters:

sf — any product of Schur functions

Synopsis:

- The `s2m` function converts a monomial in the Schur functions to an element expressed in the monomial symmetric function basis.

- This function has the option remember set.

- Whenever there is a conflict between the function name `s2m` and another name used in the same session, use the long form `SYMF['s2m']`.

Examples:

```
> with(SYMF):
> s2m(s[2,1]*s[1,1]);

   m[3, 2] + 2 m[3, 1, 1] + 4 m[2, 2, 1] + 9 m[2, 1, 1, 1] + 20 m[1, 1, 1, 1, 1]
```

SEE ALSO: `m2s`

5.12.60 s2p — convert a monomial in the Schur functions to the p-basis

Calling Sequence:

s2p(sf)

SYMF[s2p](sf)

Parameters:

sf — any product of Schur functions

Synopsis:

- The s2p function converts a monomial in the Schur functions to an element expressed in the p-basis.
- This function has the option remember set.
- Whenever there is a conflict between the function name s2p and another name used in the same session, use the long form SYMF['s2p'].

Examples:

```
> with(SYMF):
> s2p(s[2,1]*s[3,1]);

                          2                2             4          3
  1/12 p3 p4 + 1/24 p3 p2  - 1/12 p3 p2 p1  - 1/24 p3 p1  - 1/12 p1  p4

                        3   2          5             7
              - 1/24 p1  p2  + 1/12 p1  p2 + 1/24 p1
```

SEE ALSO: p2s

5.13 CLG, a package to handle characters of linear groups

Calling Sequence:

CLG[<function>]($args$)

<function>($args$)

Synopsis:

- This package provides functions to manipulated characters of the linear group $GL(n)$.
- One chooses the order using the function CLG_n (default degree is 5).
- Whenever there is a conflict between a function name in CLG and another name used in the same session, use the long form CLG[<function>].
- The available functions are:

CLG_n	DimRep_n	Gauss	LRrSf_n
MuirSf_n	PieriSf_n	Toe_n	Toh_n
Tom_n	Top_n	Tos_n	Tox_n
e2x_n	h2x_n	m2x_n	p2x_n
s2x_n	x2m_n		

- For help with a particular function do `?CLG[<function>]` or `?CLG,<function>` where `<function>` is one from the above list.
- The functions defined in the package provide a type checking on their arguments. All these functions may be called by their names in the package.

Examples:

```
> with(CLG):
> CLG_n(5);
```

5

5.13.1 CLG_n — fix the order of the linear group

Calling Sequence:

`CLG_n`(n)

`CLG[CLG_n]`(n)

Parameters:

n — any positive integer

Synopsis:

- The `CLG_n` function may be used to fix the linear group degree to n. Default order is 5.
- When called without any argument, it returns the current value of `_CLGn`.
- Whenever there is a conflict between the function name `CLG_n` and another name used in the same session, use the long form `CLG['CLG_n']`.

Examples:

```
> with(CLG):
> CLG_n(5);
```

5

SEE ALSO:

✘

5.13.2 DimRep_n — compute the dimension of a representation

Calling Sequence:

DimRep_n($part$)

DimRep_n($part, part_{in}$)

CLG[DimRep_n]($part$)

CLG[DimRep_n]($part, part_{in}$)

Parameters:

$part, part_{in}$ — any lists denoting partitions

Synopsis:

- The DimRep_n function computes the dimension of the irreducible representation of index $part$, for the linear group of order _CLGn.
- When called with a second argument $part_{in}$, it computes the dimension of the representation indexed by the skew partition $part/part_{in}$.
- Adding the parameter $'q'$, it computes the specialization of s_{part} or $s_{part/part_{in}}$ in $x_1 = 1, x_2 = q, \ldots, x_n = q^{n-1}$.
- Whenever there is a conflict between the function name DimRep_n and another name used in the same session, use the long form CLG['DimRep_n'].

Examples:

```
> with(CLG):
> DimRep_n([2,2,1]);

                                   75

> DimRep_n([2,2,1], [1,1]);

                                   50

> DimRep_n([3,1,1], [2,1], 'q');

                            4    3    2          2
                          (q  + q  + q  + q + 1)
```

See Also:

5.13.3 Gauss — compute a Gaussian polynomial

✓

Calling Sequence:

Gauss($m, n, 'q'$)

CLG[Gauss]($m, n, 'q'$)

Parameters:

m — any non-negative integer

n — any integer

Synopsis:

- The `Gauss` function computes the Gaussian polynomial corresponding to the parameters m and n. This is a polynomial in the variable $'q'$ that is the generalization of the corresponding binomial coefficient `binomial`(m, n).
- `Gauss`$(m, n, q) = 0$ whenever $n < 0$ or $n > m$.
- `Gauss`$(m, n, q) = 1$ whenever $n = 0$ or $n = m$.
- `Gauss`$(m, n, q) =$ `Gauss`$(m-1, n-1, q) + q^n$ `Gauss`$(m-1, n, q)$.
- Whenever there is a conflict between the function name `Gauss` and another name used in the same session, use the long form `CLG['Gauss']`.

Examples:

```
> with(CLG):
> Gauss(4, 2, q);

                          4    3      2
                         q  + q  + 2 q  + q + 1

> q:=2:
> Gauss(4, 2, q);

                                  35

> Gauss(4, 2, 'q');

                          4    3      2
                         q  + q  + 2 q  + q + 1
```

SEE ALSO:

5.13.4 `LRrSf_n` — Littlewood-Richardson rule

Calling Sequence:

`LRrSf_n`$(schur_1, schur_2)$

`CLG[LRrSf_n]`$(schur_1, schur_2)$

Parameters:

$schur_1, schur_2$ — any Schur functions

Synopsis:

- The `LRrSf_n` function computes the product of the two Schur functions $schur_1$ and $schur_2$ using the Littlewood-Richardson rule. The result is expressed in the basis of Schur functions indexed by partitions of length $\leq$ `_CLGn`.

- Whenever there is a conflict between the function name `LRrSf_n` and another name used in the same session, use the long form `CLG['LRrSf_n']`.

Examples:

```
> with(CLG):
> _CLGn;
                                  5

> LRrSf_n(s[6,3,3,1], s[2,2]);

    s[6, 3, 3, 3, 2] + s[6, 4, 3, 2, 2] + s[6, 4, 3, 3, 1]

    + s[6, 4, 4, 2, 1] + s[6, 5, 3, 2, 1] + s[6, 5, 3, 3]

    + s[6, 5, 4, 1, 1] + s[6, 5, 4, 2] + s[6, 5, 5, 1] + s[7, 3, 3, 2, 2]

    + s[7, 3, 3, 3, 1] + 2 s[7, 4, 3, 2, 1] + s[7, 4, 3, 3]

    + s[7, 4, 4, 1, 1] + s[7, 4, 4, 2] + s[7, 5, 3, 1, 1] + s[7, 5, 3, 2]

    + s[7, 5, 4, 1] + s[8, 3, 3, 2, 1] + s[8, 3, 3, 3] + s[8, 4, 3, 1, 1]

    + s[8, 4, 3, 2] + s[8, 5, 3, 1]
```

SEE ALSO: `MuirSf_n`, `PieriSf_n`

5.13.5 MuirSf_n — Muir's formula

Calling Sequence:

`MuirSf_n`(*mono*, *schur*)

`CLG[MuirSf_n]`(*mono*, *schur*)

Parameters:

mono — any monomial symmetric function

schur — any Schur function

Synopsis:

- The `MuirSf_n` function computes the product of the monomial symmetric function $mono = m_{part}$ by the Schur function *schur*, using the Muir formula. The result is expressed in the basis of Schur functions indexed by partitions of length $\leq$ `_CLGn`.

- Whenever there is a conflict between the function name `MuirSf_n` and another name used in the same session, use the long form `CLG['MuirSf_n']`.

Examples:

```
> with(CLG):
> _CLGn;
```

```
                                  5

> MuirSf_n(m[2,1], s[4,2,2,1]);

  s[6, 3, 2, 1] + s[4, 4, 3, 1] - s[5, 3, 3, 1] + s[5, 4, 2, 1]

   + s[6, 2, 2, 1, 1] + s[6, 2, 2, 2] - s[4, 3, 3, 1, 1] - s[4, 3, 3, 2]

   + s[4, 4, 2, 1, 1] + s[4, 4, 2, 2] + s[4, 2, 2, 2, 2]
```

SEE ALSO: LRrSf_n, PieriSf_n

5.13.6 PieriSf_n — Pieri's formula

Calling Sequence:

PieriSf_n($f, schur$)

CLG[PieriSf_n]($f, schur$)

Parameters:

f — any elementary or complete symmetric function

schur — any Schur function

Synopsis:

- The PieriSf_n function computes the product of the elementary / complete symmetric function e_k / h_k by the Schur function *schur*, using the Pieri formula. The result is expressed in the basis of Schur functions indexed by partitions of length $\leq$ _CLGn.
- Whenever there is a conflict between the function name PieriSf_n and another name used in the same session, use the long form CLG['PieriSf_n'].

Examples:

```
> with(CLG):
> _CLGn;

                                  5

> PieriSf_n(e2, s[2,1]);

            s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 2] + s[3, 1, 1]

> PieriSf_n(h3, s[3,2,2,1]);

     s[6, 2, 2, 1] + s[5, 3, 2, 1] + s[5, 2, 2, 2] + s[4, 3, 2, 2]

          + s[5, 2, 2, 1, 1] + s[4, 3, 2, 1, 1] + s[3, 3, 2, 2, 1]

          + s[4, 2, 2, 2, 1]
```

SEE ALSO: LRrSf_n, MuirSf_n

5.13.7 Toe_n — convert any symmetric function to a e-polynomial

Calling Sequence:

Toe_n(sf)

Toe_n(sf, b)

CLG[Toe_n](sf)

CLG[Toe_n](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Toe_n function converts any symmetric function to a e-polynomial.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Toe_n(sf, 'e') does not affect the argument sf.
- One may add 'noexpand' just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by Toe_n(sf, b, 'collect'). For instance, Toe_n(sf, 'e', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Toe_n and another name used in the same session, use the long form CLG['Toe_n'].

Examples:

```
> with(CLG):
> _CLGn;
                                5

> Toe_n((1+q)^500*s[3,2,2,1]):                 # expands the input
> Toe_n((1+q)^500*s[3,2,2,1],noexpand):        # does not expand (1+q)^500
> Toe_n((1+q)^500*s[3,2,2,1],collect):         # collects the result
> Toe_n((1+q)^500*s[3,2,2,1],noexpand,'s'):    # the most efficient
> Toe_n((1+q)^500*s[3,2,2,1],'s',collect):     # specifies a basis
> Toe_n((1+q)^500*s[3,2,2,1],noexpand,'s',collect):
> Toe_n(s[2,1]*h4 - 2*q*p3);

                                   2          2        3       2
        - e4 e2 e1 + e4 e3 + 5 e3 e1  e2 - 2 e3  e1 + e2  e1 - e2  e3

              2   3      5        4                                 3
        - 3 e2  e1   + e1  e2 - e1  e3 - 6 q e3 + 6 q e2 e1 - 2 q e1
```

SEE ALSO: TYP[Sf], Toh_n, Top_n, Tos_n

5.13.8 Toh_n — convert any symmetric function to a h-polynomial

Calling Sequence:

Toh_n(sf)

Toh_n(sf, b)

CLG[Toh_n](sf)

CLG[Toh_n](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Toh_n function converts any symmetric function to a h-polynomial.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Toh_n(sf, 'h') does not affect the argument sf.
- One may add 'noexpand' just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by Toh_n(sf, b, 'collect'). For instance, Toh_n(sf, 'h', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Toh_n and another name used in the same session, use the long form CLG['Toh_n'].

Examples:

```
> with(CLG):
> _CLGn;

                                   5

> Toh_n((1+q)^500*s[3,2,2,1]):                # expands the input
> Toh_n((1+q)^500*s[3,2,2,1],noexpand):       # does not expand (1+q)^500
> Toh_n((1+q)^500*s[3,2,2,1],collect):        # collects the result
> Toh_n((1+q)^500*s[3,2,2,1],noexpand,'s'):   # the most efficient
> Toh_n((1+q)^500*s[3,2,2,1],'s',collect):    # specifies a basis
> Toh_n((1+q)^500*s[3,2,2,1],noexpand,'s',collect):
> Toh_n(s[2,1]*h4 - 2*q*p3);

                                                           3
               h4 h2 h1 - h4 h3 - 6 q h3 + 6 q h2 h1 - 2 q h1
```

SEE ALSO: TYP[Sf], Toe_n, Top_n, Tos_n

5.13.9 Tom_n — express any symmetric function in the basis of basis of monomial symmetric functions

✘

Calling Sequence:

Tom_n(sf)

Tom_n(sf, b)

CLG[Tom_n](sf)

CLG[Tom_n](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Tom_n function expresses any symmetric function in the basis of monomial symmetric functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Tom_n(sf, 'm') does not affect the argument sf.
- One may add 'noexpand' just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: Tom_n($sf, b,$'collect'). For instance, Tom_n(sf, 'm', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Tom_n and another name used in the same session, use the long form CLG['Tom_n'].

✂ ....

Examples:

```
> with(CLG):
> _CLGn;

                                    5

> Tom_n((1+q)^500*e3*e4):                     # expands the input
> Tom_n((1+q)^500*e3*e4,noexpand):            # does not expand (1+q)^500
> Tom_n((1+q)^500*e3*e4,collect):             # collects the result
> Tom_n((1+q)^500*e3*e4,noexpand,'e'):        # the most efficient
> Tom_n((1+q)^500*e3*e4,'e',collect):         # specifies a basis
> Tom_n((1+q)^500*e3*e4,noexpand,'e',collect):
> Tom_n(s[2,1]*h4 - 2*q*p3);

 m[6, 1] + 2 m[5, 2] + 4 m[5, 1, 1] + 2 m[4, 3] + 6 m[4, 2, 1]
```

```
+ 11 m[4, 1, 1, 1] + 6 m[3, 3, 1] + 8 m[3, 2, 2] + 14 m[3, 2, 1, 1]

+ 24 m[3, 1, 1, 1, 1] + 17 m[2, 2, 2, 1] + 28 m[2, 2, 1, 1, 1] - 2 q m[3]
```

SEE ALSO: Toe_n, Toh_n, Top_n, Tos_n

5.13.10 Top_n — convert any symmetric function to a p-polynomial

Calling Sequence:

Top_n(sf)

Top_n(sf, b)

CLG[Top_n](sf)

CLG[Top_n](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Top_n function converts any symmetric function to a p-polynomial.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Top_n(sf, 'p') does not affect the argument sf.
- One may add 'noexpand' just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by Top_n(sf, b, 'collect'). For instance, Top_n(sf, 'p', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Top_n and another name used in the same session, use the long form CLG['Top_n'].

Examples:

```
> with(CLG):
> _CLGn;

                              5

> Top_n((1+q)^500*s[3,2,2,1]):               # expands the input
> Top_n((1+q)^500*s[3,2,2,1],noexpand):      # does not expand (1+q)^500
> Top_n((1+q)^500*s[3,2,2,1],collect):       # collects the result
> Top_n((1+q)^500*s[3,2,2,1],noexpand,'s'):  # the most efficient
```

```
> Top_n((1+q)^500*s[3,2,2,1],'s',collect):     # specifies a basis
> Top_n((1+q)^500*s[3,2,2,1],noexpand,'s',collect):
> Top_n(s[2,1]*h4 - 2*q*p3);

               3                          4         2             2   3
  1/12 p4 p1  - 1/12 p4 p3 + 7/72 p3 p1  - 1/9 p3  p1 + 1/24 p2  p1

               2                5             2             7
    - 1/24 p2  p3 + 1/12 p2 p1  - 1/12 p2 p1  p3 + 1/72 p1  - 2 q p3
```

SEE ALSO: TYP[Sf], Toe_n, Toh_n, Tos_n

5.13.11 Tos_n — express any symmetric function in the basis of Schur functions

Calling Sequence:

Tos_n(sf)

Tos_n(sf, b)

CLG[Tos_n](sf)

CLG[Tos_n](sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The Tos_n function expresses any symmetric function in the basis of Schur functions, *i.e.* as a linear combination of Schur functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- The call Tos_n(sf, 's') does not affect the argument sf.
- One may add 'noexpand' just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by Tos_n(sf, b, 'collect'). For instance, Tos_n(sf, 's', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Tos_n and another name used in the same session, use the long form CLG['Tos_n'].

Examples:

```
> with(CLG):
> _CLGn;

                                  5

> Tos_n((1+q)^500*e3*e4):                      # expands the input
> Tos_n((1+q)^500*e3*e4,noexpand):             # does not expand (1+q)^500
> Tos_n((1+q)^500*e3*e4,collect):              # collects the result
> Tos_n((1+q)^500*e3*e4,noexpand,'e'):         # the most efficient
> Tos_n((1+q)^500*e3*e4,'e',collect):          # specifies a basis
> Tos_n((1+q)^500*e3*e4,noexpand,'e',collect):
> Tos_n(s[2,1]*h4 - 2*q*p3);

           s[4, 2, 1] + s[5, 1, 1] + s[5, 2] + s[6, 1]

           - 2 (s[3] - s[2, 1] + s[1, 1, 1]) q
```

SEE ALSO: `TYP[Sf]`, `Toe_n`, `Toh_n`, `Top_n`

5.13.12 `Tox_n` — express any symmetric function in the basis of monomials

Calling Sequence:

`Tox_n`(sf)

`Tox_n`(sf, b)

`CLG[Tox_n]`(sf)

`CLG[Tox_n]`(sf, b)

Parameters:

sf — any symmetric function

b — any name of a known basis

Synopsis:

- The `Tox_n` function expresses any symmetric function in the basis of monomials.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- One may add `'noexpand'` just after the argument sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by `Tox_n`(sf, b, `'collect'`).
- Whenever there is a conflict between the function name `Tox_n` and another name used in the same session, use the long form `CLG['Tox_n']`.

Examples:

```
> with(CLG):
> _CLGn;

                                   5

> Tox_n((1+q)^500*e3*e4):                      # expands the input
> Tox_n((1+q)^500*e3*e4,noexpand):             # does not expand (1+q)^500
> Tox_n((1+q)^500*e3*e4,collect):              # collects the result
> Tox_n((1+q)^500*e3*e4,noexpand,'e'):         # the most efficient
> Tox_n((1+q)^500*e3*e4,'e',collect):          # specifies a basis
> Tox_n((1+q)^500*e3*e4,noexpand,'e',collect):
> Tox_n(p1^2);
                 2
   2 x2 x1 + x1  + 2 x5 x4 + 2 x5 x3 + 2 x5 x2 + 2 x5 x1 + 2 x4 x3

                                     2     2     2     2
   + 2 x4 x1 + 2 x3 x2 + 2 x3 x1 + x4  + x3  + x2  + x5  + 2 x4 x2
```

SEE ALSO: TYP[Sf]

5.13.13 e2x_n — from e-basis to the basis of monomials

Calling Sequence:

e2x_n(sf)

CLG[e2x_n](sf)

Parameters:

sf — any expression in e-basis

Synopsis:

- The e2x_n function converts an expression from e-basis to the basis of monomials.
- The result is not expanded.
- Whenever there is a conflict between the function name e2x_n and another name used in the same session, use the long form CLG['e2x_n'].

Examples:

```
> with(CLG):
> CLG_n(3);
                                       3
> e2x_n(e2);
                             x3 (x2 + x1) + x2 x1

> expand( e2x_n(2*e1*e2 - 4*q^2*e1^2) );

               2            2            2             2           2
 6 x3 x2 x1 - 8 q  x3 x2 - 8 q  x3 x1 - 8 q  x2 x1 + 2 x3  x2 + 2 x3  x1
```

```
            2           2         2                2         2   2        2   2
+ 2 x3 x2  + 2 x3 x1  + 2 x2  x1 + 2 x2 x1  - 4 q  x3  - 4 q  x2

    2   2
- 4 q  x1
```

SEE ALSO:

5.13.14 h2x_n — from h-basis to the basis of monomials

Calling Sequence:

`h2x_n`(sf)

`CLG[h2x_n]`(sf)

Parameters:

sf — any expression in h-basis

Synopsis:

- The `h2x_n` function converts an expression from h-basis to the basis of monomials.
- The result is not expanded.
- Whenever there is a conflict between the function name `h2x_n` and another name used in the same session, use the long form `CLG['h2x_n']`.

Examples:

```
> with(CLG):
> CLG_n(3);

                                        3

> h2x_n(h2);

                                                              2
                    x3 (x3 + x2 + x1) + x2 (x2 + x1) + x1

> expand( h2x_n(2*h1*h2 - 4*q^2*h1^2) );

                 3       2          2             2          2        2
6 x3 x2 x1 + 2 x1  + 4 x3  x2 + 4 x3  x1 + 4 x3 x2  + 4 x3 x1  + 4 x2  x1

             2       3       3      2             2             2
    + 4 x2 x1  + 2 x3  + 2 x2  - 8 q  x3 x2 - 8 q  x3 x1 - 8 q  x2 x1

          2   2       2   2       2   2
    - 4 q  x3  - 4 q  x2  - 4 q  x1
```

SEE ALSO:

5.13.15 m2x_n — from monomial symmetric function basis to the basis of monomials

Calling Sequence:

m2x_n(sf)

CLG[m2x_n](sf)

Parameters:

sf — any expression in monomial symmetric function basis

Synopsis:

- The m2x_n function converts an expression from monomial symmetric function basis to the basis of monomials.
- The result is not expanded.
- Whenever there is a conflict between the function name m2x_n and another name used in the same session, use the long form CLG['m2x_n'].

Examples:

```
> with(CLG):
> CLG_n(3);

                                        3

> m2x_n(m[2,1]);

              2        2            2        2     2            2
            x1  x2 + x1  x3 + x1 x2  + x1 x3  + x2  x3 + x2 x3

> m2x_n(2*m[2,1,1,0] + 4*q^2*m[1,1,0]);

      2                 2                 2      2
  2 x1  x2 x3 + 2 x1 x2  x3 + 2 x1 x2 x3  + 4 q  (x1 x2 + x1 x3 + x2 x3)
```

See Also:

5.13.16 p2x_n — from p-basis to the basis of monomials

Calling Sequence:

p2x_n(sf)

CLG[p2x_n](sf)

Parameters:

sf — any expression in p-basis

Synopsis:

- The p2x_n function converts an expression from p-basis to the basis of monomials.
- The result is not expanded.

- Whenever there is a conflict between the function name `p2x_n` and another name used in the same session, use the long form `CLG['p2x_n']`.

Examples:

```
> with(CLG):
> CLG_n(3);

                                       3

> p2x_n(p2);

                                  2     2     2
                                x1  + x2  + x3

> p2x_n(2*p2*p3 + 4*q^2*p1^2);

          2     2     2     3     3     3                    2  2
     2 (x1  + x2  + x3 ) (x1  + x2  + x3 ) + 4 (x1 + x2 + x3)  q
```

SEE ALSO:

5.13.17 s2x_n — from Schur function basis to the basis of monomials

Calling Sequence:

`s2x_n(`sf`)`

`CLG[s2x_n](`sf`)`

Parameters:

sf — any expression in Schur function basis

Synopsis:

- The `s2x_n` function converts an expression from Schur function basis to the basis of monomials.
- The result is not expanded.
- Whenever there is a conflict between the function name `s2x_n` and another name used in the same session, use the long form `CLG['s2x_n']`.

Examples:

```
> with(CLG):
> CLG_n(3);
                                       3

> s2x_n(s[2,1]);

                                                       2      2           2
  x3 (x3 (x2 + x1) + x2 x1) + x3 (x2 (x2 + x1) + x1 ) + x2  x1 + x2 x1

> s2x_n(2*s[2,1,1,0] + 4*q^2*s[1,2,1]);
```

```
                2                        2              2
          2 x3   x2 x1 + 2 x3 (x2   x1 + x2 x1 )
```

SEE ALSO:

5.13.18 x2m_n — from the basis of monomials to monomial symmetric function basis

Calling Sequence:

x2m_n(*pol*)

CLG[x2m_n](*pol*)

Parameters:

pol — any expanded polynomial in $x_1, x_2, x_3, \ldots$ (assumed to be symmetrical)

Synopsis:

- The x2m_n function converts any symmetric polynomial into a linear combination of monomial symmetric functions.
- Whenever there is a conflict between the function name x2m_n and another name used in the same session, use the long form CLG['x2m_n'].

Examples:

```
> with(CLG):
> x2m_n( expand( (1+x1) * (1+x2) * (1+x3) ) );

                      1 + m[1] + m[1, 1] + m[1, 1, 1]
```

SEE ALSO: m2x_n, SP[x2X]

5.14 Noncommutative and quasi-symmetric functions with NCSF

✓

Calling Sequence:

NCSF[<function>]$(args)$

<function>$(args)$

Synopsis:

- This package provides functions to work with noncommutative symmetric functions and quasi-symmetric functions.

- Whenever there is a conflict between a function name in NCSF and another name used in the same session, use the long form NCSF[<function>].

- The available functions are:

ConjugCompo	E2FMat	E2MMat	E2QPhMat
E2QPsMat	F2EMat	F2MMat	F2QPhMat
F2QPsMat	Internal	IsFiner	L2PhMat
L2PsMat	L2RMat	L2SMat	ListCompo
ListCompoFiner	ListCompoNotFiner	M2EMat	M2FMat
M2QPhMat	M2QPsMat	MajorIndex	Pairing
PermDesc	Ph2LMat	Ph2PsMat	Ph2RMat
Ph2SMat	Pleth	Ps2LMat	Ps2PhMat
Ps2RMat	Ps2SMat	QPh2EMat	QPh2FMat
QPh2MMat	QPh2QPsMat	QPs2EMat	QPs2FMat
QPs2MMat	QPs2QPhMat	R2LMat	R2PhMat
R2PsMat	R2SMat	S2LMat	S2PhMat
S2PsMat	S2RMat	Qsf2x	Sf2Qsf
Shuffle	SpDirect	SpInverse	ToE
ToF	ToL	ToM	ToPh
ToPs	ToQPh	ToQPs	ToR
ToS			

- For help with a particular function do ?NCSF[<function>] or ?NCSF,<function> where <function> is one from the above list.

Examples:

```
> with(NCSF):
> Ps2SMat(3);
```

```
                    [ 3  -1  -2   1 ]
                    [               ]
                    [ 0   2   0  -1 ]
                    [               ]
                    [ 0   0   2  -1 ]
                    [               ]
                    [ 0   0   0   1 ]
```

5.14.1 `ConjugCompo` — conjugate a composition

Calling Sequence:

`ConjugCompo`(*compo*)

`NCSF[ConjugCompo]`(*compo*)

Parameters:

compo — any list denoting a composition

Synopsis:

- The `ConjugCompo` function calculates the conjugate composition of a given composition *compo*.
- Whenever there is a conflict between the function name `ConjugCompo` and another name used in the same session, use the long form `NCSF['ConjugCompo']`.

Examples:

```
> with(NCSF):
> ConjugCompo([3,4,1,2,1]);
```

```
                  [2, 3, 1, 1, 2, 1, 1]
```

See Also:

5.14.2 `ListCompo` — generate all compositions of a given weight

Calling Sequence:

`ListCompo`(n)

`NCSF[ListCompo]`(n)

Parameters:

n — any positive integer

Synopsis:

- The `ListCompo` function generates all compositions of a given weight n.

- The result is a list of compositions in reverse lexicographic order.
- Whenever there is a conflict between the function name `ListCompo` and another name used in the same session, use the long form `NCSF['ListCompo']`.

Examples:

```
> with(NCSF):
> ListCompo(3);

                    [[3], [2, 1], [1, 2], [1, 1, 1]]
```

SEE ALSO:

5.14.3 `E2FMat` — transition matrix from E-basis to F-basis

Calling Sequence:

`E2FMat`(n)

`NCSF[E2FMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The `E2FMat` function computes the transition matrix from the E-basis (elementary quasi-symmetric functions) to the F-basis (ribbon Schur quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `E2FMat` and another name used in the same session, use the long form `NCSF['E2FMat']`.

Examples:

```
> with(NCSF):
> E2FMat(3);

                              [  1  -1  -1  1 ]
                              [               ]
                              [ -1   0   1  0 ]
                              [               ]
                              [ -1   1   0  0 ]
                              [               ]
                              [  1   0   0  0 ]
```

SEE ALSO: `F2EMat`

5.14.4 E2MMat — transition matrix from E-basis to M-basis

Calling Sequence:

E2MMat(n)

NCSF[E2MMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The **E2MMat** function computes the transition matrix from the E-basis (elementary quasi-symmetric functions) to the M-basis (monomial quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name **E2MMat** and another name used in the same session, use the long form **NCSF['E2MMat']**.

Examples:

```
> with(NCSF):
> E2MMat(3);

                    [  1    0    0   0 ]
                    [                  ]
                    [ -1   -1    0   0 ]
                    [                  ]
                    [ -1    0   -1   0 ]
                    [                  ]
                    [  1    1    1   1 ]
```

SEE ALSO: M2EMat

5.14.5 E2QPhMat — transition matrix from E-basis to QPh-basis

Calling Sequence:

E2QPhMat(n)

NCSF[E2QPhMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The **E2QPhMat** function computes the transition matrix from the E-basis (elementary quasi-symmetric functions) to the QPh-basis (dual basis of noncommutative power sums symmetric functions of the second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name **E2QPhMat** and another name used in the same session, use the long form **NCSF['E2QPhMat']**.

Examples:

```
> with(NCSF):
> E2QPhMat(3);
```

```
                    [  1      0      0     0  ]
                    [                         ]
                    [ -1/2  -1/2     0     0  ]
                    [                         ]
                    [ -1/2    0    -1/2    0  ]
                    [                         ]
                    [  1/3   1/4    1/4   1/6 ]
```

SEE ALSO: `QPh2EMat`

5.14.6 `E2QPsMat` — transition matrix from E-basis to QPs-basis

Calling Sequence:

`E2QPsMat`(n)

`NCSF[E2QPsMat]`(n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The `E2QPsMat` function computes the transition matrix from the E-basis (elementary quasi-symmetric functions) to the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `E2QPsMat` and another name used in the same session, use the long form `NCSF['E2QPsMat']`.

Examples:

```
> with(NCSF):
> E2QPsMat(3);
```

```
                    [  1      0      0     0  ]
                    [                         ]
                    [ -2/3  -1/3     0     0  ]
                    [                         ]
                    [ -1/3    0    -2/3    0  ]
                    [                         ]
                    [  1/3   1/6    1/3   1/6 ]
```

SEE ALSO: `QPs2EMat`

5.14.7 F2EMat — transition matrix from F-basis to E-basis

Calling Sequence:

F2EMat(n)

NCSF[F2EMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The **F2EMat** function computes the transition matrix from the F-basis (ribbon Schur quasi-symmetric functions) to the E-basis (elementary quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name **F2EMat** and another name used in the same session, use the long form **NCSF['F2EMat']**.

Examples:

```
> with(NCSF):
> F2EMat(3);

                    [ 0  0  0  1 ]
                    [            ]
                    [ 0  0  1  1 ]
                    [            ]
                    [ 0  1  0  1 ]
                    [            ]
                    [ 1  1  1  1 ]
```

SEE ALSO: E2FMat

5.14.8 F2MMat — transition matrix from F-basis to M-basis

Calling Sequence:

F2MMat(n)

NCSF[F2MMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The **F2MMat** function computes the transition matrix from the F-basis (ribbon Schur quasi-symmetric functions) to the M-basis (monomial quasi-symmetric function).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name **F2MMat** and another name used in the same session, use the long form **NCSF['F2MMat']**.

Examples:

```
> with(NCSF):
> F2MMat(3);
```

```
                    [ 1   1   1   1 ]
                    [               ]
                    [ 0   1   0   1 ]
                    [               ]
                    [ 0   0   1   1 ]
                    [               ]
                    [ 0   0   0   1 ]
```

SEE ALSO: `M2FMat`

5.14.9 `F2QPhMat` — transition matrix from F-basis to QPh-basis

✘

Calling Sequence:

`F2QPhMat`(n)

`NCSF[F2QPhMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The `F2QPhMat` function computes the transition matrix from the F-basis (ribbon Schur quasi-symmetric functions) to the QPh-basis (dual basis of noncommutative power sums symmetric function of second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `F2QPhMat` and another name used in the same session, use the long form `NCSF['F2QPhMat']`.

Examples:

```
> with(NCSF):
> F2QPhMat(3);
```

```
              [  1/3    1/4    1/4   1/6 ]
              [                          ]
              [ -1/6    1/4   -1/4   1/6 ]
              [                          ]
              [ -1/6   -1/4    1/4   1/6 ]
              [                          ]
              [  1/3   -1/4   -1/4   1/6 ]
```

SEE ALSO: `QPh2FMat`

5.14.10 F2QPsMat — transition matrix from F-basis to QPs-basis

Calling Sequence:

F2QPsMat(n)

NCSF[F2QPsMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The F2QPsMat function computes the transition matrix from the F-basis (ribbon Schur quasi-symmetric functions) to the QPs-basis (dual basis of noncommutative power sums symmetric function of first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name F2QPsMat and another name used in the same session, use the long form NCSF['F2QPsMat'].

Examples:

```
> with(NCSF):
> F2QPsMat(3);

                    [  1/3   1/6   1/3  1/6 ]
                    [                       ]
                    [   0    1/6  -1/3  1/6 ]
                    [                       ]
                    [ -1/3  -1/6   1/3  1/6 ]
                    [                       ]
                    [  1/3  -1/6  -1/3  1/6 ]
```

SEE ALSO: QPs2FMat

5.14.11 ListCompoFiner — generate all compositions finer than a given one

Calling Sequence:

ListCompoFiner(*compo*)

NCSF[ListCompoFiner](*compo*)

Parameters:

compo — any list denoting a composition

Synopsis:

- The ListCompoFiner function generates all compositions finer than a given composition *compo*. The result is a list of compositions.
- Whenever there is a conflict between the function name ListCompoFiner and another name used in the same session, use the long form NCSF['ListCompoFiner'].

Examples:

```
> with(NCSF):
> ListCompoFiner([2,1,3]);

   [[2, 1, 3], [2, 1, 2, 1], [2, 1, 1, 2], [2, 1, 1, 1, 1], [1, 1, 1, 3],

          [1, 1, 1, 2, 1], [1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 1]]
```

SEE ALSO:

5.14.12 `Internal` — internal product

Calling Sequence:

`Internal`($ncsf_1, ncsf_2$)

`NCSF[Internal]`($ncsf_1, ncsf_2$)

Parameters:

$ncsf_1, ncsf_2$ — any noncommutative symmetric functions

Synopsis:

- The `Internal` function calculates the internal product between the two noncommutative symmetric functions $ncsf_1$ and $ncsf_2$.
- We define such a product in the S-basis. Let $I = [i_1, i_2, ..., i_p]$ and $J = [j_1, j_2, ..., j_q]$ be two compositions: then
- `Internal`(S^I, S^J)=0 if $i_1 + i_2 + \cdots + i_p \neq j_1 + j_2 + \cdots + j_q$, and otherwise
- `Internal`(S^I, S^J)=$\sum S^M$ where the sum is over all matrices $M = (m_{ij})$ of nonnegative integers of order $p \times q$ with row sums vector I and column sums vector J, and $S^M = S^{m_{11}} S^{m_{12}} \cdots S^{m_{qp}}$.
- The result is expressed in the S-basis.
- Whenever there is a conflict between the function name `Internal` and another name used in the same session, use the long form `NCSF['Internal']`.

Examples:

```
> with(NCSF):
> Internal(L[2,1,1]+Ph[2,2], R[2,1,1]);

              - 4 S[2, 2] + 3 S[2, 1, 1] + 2 S[1, 1, 2]
```

SEE ALSO:

5.14.13 IsFiner — test whether a composition is finer than another one

Calling Sequence:

Isfiner($compo_1, compo_2$)

NCSF[IsFiner]($compo_1, compo_2$)

Parameters:

$compo_1, compo_2$ — any lists denoting compositions

Synopsis:

- The IsFiner function tests whether the composition $compo_1$ is finer than the composition $compo_2$.
- Whenever there is a conflict between the function name IsFiner and another name used in the same session, use the long form SG['IsFiner'].

Examples:

```
> with(NCSF):
> IsFiner([2,1,2,1,3], [2,3,1,3]);

                                    true

> IsFiner([2,1,4,2,1,1], [3,4,1,2]);

                                    false
```

SEE ALSO: ListCompoFiner

5.14.14 L2PhMat — transition matrix from Λ-basis to Φ-basis

Calling Sequence:

L2PhMat(n)

NCSF[L2PhMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The L2PhMat function computes the transition matrix from the Λ-basis (elementary symmetric functions) to the Φ-basis (power sums symmetric functions of the second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name L2PhMat and another name used in the same session, use the long form NCSF['L2PhMat'].

Examples:

```
> with(NCSF):
> L2PhMat(3);

                    [ 1/3  -1/4  -1/4  1/6 ]
                    [                      ]
                    [  0   -1/2    0   1/2 ]
                    [                      ]
                    [  0     0   -1/2  1/2 ]
                    [                      ]
                    [  0     0     0    1  ]
```

SEE ALSO: `Ph2LMat`

5.14.15 `L2PsMat` — transition matrix from Λ-basis to Ψ-basis

Calling Sequence:

`L2PsMat`(n)

`NCSF[L2PsMat]`(n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The `L2PsMat` function computes the transition matrix from the Λ-basis (noncommutative elementary symmetric functions) to the Ψ-basis (noncommutative power sums symmetric functions of the first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `L2PsMat` and another name used in the same session, use the long form `NCSF['L2PsMat']`.

Examples:

```
> with(NCSF):
> L2PsMat(3);

                    [ 1/3  -1/3  -1/6  1/6 ]
                    [                      ]
                    [  0   -1/2    0   1/2 ]
                    [                      ]
                    [  0     0   -1/2  1/2 ]
                    [                      ]
                    [  0     0     0    1  ]
```

SEE ALSO: `Ps2LMat`

5.14.16 L2RMat — transition matrix from Λ-basis to R-basis

Calling Sequence:

L2RMat(n)

NCSF[L2RMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The L2RMat function computes the transition matrix from the Λ-basis (noncommutative elementary symmetric functions) to the R-basis (ribbon Schur functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name L2RMat and another name used in the same session, use the long form NCSF['L2RMat'].

Examples:

```
> with(NCSF):
> L2RMat(3);
```

```
[ 0  0  0  1 ]
[            ]
[ 0  0  1  1 ]
[            ]
[ 0  1  0  1 ]
[            ]
[ 1  1  1  1 ]
```

SEE ALSO: R2LMat

5.14.17 L2SMat — transition matrix from Λ-basis to S-basis

Calling Sequence:

L2SMat(n)

NCSF[L2SMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The L2SMat function computes the transition matrix from the Λ-basis (noncommutative elementary symmetric functions) to the S-basis (noncommutative complete symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name L2SMat and another name used in the same session, use the long form NCSF['L2SMat'].

Examples:

```
> with(NCSF):
> L2SMat(3);
```

```
                          [ 1  -1  -1  1 ]
                          [              ]
                          [ 0  -1   0  1 ]
                          [              ]
                          [ 0   0  -1  1 ]
                          [              ]
                          [ 0   0   0  1 ]
```

SEE ALSO: `S2LMat`

5.14.18 `M2EMat` — transition matrix from M-basis to E-basis

Calling Sequence:

`M2EMat`(n)

`NCSF[M2EMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The `M2EMat` function computes the transition matrix from the M-basis (monomial quasi-symmetric functions) to the E-basis (elementary quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `M2EMat` and another name used in the same session, use the long form `NCSF['M2EMat']`.

Examples:

```
> with(NCSF):
> M2EMat(3);
```

```
                          [  1   0   0  0 ]
                          [               ]
                          [ -1  -1   0  0 ]
                          [               ]
                          [ -1   0  -1  0 ]
                          [               ]
                          [  1   1   1  1 ]
```

SEE ALSO: `E2MMat`

5.14.19 M2FMat — transition matrix from M-basis to F-basis

Calling Sequence:

M2FMat(n)

NCSF[M2FMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The M2FMat function computes the transition matrix from the M-basis (monomial quasi-symmetric functions) to the F-basis (ribbon Schur quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name M2FMat and another name used in the same session, use the long form NCSF['M2FMat'].

Examples:

```
> with(NCSF):
> M2FMat(3);

                    [ 1  -1  -1   1 ]
                    [               ]
                    [ 0   1   0  -1 ]
                    [               ]
                    [ 0   0   1  -1 ]
                    [               ]
                    [ 0   0   0   1 ]
```

SEE ALSO: F2MMat

5.14.20 M2QPhMat — transition matrix from M-basis to QPh-basis

Calling Sequence:

M2QPhMat(n)

NCSF[M2QPhMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The M2QPhMat function computes the transition matrix from the M-basis (monomial quasi-symmetric functions) to the QPh-basis (dual basis of noncommutative power sums symmetric functions of the second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name M2QPhMat and another name used in the same session, use the long form NCSF['M2QPhMat'].

Examples:

```
> with(NCSF):
> M2QPhMat(3);

                        [   1      0      0    0 ]
                        [                        ]
                        [ -1/2     1      0    0 ]
                        [                        ]
                        [ -1/2     0      1    0 ]
                        [                        ]
                        [  1/3   -1/2   -1/2   1 ]
```

SEE ALSO: `QPh2MMat`

5.14.21 `M2QPsMat` — transition matrix from M-basis to QPs-basis

Calling Sequence:

`M2QPsMat`(n)

`NCSF[M2QPsMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The `M2QPsMat` function computes the transition matrix from the M-basis (monomial quasi-symmetric functions) to the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `M2QPsMat` and another name used in the same session, use the long form `NCSF['M2QPsMat']`.

Examples:

```
> with(NCSF):
> M2QPsMat(3);

                        [   1      0      0      0  ]
                        [                           ]
                        [ -1/3    1/3     0      0  ]
                        [                           ]
                        [ -2/3     0     2/3     0  ]
                        [                           ]
                        [  1/3   -1/6   -1/3    1/6 ]
```

SEE ALSO: `QPs2MMat`

5.14.22 MajorIndex — compute the major index of a given composition

Calling Sequence:

MajorIndex(*compo*)

NCSF[MajorIndex](*compo*)

Parameters:

compo — any list denoting a composition

Synopsis:

- The MajorIndex function computes the major index of a given composition *compo*. Let $I = [i_1, i_2, \ldots, i_m]$ be a composition, then MajorIndex(I) is set to be $(m-1)i_1 + (m-2)i_2 + \cdots + i_{m-1}$.
- Whenever there is a conflict between the function name MajorIndex and another name used in the same session, use the long form NCSF['MajorIndex'].

Examples:

```
> with(NCSF):
> MajorIndex([2,3,1,2]);

                              13
```

SEE ALSO:

5.14.23 ListCompoNotFiner — generate all compositions not finer than a given one

Calling Sequence:

ListCompoNotFiner(*compo*)

NCSF[ListCompoNotFiner](*compo*)

Parameters:

compo — any list denoting a composition

Synopsis:

- The ListCompoNotFiner function generates all compositions not finer than a given composition *compo*. The result is a list of compositions.
- Whenever there is a conflict between the function name ListCompoNotFiner and another name used in the same session, use the long form NCSF['ListCompoNotFiner'].

Examples:

```
> with(NCSF):
> ListCompoNotFiner([2,1,3]);

         [[6], [2, 4], [3, 3], [2, 1, 3]]
```

SEE ALSO: ListCompo, ListCompoInPpart, ListCompoFiner

5.14.24 `Pairing` — computes the pairing between a quasi-symmetric function and a noncommutative symmetric function

✘

Calling Sequence:

`Pairing`(qsf,$ncsf$)

`Pairing`(qsf,$ncsf$)

`NCSF[Pairing]`(qsf,$ncsf$)

`NCSF[Pairing]`(qsf,$ncsf$)

Parameters:

qsf — any quasi-symmetric function

$ncsf$ — any noncommutative symmetric function

Synopsis:

- The `Pairing` function computes the pairing between a quasi-symmetric function and a noncommutative symmetric function. The pairing is defined by:

$$\langle F[I], R[J] \rangle = 1 \quad ,\text{if } I = J \quad ,$$

  and 0 otherwise, where I and J are two compositions, F is the dual basis of Ribbon Schur basis.

- qsf is any expression in terms of the basic quasi-symmetric functions.

- $ncsf$ is any expression in terms of the basic noncommutative symmetric functions.

- The qsf and the $ncsf$ functions are expanded. qsf (resp. $ncsf$) is expressed on F basis (resp. R basis). The result is not collected.

- One may specify by a second argument, say b, that qsf (idem $ncsf$) is solely expressed in terms of the known basis b.

- The call `Pairing`(qsf,$ncsf$,$'F'$) (resp. `Pairing`(qsf,$ncsf$,$'R'$)) does not affect the argument qsf (resp. $ncsf$).

- In first argument (resp. second argument), we consider as coefficient noncommutative symmetric functions (resp. quasi-symmetric functions).

- One may add *'noexpand'* just after the argument $ncsf$ to choose not to expand qsf and $ncsf$ before treating it.

- One may collect the result by adding an argument: `Pairing`(qsf,$ncsf$, `'collect'`).

- Whenever there is a conflict between the function name `Pairing` and another name used in the same session, use the long form `NCSF['Pairing']`.

✂ ....

Examples:

```
> with(NCSF):
> Pairing((1+q)^500*F[3,2],R[3,2]):                  # expands the input
> Pairing((1+q)^500*F[3,2],R[3,2],noexpand):         # does not expand (1+q)^500
> Pairing((1+q)^500*F[3,2],R[3,2],collect):          # collects the result
> Pairing((1+q)^500*F[3,2],R[3,2],'F','R',noexpand): # the most efficient
```

```
> Pairing(F[3]*R[3],F[3]*R[3]);
                                   F[3] R[3]

> Pairing((1-q)^30*F[3,2],R[3,2],noexpand,collect);

                                          30
                                   (1 - q)
```

SEE ALSO: ToR, ToF

5.14.25 PermDesc — generate all permutations of a given descent set

Calling Sequence:

PermDesc(*Compo*)

NCSF[PermDesc](*Compo*)

Parameters:

Perm — any composition

Synopsis:

- The PermDesc function generates all compositions of a given descent set.
- The result is a list of permutation in lexicographic order.
- Whenever there is a conflict between the function name PermDesc and another name used in the same session, use the long form NCSF['PermDesc'].

Examples:

```
> with(NCSF):
> PermDesc([3,2]);
   [[1, 2, 4, 3, 5], [1, 2, 5, 3, 4], [1, 3, 4, 2, 5], [1, 3, 5, 2, 4],

       [1, 4, 5, 2, 3], [2, 3, 4, 1, 5], [2, 3, 5, 1, 4], [2, 4, 5, 1, 3],

       [3, 4, 5, 1, 2]]
```

SEE ALSO:

5.14.26 Ph2LMat — transition matrix from Φ-basis to Λ-basis

Calling Sequence:

Ph2LMat(n)

NCSF[Ph2LMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The Ph2LMat function computes the transition matrix from the Φ-basis (power sums symmetric functions of the second kind) to the Λ-basis (elementary symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name Ph2LMat and another name used in the same session, use the long form NCSF['Ph2LMat'].

Examples:

```
> with(NCSF):
> Ph2LMat(3);
                              [ 3  -3/2  -3/2  1 ]
                              [                  ]
                              [ 0   -2     0   1 ]
                              [                  ]
                              [ 0    0    -2   1 ]
                              [                  ]
                              [ 0    0     0   1 ]
```

SEE ALSO: L2PhMat

5.14.27 Ph2PsMat — transition matrix from Φ-basis to Ψ-basis

Calling Sequence:

Ph2PsMat(n)

NCSF[Ph2PsMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The Ph2PsMat function computes the transition matrix from the Φ-basis (power sums symmetric functions of the second kind) to the Ψ-basis (power sums symmetric functions of the first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name Ph2PsMat and another name used in the same session, use the long form NCSF['Ph2PsMat'].

Examples:

```
> with(NCSF):
> Ph2PsMat(3);
                              [ 1  -1/4  1/4  0 ]
                              [                 ]
                              [ 0    1    0   0 ]
                              [                 ]
                              [ 0    0    1   0 ]
                              [                 ]
                              [ 0    0    0   1 ]
```

SEE ALSO: Ps2PhMat

5.14.28 Ph2RMat — transition matrix from Φ-basis to R-basis

Calling Sequence:

Ph2RMat(n)

NCSF[Ph2RMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The Ph2RMat function computes the transition matrix from the Φ-basis (power sums symmetric functions of the second kind) to the R-basis (ribbon Schur functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name Ph2RMat and another name used in the same session, use the long form NCSF['Ph2RMat'].

Examples:

```
> with(NCSF):
> Ph2RMat(3);

                        [ 1  -1/2  -1/2   1 ]
                        [                   ]
                        [ 1    1    -1   -1 ]
                        [                   ]
                        [ 1   -1     1   -1 ]
                        [                   ]
                        [ 1    1     1    1 ]
```

SEE ALSO: R2PhMat

5.14.29 Ph2SMat — transition matrix from Φ-basis to S-basis

Calling Sequence:

Ph2SMat(n)

NCSF[Ph2SMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The Ph2SMat function computes the transition matrix from the Φ-basis (power sums symmetric functions of the second kind) to the S-basis (complete symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name Ph2SMat and another name used in the same session, use the long form NCSF['Ph2SMat'].

Examples:

```
> with(NCSF):
> Ph2SMat(3);

                    [ 3  -3/2  -3/2   1 ]
                    [                   ]
                    [ 0    2     0   -1 ]
                    [                   ]
                    [ 0    0     2   -1 ]
                    [                   ]
                    [ 0    0     0    1 ]
```

SEE ALSO: `S2PhMat`

5.14.30 `Pleth` — compute the plethysm of symmetric function and a quasi-symmetric function

Calling Sequence:

`Pleth`(sf, qsf)

`Pleth`(sf, qsf)

`NCSF[Pleth]`(sf, qsf)

`NCSF[Pleth]`(sf, qsf)

Parameters:

sf — any symmetric function

qsf — any quasi-symmetric function

Synopsis:

- The `Pleth` function computes the plethysm of a symmetric function and a quasi-symmetric function. The plethysm is defined by:

$$\texttt{Pleth}(p_k, M[i_1, \ldots, i_r]) = M[k\,i_1, \ldots, k\,i_r] \quad .$$

- sf is any expression in terms of the basic symmetric functions.
- qsf is any expression in terms of the basic quasi-symmetric functions.
- The sf and the qsf functions are expanded and the result is not collected.
- One may specify by a second argument, say b, that sf (idem qsf) is solely expressed in terms of the known basis b.
- The call `Pleth`$(sf, qsf, 'L')$ (resp. `Pleth`$(sf, qsf, 'M')$) does not affect the argument sf (resp. qsf).
- One may add $'noexpand'$ just after the argument qsf to choose not to expand sf and qsf before treating it.
- One may collect the result by adding an argument: `Pleth`$(sf, qsf, 'collect')$.

- Whenever there is a conflict between the function name **Pleth** and another name used in the same session, use the long form NCSF['Pleth'].

Examples:

```
> with(NCSF):
> with(NCSF):
> Pleth((1+q)^500*s[3,2],F[1,1]):                   #   expands the input
> Pleth((1+q)^500*s[3,2],F[1,1],noexpand):          # does not expand (1+q)^500
> Pleth((1+q)^500*s[3,2],F[1,1],collect):           # collects the result
> Pleth((1+q)^500*s[3,2],F[1,1],'s','F',noexpand):  # the most efficient
> Pleth(q*s[3],M[1,1],noexpand,collect);

     F[1, 1, 2, 1, 1] q + F[1, 1, 2, 2] q + F[2, 2, 2] q + F[2, 2, 1, 1] q

      + F[2, 3, 1] q + F[1, 3, 2] q + F[1, 1, 1, 2, 1] q + F[2, 1, 1, 2] q

          + F[1, 2, 1, 2] q + F[1, 2, 1, 1, 1] q + F[1, 1, 1, 1, 1, 1] q

             + F[3, 3] q + F[2, 1, 2, 1] q + 2 F[1, 2, 2, 1] q
```

SEE ALSO:

5.14.31 Ps2LMat — transition matrix from Ψ-basis to Λ-basis

Calling Sequence:

Ps2LMat(n)

NCSF[Ps2LMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The Ps2LMat function computes the transition matrix from the Ψ-basis (power sums symmetric functions of the first kind) to the Λ-basis (elementary symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name **Ps2LMat** and another name used in the same session, use the long form NCSF['Ps2LMat'].

Examples:

```
> with(NCSF):
> Ps2LMat(3);

                              [ 3  -2  -1  1 ]
                              [              ]
                              [ 0  -2   0  1 ]
                              [              ]
```

```
[ 0   0  -2  1 ]
[              ]
[ 0   0   0  1 ]
```

SEE ALSO: L2PsMat

5.14.32 Ps2PhMat — transition matrix from Ψ-basis to Φ-basis

Calling Sequence:

Ps2PhMat(n)

NCSF[Ps2PhMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The Ps2PhMat function computes the transition matrix from the Ψ-basis (power sums symmetric functions of the first kind) to the Φ-basis (power sums symmetric functions of the second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name Ps2PhMat and another name used in the same session, use the long form NCSF['Ps2PhMat'].

Examples:

```
> with(NCSF):
> Ps2PhMat(3);
```

```
[ 1  1/4  -1/4  0 ]
[                 ]
[ 0   1     0   0 ]
[                 ]
[ 0   0     1   0 ]
[                 ]
[ 0   0     0   1 ]
```

SEE ALSO: Ph2PsMat

5.14.33 Ps2RMat — transition matrix from Ψ-basis to R-basis

Calling Sequence:

Ps2RMat(n)

NCSF[Ps2RMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The `Ps2RMat` function computes the transition matrix from the Ψ-basis (power sums symmetric functions of the first kind) to the R-basis (ribbon Schur functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `Ps2RMat` and another name used in the same session, use the long form `NCSF['Ps2RMat']`.

Examples:

```
> with(NCSF):
> Ps2RMat(3);
                    [ 1   0  -1   1 ]
                    [               ]
                    [ 1   1  -1  -1 ]
                    [               ]
                    [ 1  -1   1  -1 ]
                    [               ]
                    [ 1   1   1   1 ]
```

SEE ALSO: `R2PsMat`

5.14.34 `Ps2SMat` — transition matrix from Ψ-basis to S-basis

Calling Sequence:

`Ps2SMat`(n)

`NCSF[Ps2SMat]`(n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The `Ps2SMat` function computes the transition matrix from the Ψ-basis (power sums symmetric functions of the first kind) to the S-basis (complete symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `Ps2SMat` and another name used in the same session, use the long form `NCSF['Ps2SMat']`.

Examples:

```
> with(NCSF):
> Ps2SMat(3);
                    [ 3  -1  -2   1 ]
                    [               ]
                    [ 0   2   0  -1 ]
                    [               ]
                    [ 0   0   2  -1 ]
                    [               ]
                    [ 0   0   0   1 ]
```

SEE ALSO: `S2PsMat`

5.14.35 QPh2EMat — transition matrix from QPh-basis to E-basis

Calling Sequence:

QPh2EMat(n)

NCSF[QPh2EMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The QPh2EMat function computes the transition matrix from the QPh-basis (dual basis of non-commutative power sums symmetric functions of the second kind) to the E-basis (elementary quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name QPh2EMat and another name used in the same session, use the long form NCSF['QPh2EMat'].

Examples:

```
> with(NCSF):
> QPh2EMat(3);

                    [  1   0   0  0 ]
                    [               ]
                    [ -1  -2   0  0 ]
                    [               ]
                    [ -1   0  -2  0 ]
                    [               ]
                    [  1   3   3  6 ]
```

SEE ALSO: E2QPhMat

5.14.36 QPh2FMat — transition matrix from QPh-basis to F-basis

Calling Sequence:

QPh2FMat(n)

NCSF[QPh2FMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The QPh2FMat function computes the transition matrix from the QPh-basis (dual basis of non-commutative power sums symmetric functions of the second kind) to the F-basis (ribbon Schur quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name QPh2FMat and another name used in the same session, use the long form NCSF['QPh2FMat'].

Examples:

```
> with(NCSF):
> QPh2FMat(3);
```

```
[ 1  -1  -1   1 ]
[               ]
[ 1   1  -1  -1 ]
[               ]
[ 1  -1   1  -1 ]
[               ]
[ 1   2   2   1 ]
```

SEE ALSO: F2QPhMat

5.14.37 QPh2MMat — transition matrix from QPh-basis to M-basis

✘

Calling Sequence:

QPh2MMat(n)

NCSF[QPh2MMat](n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The QPh2MMat function computes the transition matrix from the QPh-basis (dual basis of noncommutative power sums symmetric functions of the second kind) to the M-basis (monomial quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name QPh2MMat and another name used in the same session, use the long form NCSF['QPh2MMat'].

Examples:

```
> with(NCSF):
> QPh2MMat(3);
```

```
[ 1  0  0  0 ]
[            ]
[ 1  2  0  0 ]
[            ]
[ 1  0  2  0 ]
[            ]
[ 1  3  3  6 ]
```

SEE ALSO: M2QPhMat

5.14.38 QPh2QPsMat — transition matrix from QPh-basis to QPs-basis

Calling Sequence:

QPh2QPsMat(n)

NCSF[QPh2QPsMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The QPh2QPsMat function computes the transition matrix from the QPh-basis (dual basis of noncommutative power sums symmetric functions of the second kind) to the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name QPh2QPsMat and another name used in the same session, use the long form NCSF['QPh2QPsMat'].

Examples:

```
> with(NCSF):
> QPh2QPsMat(3);
```

```
                    [   1    0    0   0 ]
                    [                   ]
                    [  1/3  2/3   0   0 ]
                    [                   ]
                    [ -1/3   0   4/3  0 ]
                    [                   ]
                    [   0    0    0   1 ]
```

SEE ALSO: QPs2QPhMat

5.14.39 QPs2EMat — transition matrix from QPs-basis to E-basis

Calling Sequence:

QPs2EMat(n)

NCSF[QPs2EMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The QPs2EMat function computes the transition matrix from the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind) to the E-basis (elementary quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name QPs2EMat and another name used in the same session, use the long form NCSF['QPs2EMat'].

Examples:

```
> with(NCSF):
> QPs2EMat(3);

                    [   1     0     0    0 ]
                    [                      ]
                    [  -2    -3     0    0 ]
                    [                      ]
                    [ -1/2    0   -3/2   0 ]
                    [                      ]
                    [   1     3     3    6 ]
```

SEE ALSO: E2QPsMat

5.14.40 QPs2FMat — transition matrix from QPs-basis to F-basis

Calling Sequence:

QPs2FMat(n)

NCSF[QPs2FMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The QPs2FMat function computes the transition matrix from the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind) to the F-basis (ribbon Schur quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name QPs2FMat and another name used in the same session, use the long form NCSF['QPs2FMat'].

Examples:

```
> with(NCSF):
> QPs2FMat(3);

                    [ 1  -1   -1     1  ]
                    [                   ]
                    [ 1   2   -1    -2  ]
                    [                   ]
                    [ 1  -1   1/2  -1/2 ]
                    [                   ]
                    [ 1   2    2     1  ]
```

SEE ALSO: F2QPsMat

5.14.41 `QPs2MMat` — transition matrix from QPs-basis to M-basis

Calling Sequence:

`QPs2MMat`(n)

`NCSF[QPs2MMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The **`QPs2MMat`** function computes the transition matrix from the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind) to the M-basis (monomial quasi-symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name **`QPs2MMat`** and another name used in the same session, use the long form `NCSF['QPs2MMat']`.

Examples:

```
> with(NCSF):
> QPs2MMat(3);
```

```
                    [ 1  0   0   0 ]
                    [              ]
                    [ 1  3   0   0 ]
                    [              ]
                    [ 1  0  3/2  0 ]
                    [              ]
                    [ 1  3   3   6 ]
```

SEE ALSO: `M2QPsMat`

5.14.42 `QPs2QPhMat` — transition matrix from QPs-basis to QPh-basis

Calling Sequence:

`QPs2QPhMat`(n)

`NCSF[QPs2QPhMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The **`QPs2QPhMat`** function computes the transition matrix from the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind) to the QPh-basis (dual basis of noncommutative power sums symmetric functions of the second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.

- Whenever there is a conflict between the function name QPs2QPhMat and another name used in the same session, use the long form NCSF['QPs2QPhMat'].

Examples:

```
> with(NCSF):
> QPs2QPhMat(3);
```

```
                    [   1     0     0   0 ]
                    [                     ]
                    [ -1/2  3/2     0   0 ]
                    [                     ]
                    [  1/4    0   3/4   0 ]
                    [                     ]
                    [   0     0     0   1 ]
```

SEE ALSO: QPh2QPsMat

5.14.43 Qsf2x — from quasi-symmetric function to the basis of monomials

Calling Sequence:

Qsf2x(qsf, var)

NCSF[Qsf2x](qsf, var)

Parameters:

qsf — any quasi-symmetric function

var — a list of variables

Synopsis:

- The tt Qsf2x functions converts a quasi-symmetric function to the basis of monomials.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may add 'noexpand' just after the argument var to choose not to expand the quasi-symmetric function qsf before treating it.
- Whenever there is a conflict between the function name Qsf2x and another name used in the same session, use the long form NCSF['Qsf2x'].

Examples:

```
> with(NCSF):
> Qsf2x(F[2]&*M[1,1],[x1,x2,x3]);
     3        3        3          2                 2        2   2      2   2
   x1  x2 + x1  x3 + x2  x3 + 3 x1  x2 x3 + 3 x1 x2  x3 + x1  x2  + x1  x3

            2   2             2        3        3        3
        + x2  x3  + 3 x1 x2 x3  + x1 x2  + x1 x3  + x2 x3
```

SEE ALSO: ToE, ToQPh, ToQPs, ToF, ToM

5.14.44 `R2LMat` — transition matrix from R-basis to Λ-basis

Calling Sequence:

`R2LMat`(n)

`NCSF[R2LMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The `R2LMat` function computes the transition matrix from the R-basis (ribbon Schur functions) to the Λ-basis (elementary symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `R2LMat` and another name used in the same session, use the long form `NCSF['R2LMat']`.

Examples:

```
> with(NCSF):
> R2LMat(3);
                        [  1  -1  -1  1 ]
                        [               ]
                        [ -1   0   1  0 ]
                        [               ]
                        [ -1   1   0  0 ]
                        [               ]
                        [  1   0   0  0 ]
```

SEE ALSO: `L2RMat`

5.14.45 `R2PhMat` — transition matrix from R-basis to Φ-basis

Calling Sequence:

`R2PhMat`(n)

`NCSF[R2PhMat]`(n)

Parameters:

n — any positive integer $(n > 1)$

Synopsis:

- The `R2PhMat` function computes the transition matrix from the R-basis (ribbon Schur functions) to the Φ-basis (power sums symmetric functions of the second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `R2PhMat` and another name used in the same session, use the long form `NCSF['R2PhMat']`.

Examples:

```
> with(NCSF):
> R2PhMat(3);

                    [  1/3   1/4   1/4  1/6 ]
                    [                       ]
                    [ -1/3   1/4  -1/4  1/3 ]
                    [                       ]
                    [ -1/3  -1/4   1/4  1/3 ]
                    [                       ]
                    [  1/3  -1/4  -1/4  1/6 ]
```

SEE ALSO: Ph2RMat

5.14.46 R2PsMat — transition matrix from R-basis to Ψ-basis

Calling Sequence:

R2PsMat(n)

NCSF[R2PsMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The R2PsMat function computes the transition matrix from the R-basis (ribbon Schur functions) to the Ψ-basis (power sums symmetric functions of the first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name R2PsMat and another name used in the same session, use the long form NCSF['R2PsMat'].

Examples:

```
> with(NCSF):
> R2PsMat(3);

                    [  1/3   1/6   1/3  1/6 ]
                    [                       ]
                    [ -1/3   1/3  -1/3  1/3 ]
                    [                       ]
                    [ -1/3  -1/6   1/6  1/3 ]
                    [                       ]
                    [  1/3  -1/3  -1/6  1/6 ]
```

SEE ALSO: Ps2RMat

5.14.47 R2SMat — transition matrix from R-basis to S-basis

Calling Sequence:

R2SMat(n)

NCSF[R2SMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The R2SMat function computes the transition matrix from the R-basis (ribbon Schur functions) to the S-basis (complete symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name R2SMat and another name used in the same session, use the long form NCSF['R2SMat'].

Examples:

```
> with(NCSF):
> R2SMat(3);
```

```
[  1   0   0  0 ]
[               ]
[ -1   1   0  0 ]
[               ]
[ -1   0   1  0 ]
[               ]
[  1  -1  -1  1 ]
```

SEE ALSO: S2RMat

5.14.48 S2LMat — transition matrix from S-basis to Λ-basis

Calling Sequence:

S2LMat(n)

NCSF[S2LMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The S2LMat function computes the transition matrix from the S-basis (complete symmetric functions) to the Λ-basis (elementary symmetric functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name S2LMat and another name used in the same session, use the long form NCSF['S2LMat'].

Examples:

```
> with(NCSF):
> S2LMat(3);

                              [ 1  -1  -1  1 ]
                              [              ]
                              [ 0  -1   0  1 ]
                              [              ]
                              [ 0   0  -1  1 ]
                              [              ]
                              [ 0   0   0  1 ]
```

SEE ALSO: L2SMat

5.14.49 S2PhMat — transition matrix from S-basis to Φ-basis

Calling Sequence:

S2PhMat(n)

NCSF[S2PhMat](n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The S2PhMat function computes the transition matrix from the S-basis (complete symmetric functions) to the Φ-basis (power sums symmetric functions of the second kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name S2PhMat and another name used in the same session, use the long form NCSF['S2PhMat'].

Examples:

```
> with(NCSF):
> S2PhMat(3);

                         [ 1/3  1/4  1/4  1/6 ]
                         [                    ]
                         [  0   1/2   0   1/2 ]
                         [                    ]
                         [  0    0   1/2  1/2 ]
                         [                    ]
                         [  0    0    0    1  ]
```

SEE ALSO: Ph2SMat

5.14.50 `S2PsMat` — transition matrix from S-basis to Ψ-basis

Calling Sequence:

`S2PsMat`(n)

`NCSF[S2PsMat]`(n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The `S2PsMat` function computes the transition matrix from the S-basis (complete symmetric functions) to the Ψ-basis (power sums symmetric functions of the first kind).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `S2PsMat` and another name used in the same session, use the long form `NCSF['S2PsMat']`.

Examples:

```
> with(NCSF):
> S2PsMat(3);

                         [ 1/3  1/6  1/3  1/6 ]
                         [                    ]
                         [  0   1/2   0   1/2 ]
                         [                    ]
                         [  0    0   1/2  1/2 ]
                         [                    ]
                         [  0    0    0    1  ]
```

SEE ALSO: `Ps2SMat`

5.14.51 `S2RMat` — transition matrix from S-basis to R-basis

Calling Sequence:

`S2RMat`(n)

`NCSF[S2RMat]`(n)

Parameters:

n — any positive integer ($n > 1$)

Synopsis:

- The `S2RMat` function computes the transition matrix from the S-basis (complete symmetric functions) to the R-basis (ribbon Schur functions).
- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name `S2RMat` and another name used in the same session, use the long form `NCSF['S2RMat']`.

Examples:

```
> with(NCSF):
> S2RMat(3);
```

```
                                   [ 1  0  0  0 ]
                                   [            ]
                                   [ 1  1  0  0 ]
                                   [            ]
                                   [ 1  0  1  0 ]
                                   [            ]
                                   [ 1  1  1  1 ]
```

SEE ALSO: `R2SMat`

5.14.52 `Sf2Qsf` — expand a symmetric function on F basis (dual basis of noncommutative ribbon Schur function)

✘

Calling Sequence:

`Sf2Qsf`(sf)

`NCSF[Sf2Qsf]`(sf)

Parameters:

sf — any symmetric function

Synopsis:

- The `Sf2Qsf` function computes the expansion of sf on F-basis (dual basis of noncommutative ribbon Schur function)
- The result is expressed on F-basis
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may add `'noexpand'` just after sf to choose not to expand the symmetric function sf before treating it.
- One may add b just after sf or after the argument `noexpand` to specify that the input is expressed on b-basis.

  - One may collect the result by adding a third argument: `Sf2Qsf`(sf, `'collect'`).
- Whenever there is a conflict between the function name `Sf2Qsf` and another name used in the same session, use the long form `NCSF['Sf2Qsf']`.

Examples:

```
> with(NCSF):
> Sf2Qsf((1-q)^30*s[2,1]):             # expand the input
> Sf2Qsf((1-q)^30*s[2,1],noexpand):    # does not expand the input
> Sf2Qsf((1-q)^30*s[2,1],collect):     # collect the result
```

```
> Sf2Qsf((1-q)^30*s[2,1]+s[3],noexpand,collect);

                     30                  30
              (1 - q)   F[2, 1] + (1 - q)   F[1, 2] + F[3]
```

SEE ALSO: `Pairing, ToF`

5.14.53 `Shuffle` — compute the shuffle of two words

Calling Sequence:

`Shuffle`(u, v)

`NCSF[Shuffle]`(u, v)

Parameters:

u, v — two words represented by lists

Synopsis:

- The `Shuffle` function computes the shuffle of two words. We use the following definition:

$$\texttt{Shuffle}(a\,u, b\,v) = a\,\texttt{Shuffle}(u, b\,v) + b\,\texttt{Shuffle}(a\,u, v) \quad ,$$

  and `Shuffle`$(1, a)$ = `Shuffle`$(a, 1) = a$, where u and v are two words, a and b two letters, 1 is the empty word.

- The result is a sums of words.

- The argument are expanded and the result is not collected.

- One may add `'noexpand'` just after the arguments to choose not to expand.

- One may add a formal parameter just after the arguments or after the argument `noexpand` to specify that we compute the q-shuffle defined by:

$$\texttt{Shuffle}(a\,u, b\,v, q) = a\,\texttt{Shuffle}(u, b\,v, q) + q^{|a\,u|}b\,\texttt{Shuffle}(a\,u, v, q) \quad .$$

- One may collect the result by adding a third argument: `Shuffle`$(u, v,$ `'collect'`).

- Whenever there is a conflict between the function name `Shuffle` and another name used in the same session, use the long form `NCSF['Shuffle']`.

Examples:

```
> with(NCSF):
> Shuffle((1-q)^30*[2,1],[2,1]):             # expand the input
> Shuffle((1-q)^30*[2,1],[2,1],noexpand):    # does not expand the input
> Shuffle((1-q)^30*[2,1],[2,1],collect):     # collect the result
> Shuffle((1-q)^30*[2,1],[2,1],noexpand,'q');
                     30
              (1 - q)   (2 [2, 1, 2, 1] + 4 [2, 2, 1, 1])

> Shuffle((1-q)^30*[2,1],[2,1],noexpand,collect);
                   30                            30
```

```
             2 (1 - q)    [2, 1, 2, 1] + 4 (1 - q)    [2, 2, 1, 1]

> Shuffle((1-q)^30*[2,1],[2,1],noexpand,'q',collect);
            30       4                        30         2    3
    (1 - q)    (1 + q ) [2, 1, 2, 1] + (1 - q)    (q + 2 q  + q ) [2, 2, 1, 1]
```

SEE ALSO: ToF

5.14.54 SpDirect — specialization $A \longrightarrow A/(1-q)$

Calling Sequence:

SpDirect$(ncsf, q)$

NCSF[SpDirect]$(ncsf, q)$

Parameters:

$ncsf$ — any noncommutative symmetric function

q — any formal parameter

Synopsis:

- The SpDirect function computes the transformation $A \longrightarrow A/(1-q)$ where A is a set of variables, on the non commutative symmetric function $ncsf$.
- Whenever there is a conflict between the function name SpDirect and another name used in the same session, use the long form NCSF['SpDirect'].

Examples:

```
> with(NCSF):
> SpDirect(Ps[2,1], 'r');

                   S[2, 1]              S[1, 1, 1]
         2 -------------------- - --------------------
                    2                      2
           (- 1 + r ) (- 1 + r)   (- 1 + r ) (- 1 + r)

> SpDirect(q^2*Ph[2,2], 't');

              2                 2                  2                  2
    S[2, 2] q      S[2, 1, 1] q      S[1, 1, 2] q    S[1, 1, 1, 1] q
  4 ----------- - 2 ------------- - 2 ------------- + ----------------
            2 2               2 2                2 2                2 2
    (- 1 + t )       (- 1 + t )        (- 1 + t )         (- 1 + t )
```

SEE ALSO: SpInverse

5.14.55 SpInverse — specialization $A \longrightarrow A\,(1-q)$

Calling Sequence:

SpInverse($ncsf, q$)

NCSF[SpInverse]($ncsf, q$)

Parameters:

$ncsf$ — any noncommutative symmetric function

q — any formal parameter

Synopsis:

- The SpInverse function computes the transformation $A \longrightarrow A\,(1-q)$ where A is a set of variables, on the non commutative symmetric function $ncsf$.
- We recall that $S^n((1-q)\,A) = (1-q)\sum_{k=0}^{n-1}(-q)^k R_{1^k,n-k}$.
- Whenever there is a conflict between the function name SpInverse and another name used in the same session, use the long form NCSF['SpInverse'].

Examples:

```
> with(NCSF):
> SpInverse(S[3], 'q');

                                                        2
  - (- 1 + q) R[3] + (- 1 + q) q R[1, 2] - (- 1 + q) q  R[1, 1, 1]

> SpInverse(L[2,1] + t*R[1,1,1], 'q');

                             2    2                    2
     - q (1 - 2 q - t q + t q  + q ) R[3] - (- 1 + q)  q R[2, 1]

  + (- 1 + q) (q + t q - 1) R[1, 2] - (- 1 + q) (t - q + 1) R[1, 1, 1]
```

SEE ALSO: Internal, SpDirect

5.14.56 ToE — express any quasi-symmetric function in the E-basis

Calling Sequence:

ToE(qsf)

ToE(qsf, b)

NCSF[ToE](qsf)

NCSF[ToE](qsf, b)

Parameters:

qsf — any quasi-symmetric function

b — any name of a known basis

Synopsis:

- The `ToE` function computes the expansion of qsf in the E-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call `ToE`(qsf, `'E'`) does not affect the argument qsf.
- One may add `'noexpand'` just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: `ToE`(qsf, b, `'collect'`). For instance, `ToE`(qsf, `'E'`, `'collect'`) may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToE` and another name used in the same session, use the long form `NCSF['ToE']`.

Examples:

```
> with(NCSF):
> ToE((1+q)^500*E[2,3]):                    # expands the input
> ToE((1+q)^500*E[2,3],noexpand):           # does not expand (1+q)^500
> ToE((1+q)^500*E[2,3],collect):            # collects the result
> ToE((1+q)^500*E[2,3],noexpand,'E'):       # the most efficient
> ToE((1+q)^500*E[2,3],'E',collect):        # specifies a basis
> ToE((1+q)^500*E[2,3],noexpand,'E',collect):
> ToE((1-q)^50*F[1,2,1]+QPh[2]&*F[1,1],noexpand,collect);
              50                                50
    ((1 - q)    - 2) E[2, 2] + (- 1 + (1 - q)  ) E[2, 1, 1]

                          50                      50
         + (- 1 + (1 - q)  ) E[1, 1, 2] + (1 - q)   E[1, 1, 1, 1] - E[1, 3]

         - E[1, 2, 1] - E[4] - E[3, 1]
```

SEE ALSO: `ToQPh, ToQPs, ToF, ToM`

5.14.57 `ToF` — express any quasi-symmetric function in the F-basis

✘

Calling Sequence:

`ToF`(qsf)

`ToF`(qsf, b)

`NCSF[ToF]`(qsf)

`NCSF[ToF]`(qsf, b)

Parameters:

qsf — any quasi-symmetric function

b — any name of a known basis

Synopsis:

- The `ToF` function computes the expansion of qsf in the F-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call `ToF`(qsf, `'F'`) does not affect the argument qsf.
- One may add `'noexpand'` just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: `ToF`(qsf, b, `'collect'`). For instance, `ToF`(qsf, `'F'`, `'collect'`) may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToF` and another name used in the same session, use the long form `NCSF['ToF']`.

Examples:

```
> with(NCSF):
> ToF((1+q)^500*F[2,3]):                     # expands the input
> ToF((1+q)^500*F[2,3],noexpand):            # does not expand (1+q)^500
> ToF((1+q)^500*F[2,3],collect):             # collects the result
> ToF((1+q)^500*F[2,3],noexpand,'F'):        # the most efficient
> ToF((1+q)^500*F[2,3],'F',collect):         # specifies a basis
> ToF((1+q)^500*QPh[2,1]+t*M[3],noexpand,collect);
                       500                                500
      (- 1/2 (1 + q)    + t) F[1, 1, 1] + (1/2 (1 + q)    + t) F[3]

                             500                                500
           + (1/2 (1 + q)    - t) F[2, 1] + (- 1/2 (1 + q)    - t) F[1, 2]
```

SEE ALSO: `ToE, ToPs, ToF, ToM, Shuffle`

5.14.58 `ToL` — express any noncommutative symmetric function in the Λ-basis

Calling Sequence:

`ToL`($ncsf$)

`NCSF[ToL]`($ncsf$)

Parameters:

$ncsf$ — any noncommutative symmetric function

Synopsis:

- The `ToL` function computes the expansion of *ncsf* in the Λ-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function *ncsf* is expanded and the result is not collected.
- One may specify by a second argument, say b, that *ncsf* is solely expressed in terms of the known basis b.
- The call `ToL`(*ncsf*, `'L'`) does not affect the argument *ncsf*.
- One may add `'noexpand'` just after the argument *ncsf* to choose not to expand the noncommutative symmetric function *ncsf* before treating it.
- One may collect the result by adding a third argument: `ToL`(*ncsf*, b, `'collect'`). For instance, `ToL`(*ncsf*, `'L'`, `'collect'`) may be used to collect the argument *ncsf*.
- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToL` and another name used in the same session, use the long form `NCSF['ToL']`.

Examples:

```
> with(NCSF):
> f:=(1-q^2)*Ps[2,1]:
> g:=2*R[2,1]:
> ToL(f &* g);
                  2                          2
      - 4 (- 1 + q ) L[2, 1, 3] + 4 (- 1 + q ) L[2, 1, 1, 2]

                2                            2
    + 2 (- 1 + q ) L[1, 1, 1, 3] - 2 (- 1 + q ) L[1, 1, 1, 1, 2]

> ToL(g &* f);

                  2                          2
      - 4 (- 1 + q ) L[3, 2, 1] + 2 (- 1 + q ) L[3, 1, 1, 1]

                2                            2
    + 4 (- 1 + q ) L[1, 2, 2, 1] - 2 (- 1 + q ) L[1, 2, 1, 1, 1]

> ToL( Internal((1-q)*g, S[1,1,1]) );

                        - 4 (- 1 + q) L[1, 1, 1]
```

SEE ALSO: `ToPh, ToPs, ToR, ToS`

5.14.59 ToM — express any quasi-symmetric function in the M-basis

Calling Sequence:

ToM(qsf)

ToM(qsf, b)

NCSF[ToM](qsf)

NCSF[ToM](qsf, b)

Parameters:

qsf — any quasi-symmetric function

b — any name of a known basis

Synopsis:

- The ToM function computes the expansion of qsf in the M-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call ToM(qsf, 'M') does not affect the argument qsf.
- One may add **'noexpand'** just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: ToM(qsf, b, 'collect'). For instance, ToM(qsf, 'M', 'collect') may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToM and another name used in the same session, use the long form NCSF['ToM'].

Examples:

```
> with(NCSF):
> ToM((1+q)^500*M[2,3]):                        # expands the input
> ToM((1+q)^500*M[2,3],noexpand):               # does not expand (1+q)^500
> ToM((1+q)^500*M[2,3],collect):                # collects the result
> ToM((1+q)^500*M[2,3],noexpand,'M'):       # the most efficient
> ToM((1+q)^500*M[2,3],'M',collect):        # specifies a basis
> ToM((1+q)^500*QPh[2,1,2]+t^3*M[3,2],noexpand,collect);
                   500                    500                     500
      1/6 (1 + q)    M[5] + 1/2 (1 + q)    M[2, 3] + (1 + q)    M[2, 1, 2]

                             500    3
           + (1/2 (1 + q)    + t ) M[3, 2]
```

SEE ALSO: ToE, ToPs, ToF, ToM

5.14.60 ToPh — express any noncommutative symmetric function in the Φ-basis

Calling Sequence:

ToPh($ncsf$)

NCSF[ToPh]($ncsf$)

Parameters:

$ncsf$ — any noncommutative symmetric function

Synopsis:

- The ToPh function computes the expansion of $ncsf$ in the Φ-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function $ncsf$ is expanded and the result is not collected.
- One may specify by a second argument, say b, that $ncsf$ is solely expressed in terms of the known basis b.
- The call ToPh($ncsf$, 'Ph') does not affect the argument $ncsf$.
- One may add 'noexpand' just after the argument $ncsf$ to choose not to expand the noncommutative symmetric function $ncsf$ before treating it.
- One may collect the result by adding a third argument: ToPh($ncsf, b$, 'collect'). For instance, ToPh($ncsf$, 'Ph', 'collect') may be used to collect the argument $ncsf$.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToPh and another name used in the same session, use the long form NCSF['ToPh'].

Examples:

```
> with(NCSF):
> f:=(1-q^2)*Ps[2,1]:
> g:=2*R[2,1]:
> ToPh(f &* g);

                    2                             2
      2/3 (- 1 + q ) Ph[2, 1, 3] - 1/2 (- 1 + q ) Ph[2, 1, 2, 1]

                  2                               2
   + 1/2 (- 1 + q ) Ph[2, 1, 1, 2] - 2/3 (- 1 + q ) Ph[2, 1, 1, 1, 1]

> ToPh(g &* f);

                    2                             2
      2/3 (- 1 + q ) Ph[3, 2, 1] - 1/2 (- 1 + q ) Ph[2, 1, 2, 1]

                  2                               2
   + 1/2 (- 1 + q ) Ph[1, 2, 2, 1] - 2/3 (- 1 + q ) Ph[1, 1, 1, 2, 1]
```

SEE ALSO: ToL, ToPs, ToR, ToS

5.14.61 ToPs — express any noncommutative symmetric function in the Ψ-basis

Calling Sequence:

ToPs($ncsf$)

NCSF[ToPs]($ncsf$)

Parameters:

$ncsf$ — any noncommutative symmetric function

Synopsis:

- The ToPs function computes the expansion of $ncsf$ in the Ψ-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function $ncsf$ is expanded and the result is not collected.
- One may specify by a second argument, say b, that $ncsf$ is solely expressed in terms of the known basis b.
- The call ToPs($ncsf$, 'Ps') does not affect the argument $ncsf$.
- One may add 'noexpand' just after the argument $ncsf$ to choose not to expand the noncommutative symmetric function $ncsf$ before treating it.
- One may collect the result by adding a third argument: ToPs($ncsf, b$, 'collect'). For instance, ToPs($ncsf$, 'Ps', 'collect') may be used to collect the argument $ncsf$.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToPs and another name used in the same session, use the long form NCSF['ToPs'].

Examples:

```
> with(NCSF):
> f:=(1-q^2)*Ps[2,1]:
> g:=2*R[2,1]:
> ToPs(f &* g);

                  2                            2
       2/3 (- 1 + q ) Ps[2, 1, 3] - 2/3 (- 1 + q ) Ps[2, 1, 2, 1]

                 2                               2
    + 2/3 (- 1 + q ) Ps[2, 1, 1, 2] - 2/3 (- 1 + q ) Ps[2, 1, 1, 1, 1]

> ToPs(g &* f);

                  2                            2
       2/3 (- 1 + q ) Ps[3, 2, 1] - 2/3 (- 1 + q ) Ps[2, 1, 2, 1]

                 2                               2
    + 2/3 (- 1 + q ) Ps[1, 2, 2, 1] - 2/3 (- 1 + q ) Ps[1, 1, 1, 2, 1]
```

See Also: ToL, ToPh, ToR, ToS

5.14.62 ToQPh — express any quasi-symmetric function in the QPh-basis

Calling Sequence:

ToQPh(qsf)

ToQPh(qsf, b)

NCSF[ToQPh](qsf)

NCSF[ToQPh](qsf, b)

Parameters:

qsf — any quasi-symmetric function

b — any name of a known basis

Synopsis:

- The **ToQPh** function computes the expansion of qsf in the QPh-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call **ToQPh**(qsf, **'QPh'**) does not affect the argument qsf.
- One may add **'noexpand'** just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: **ToQPh**(qsf, b, **'collect'**). For instance, **ToQPh**(qsf, **'QPh'**, **'collect'**) may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the **&*** operator.
- Whenever there is a conflict between the function name **ToQPh** and another name used in the same session, use the long form **NCSF['ToQPh']**.

Examples:

```
> with(NCSF):
> ToQPh((1+q)^500*QPh[2,3]):                  # expands the input
> ToQPh((1+q)^500*QPh[2,3],noexpand):         # does not expand (1+q)^500
> ToQPh((1+q)^500*QPh[2,3],collect):          # collects the result
> ToQPh((1+q)^500*QPh[2,3],noexpand,'QPh'):   # the most efficient
> ToQPh((1+q)^500*QPh[2,3],'QPh',collect):    # specifies a basis
> ToQPh((1+q)^500*QPh[2,3]+M[4,1]+t^3*M[3,2],noexpand,collect);
            500                                3                 3
   (1 + q)    QPh[2, 3] + QPh[4, 1] + (- 1/2 t  - 1/2) QPh[5] + t  QPh[3, 2]
```

SEE ALSO: ToE, ToPs, ToF, ToM

5.14.63 `ToQPs` — express any quasi-symmetric function in the QPs-basis

Calling Sequence:

`ToQPs`(qsf)

`ToQPs`(qsf, b)

`NCSF[ToQPs]`(qsf)

`NCSF[ToQPs]`(qsf, b)

Parameters:

qsf — any quasi-symmetric function

b — any name of a known basis

Synopsis:

- The `ToQPs` function computes the expansion of qsf in the QPs-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call `ToQPs`(qsf, `'QPs'`) does not affect the argument qsf.
- One may add `'noexpand'` just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: `ToQPs`(qsf, b, `'collect'`). For instance, `ToQPs`(qsf, `'QPs'`, `'collect'`) may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToQPs` and another name used in the same session, use the long form `NCSF['ToQPs']`.

Examples:

```
> with(NCSF):
> ToQPs((1+q)^500*QPs[2,3]):                      # expands the input
> ToQPs((1+q)^500*QPs[2,3],noexpand):             # does not expand (1+q)^500
> ToQPs((1+q)^500*QPs[2,3],collect):              # collects the result
> ToQPs((1+q)^500*QPs[2,3],noexpand,'QPs'):       # the most efficient
> ToQPs((1+q)^500*QPs[2,3],'QPs',collect):        # specifies a basis
> ToQPs((1+q)^500*QPh[2,3]+M[4,1]+t^3*M[3,2],noexpand,collect);
                      3                      3               500
  1/5 QPs[4, 1] + 2/5 t  QPs[3, 2] + (- 2/5 t  - 1/10 (1 + q)    - 1/5) QPs[5]

                   500
       + 3/5 (1 + q)    QPs[2, 3]
```

SEE ALSO: `ToE, ToQPh, ToF, ToM`

5.14.64 ToR — express any noncommutative symmetric function in the R-basis

Calling Sequence:

ToR(*ncsf*)

NCSF[ToR](*ncsf*)

Parameters:

ncsf — any noncommutative symmetric function

Synopsis:

- The ToR function computes the expansion of *ncsf* in the ribbon Schur functions basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function *ncsf* is expanded and the result is not collected.
- One may specify by a second argument, say b, that *ncsf* is solely expressed in terms of the known basis b.
- The call ToR(*ncsf*, 'R') does not affect the argument *ncsf*.
- One may add **'noexpand'** just after the argument *ncsf* to choose not to expand the noncommutative symmetric function *ncsf* before treating it.
- One may collect the result by adding a third argument: ToR(*ncsf*, *b*, 'collect'). For instance, ToR(*ncsf*, 'R', 'collect') may be used to collect the argument *ncsf*.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToR and another name used in the same session, use the long form NCSF['ToR'].

Examples:

```
> with(NCSF):
> f:=(1-q^2)*S[2,1]:
> g:=2*L[3]:
> ToR(f &* g);
                      2                          2
         - 2 (- 1 + q ) R[4, 1, 1] - 2 (- 1 + q ) R[3, 1, 1, 1]

                    2                             2
       - 2 (- 1 + q ) R[2, 2, 1, 1] - 2 (- 1 + q ) R[2, 1, 1, 1, 1]

> ToR(g &* f);

                      2                          2
         - 2 (- 1 + q ) R[1, 1, 4] - 2 (- 1 + q ) R[1, 1, 3, 1]

                    2                             2
       - 2 (- 1 + q ) R[1, 1, 1, 3] - 2 (- 1 + q ) R[1, 1, 1, 2, 1]

> ToR(Internal((1-q)/(1+q)*L[2,1], S[3]) + Ps[1,1,1]);
```

```
                          R[1, 2]     R[1, 1, 1]
      R[3] + R[2, 1] + 2 ------- + 2 ----------
                           1 + q        1 + q
```

SEE ALSO: ToL, ToPh, ToPs, ToS

5.14.65 ToS — express any noncommutative symmetric function in the S-basis

Calling Sequence:

ToS(*ncsf*)

NCSF[ToS](*ncsf*)

Parameters:

ncsf — any noncommutative symmetric function

Synopsis:

- The ToS function computes the expansion of *ncsf* in the S-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function *ncsf* is expanded and the result is not collected.
- One may specify by a second argument, say b, that *ncsf* is solely expressed in terms of the known basis b.
- The call ToS(*ncsf*, 'S') does not affect the argument *ncsf*.
- One may add 'noexpand' just after the argument *ncsf* to choose not to expand the noncommutative symmetric function *ncsf* before treating it.
- One may collect the result by adding a third argument: ToS(*ncsf*, b, 'collect'). For instance, ToS(*ncsf*, 'S', 'collect') may be used to collect the argument *ncsf*.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToS and another name used in the same session, use the long form NCSF['ToS'].

Examples:

```
> with(NCSF):
> f:=(1-q^2)*Ps[2,1]:
> g:=2*R[2,1]:
> ToS(f &* g);

                   2                          2
         4 (- 1 + q ) S[2, 1, 3] - 4 (- 1 + q ) S[2, 1, 2, 1]

                  2                             2
      - 2 (- 1 + q ) S[1, 1, 1, 3] + 2 (- 1 + q ) S[1, 1, 1, 2, 1]

> ToS(g &* f);
```

```
                 2                          2
     4 (- 1 + q ) S[3, 2, 1] - 2 (- 1 + q ) S[3, 1, 1, 1]

                2                            2
   - 4 (- 1 + q ) S[2, 1, 2, 1] + 2 (- 1 + q ) S[2, 1, 1, 1, 1]
```

```
> ToS( Internal((1-q^3)/(1+q)*L[2,1], f) );
```

```
                   4    3                 4    3
  - 2 S[2, 1] (q  - q  - q + 1) + (q  - q  - q + 1) S[1, 1, 1]
```

SEE ALSO: ToL, ToPh, ToPs, ToR

5.15 Using `TAB` to manipulate tableaux

Calling Sequence:

`TAB[<function>]`(*args*)

`<function>`(*args*)

Synopsis:

- This package provides functions to work on tableaux and contretableaux.
- To get more information on tableaux do `?TYP,Tableau` or `?TYP[Tableau]`.
- To get more information on contretableaux do `?TYP,CTableau` or `?TYP[CTableau]`.
- Whenever there is a conflict between a function name in `TAB` and another name used in the same session, use the long form `TAB[<function>]`.
- The available functions are:

Bump	BumpC	CTab2Mat	CTab2Part
CTab2Word	InvSchensted	ListStdTab	ListTab
ListYama	Mat2CTab	Mat2Tab	Part2ListStdTab
Schensted	PermOnCTab	PermOnTab	PermOnWord
StdTab2Idemp	Tab2Mat	Tab2Part	Tab2Word
Word2Charge	Word2CoCharge	Word2CTab	Word2LeftKey
Word2RightKey	Word2StdWord	Word2Tab	Yama2StdTab

- For help with a particular function do `?TAB[<function>]` or `?TAB,<function>` where `<function>` is one from the above list.

Examples:

```
> with(TAB):
> Word2CTab([5,2,4,1,3,3,2,5,4,5,1,7,4,3,1,5,1,1]);

      [[2], [3, 1], [5, 4, 1], [5, 4, 1], [5, 4, 3, 1], [7, 5, 3, 2, 1]]

> CTab2Mat(");

                              [ 2  3  5  5  5  7 ]
                              [                  ]
                              [    1  4  4  4  5 ]
                              [                  ]
                              [       1  1  3  3 ]
                              [                  ]
                              [             1  2 ]
                              [                  ]
                              [                1 ]
```

5.15.1 `Bump` — insert a word into a tableau

Calling Sequence:

`Bump`(t, w)

`TAB[Bump]`(t, w)

Parameters:

t — any list of list(s) denoting a tableau

w — any list of positive integers denoting a word

Synopsis:

- The `Bump` function inserts a word $[x_1, x_2, \ldots]$ into a tableau from the right. It first inserts x_1, then x_2, and so on.
- The insertion algorithm is due to Schensted.
- Whenever there is a conflict between the function name `Bump` and another name used in the same session, use the long form `TAB['Bump']`.

Examples:

```
> with(TAB):
> t:=[[3], [2,6,8]]:
> Tab2Mat( Bump(t, [4,1]) );
```

```
                    [ 3       ]
                    [         ]
                    [ 2  6    ]
                    [         ]
                    [ 1  4  8 ]
```

SEE ALSO: `TYP[IsWord]`, `Mat2Tab`, `TYP[Tableau]`, `Tab2Mat`, `Tab2Word`, `Word2Tab`

5.15.2 `BumpC` — insert a word into a contretableau

Calling Sequence:

`BumpC`(ct, w)

`TAB[BumpC]`(ct, w)

Parameters:

ct — any list of list(s) denoting a contretableau

w — any list of positive integers denoting a word

Synopsis:

- The `BumpC` function inserts a word $[x_1, x_2, \ldots]$ into a contretableau from the right. It first inserts x_1, then x_2, and so on.
- The insertion algorithm is due to Schensted.
- Whenever there is a conflict between the function name `BumpC` and another name used in the same session, use the long form `TAB['BumpC']`.

Examples:

```
> with(TAB):
> ct:=[[4,2], [7,5,3]]:
> CTab2Mat( BumpC(ct, [6,4]) );
```

```
                    [ 4  5  7 ]
                    [         ]
                    [ 2  3  6 ]
                    [         ]
                    [       4 ]
```

SEE ALSO: `CTab2Mat`, `CTab2Word`, `TYP[IsCTab]`, `Mat2CTab`, `Word2CTab`

5.15.3 CTab2Mat — convert a contretableau into a matrix

Calling Sequence:

`CTab2Mat`(ct)

`TAB[CTab2Mat]`(ct)

Parameters:

ct — any list of list(s) denoting a contretableau

Synopsis:

- The `CTab2Mat` function converts a contretableau ct (read by columns) into a matrix which is its 2-dimensionnal representation.

- Whenever there is a conflict between the function name `CTab2Mat` and another name used in the same session, use the long form `TAB['CTab2Mat']`.

Examples:

```
> with(TAB):
> CTab2Mat([[5], [5,4], [6,4,2,1]]);
```

```
                    [ 5  5  6 ]
                    [         ]
                    [    4  4 ]
                    [         ]
                    [       2 ]
                    [         ]
                    [       1 ]
```

SEE ALSO: `Mat2CTab`, `TYP[CTableau]`

5.15.4 `CTab2Part` — compute the shape of a contretableau

Calling Sequence:

`CTab2Part`(*ct*)

`TAB[CTab2Part]`(*ct*)

Parameters:

ct — any list of list(s) denoting a contretableau

Synopsis:

- The `CTab2Part` function computes the shape of a contretableau *ct*. This is the partition whose parts are the lengths of the columns of the contretableau reordered in weakly decreasing order.
- Whenever there is a conflict between the function name `CTab2Part` and another name used in the same session, use the long form `TAB['CTab2Part']`.

Examples:

```
> with(TAB):
> ct:=[[1], [2,1], [5,4,2,1], [5,4,3,2,1]]:
> CTab2Part(ct);

                              [5, 4, 2, 1]

> CTab2Mat(ct);

                             [ 1  2  5  5 ]
                             [            ]
                             [    1  4  4 ]
                             [            ]
                             [       2  3 ]
                             [            ]
                             [       1  2 ]
                             [            ]
                             [          1 ]
```

See Also: `TYP[CTableau]`, `Tab2Part`

5.15.5 `CTab2Word` — convert a contretableau into a word

Calling Sequence:

`CTab2Word`(*ct*)

`TAB[CTab2Word]`(*ct*)

Parameters:

ct — any list of list(s) denoting a contretableau

Synopsis:

- The `CTab2Word` function converts a contretableau into a word. This is done by reading, from left to right, the successive columns of the contretableau (read downwards).

- Whenever there is a conflict between the function name `CTab2Word` and another name used in the same session, use the long form `TAB['CTab2Word']`.

Examples:

```
> with(TAB):
> ct:=[[2,1], [4,3], [7,6,5]]:
> CTab2Word(ct);

                    [2, 1, 4, 3, 7, 6, 5]

> CTab2Mat(ct);

                          [ 2  4  7 ]
                          [         ]
                          [ 1  3  6 ]
                          [         ]
                          [       5 ]
```

SEE ALSO: `CTab2Mat`

5.15.6 `InvSchensted` — inverse Schensted correspondence

Calling Sequence:

`InvSchensted`(*pair*)

`InvSchensted`(*pair*,'`left`')

`TAB[InvSchensted]`(*pair*)

`TAB[InvSchensted]`(*pair*,'`left`')

Parameters:

pair — any list of one tableau and one standard tableau

Synopsis:

- The `InvSchensted` function gives the bijection between pairs of tableaux and words. The argument *pair* is a list of two tableaux of the same shape, the second one being standard and called the (right) insertion tableau. The result is the word corresponding to this pair of tableaux.
- When called with a second argument `'left'`, the standard tableau is the left insertion tableau.
- Whenever there is a conflict between the function name `InvSchensted` and another name used in the same session, use the long form `TAB['InvSchensted']`.

Examples:

```
> with(TAB):
> Schensted([1,4,2,4,2]);

          [[[4, 4], [1, 2, 2]], [[3, 5], [1, 2, 4]]]

> InvSchensted(");
```

```
                              [1, 4, 2, 4, 2]

> InvSchensted("", 'left');

                              [4, 1, 4, 2, 2]
```

SEE ALSO: `Schensted`

5.15.7 `ListStdTab` — generate all standard tableaux

✓

Calling Sequence:

`ListStdTab`(n)

`TAB[ListStdTab]`(n)

Parameters:

n — any positive integer

Synopsis:

- The `ListStdTab` function generates all standard tableaux on n letters.
- When called with a second parameter, say `'nb'`, it returns the number of standard tableaux on n letters.
- Whenever there is a conflict between the function name `ListStdTab` and another name used in the same session, use the long form `TAB['ListStdTab']`.

Examples:

```
> with(TAB):
> ListStdTab(3);

        [[[3], [2], [1]], [[2], [1, 3]], [[3], [1, 2]], [[1, 2, 3]]]

> map(Tab2Mat, ListStdTab(3));

                                                [ 3 ]
                             [ 2    ]  [ 3    ]  [   ]
               [[ 1  2  3 ], [      ], [      ], [ 2 ]]
                             [ 1  3 ]  [ 1  2 ]  [   ]
                                                [ 1 ]

> ListStdTab(15, 'nb');

                                  10349536
```

SEE ALSO: `ListTab, Part2ListStdTab, TYP[Tableau]`

5.15.8 ListTab — generate all tableaux of a given shape and evaluation

✘

Calling Sequence:

`ListTab`(*part*, *eval*)

`TAB[ListStdTab]`(*part*, *eval*)

Parameters:

part — any list denoting a partition

eval — any list denoting an evaluation

Synopsis:

- The `ListTab` function generates all tableaux of a given shape *part* and of a given weight *eval*. Note that this vector *eval* possibly has some 0.
- When called with a third parameter, say `'nb'`, it returns the number of such tableaux.
- The generator is more efficient when *eval* is sorted as an increasing sequence of integers. It can be useful if you solely want the number of tableaux because sorting the evaluation does not change this number.
- Whenever there is a conflict between the function name `ListTab` and another name used in the same session, use the long form `TAB['ListTab']`.

Examples:

```
> with(TAB):
> ListTab([3,2], [0,1,0,0,1,2,0,0,1]);

      [[[6, 9], [2, 5, 6]], [[5, 9], [2, 6, 6]], [[6, 6], [2, 5, 9]]]

> map(Tab2Mat, ");

                    [ 6  9    ]  [ 5  9    ]  [ 6  6    ]
                   [[         ], [         ], [         ]]
                    [ 2  5  6 ]  [ 2  6  6 ]  [ 2  5  9 ]

> ListTab([5,5], [1$10], 'nb');

                                   42
```

SEE ALSO: `ListStdTab`

5.15.9 Mat2CTab — convert a matrix into a contretableau

Calling Sequence:

`Mat2CTab`(m)

`TAB[Mat2CTab]`(m)

Parameters:

m — any matrix

Synopsis:

- The `Mat2CTab` function converts a matrix m into a contretableau. It does not check that the result is really a well-formed contretableau.
- Whenever there is a conflict between the function name `Mat2CTab` and another name used in the same session, use the long form `TAB['Mat2CTab']`.

Examples:

```
> with(TAB):
> CTab2Mat([[2,1], [4,3], [7,6,5]]);

                        [ 2  4  7 ]
                        [         ]
                        [ 1  3  6 ]
                        [         ]
                        [       5 ]

> Mat2CTab(");

                  [[2, 1], [4, 3], [7, 6, 5]]
```

SEE ALSO: `CTab2Mat`, `TYP[CTableau]`, `TYP[IsCTab]`

5.15.10 `Mat2Tab` — convert a matrix into a tableau

Calling Sequence:

`Mat2Tab`(m)

`TAB[Mat2Tab]`(m)

Parameters:

m — any matrix

Synopsis:

- The `Mat2Tab` function converts a matrix m into a tableau. It does not check that the result is really a well-formed tableau.
- Whenever there is a conflict between the function name `Mat2Tab` and another name used in the same session, use the long form `TAB['Mat2Tab']`.

Examples:

```
> with(TAB):
> Tab2Mat([[6], [4,5], [1,2,3]]);

                        [ 6       ]
                        [         ]
                        [ 4  5    ]
                        [         ]
                        [ 1  2  3 ]
```

```
> Mat2Tab(");

                    [[6], [4, 5], [1, 2, 3]]
```

SEE ALSO: TYP[IsTab], Tab2Mat, TYP[Tableau]

5.15.11 Part2ListStdTab — generate all standard tableaux of a given shape

Calling Sequence:

Part2ListStdTab(*part*)

TAB[Part2ListStdTab](*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The Part2ListStdTab function generates all standard tableaux of the given shape *part*. This is done by generating all Yamanouchi words of a given weight.
- Whenever there is a conflict between the function name Part2ListStdTab and another name used in the same session, use the long form TAB['Part2ListStdTab'].

Examples:

```
> with(TAB):
> Part2ListStdTab([3,1]);

          [[[2], [1, 3, 4]], [[3], [1, 2, 4]], [[4], [1, 2, 3]]]

> map(Tab2Mat, ");

                [ 2       ]  [ 3       ]  [ 4       ]
               [[         ], [         ], [         ]]
                [ 1  3  4 ]  [ 1  2  4 ]  [ 1  2  3 ]
```

SEE ALSO: TYP[IsYama], ListStdTab, ListYama, Tab2Mat, TYP[Tableau]

5.15.12 PermOnCTab — action of a permutation on a product of columns

Calling Sequence:

PermOnCTab(*perm*, *ctab*)

TAB[PermOnCTab](*perm*, *ctab*)

Parameters:

perm — any list denoting a permutation

ctab — any list of lists denoting a product of columns

Synopsis:

- The `PermOnCTab` function plaxically permutes columns in a product of columns *ctab* (not necessarily a contre-tableau, but obtained from a contretableau by permutation).
- A simple plaxic column-transposition transforms a two-column contre-tableau into the two-column tableau which is obtained from it by Schensted algorithm, or conversely. This transformation is also given by the jeu de taquin.
- Whenever there is a conflict between the function name `PermOnCTab` and another name used in the same session, use the long form `TAB['PermOnCTab']`.

Examples:

```
> with(TAB):
> PermOnCTab([2,1], [[5], [6,2], [7,4,3,1]]);

                        [[5, 2], [6], [7, 4, 3, 1]]
```

SEE ALSO: `PermOnTab`

5.15.13 `PermOnTab` — action of a permutation on a product of rows

Calling Sequence:

`PermOnTab`$(perm, tab)$

`TAB[PermOnTab]`$(perm, tab)$

Parameters:

perm — any list denoting a permutation

tab — any list of lists denoting a product of rows

Synopsis:

- The `PermOnTab` function plaxically permutes rows in a product of rows, say *tab* (not necessarily a tableau, but the image of a tableau under a row-permutation).
- A simple plaxic row-transposition transforms a two-row contre-tableau into the two-row tableau which is obtained from it by Schensted algorithm, or conversely. This transformation is also given by the jeu de taquin.
- For example, the two following products of rows are exchanged:

```
1 3 4 6              3 6
2 5              1 2 4 5
```

- Whenever there is a conflict between the function name `PermOnTab` and another name used in the same session, use the long form `TAB['PermOnTab']`.

Examples:

```
> with(TAB):
> PermOnTab([2,1], [ [1,3,4,6], [2,5]]);

                    [[3, 6], [1, 2, 4, 5]]

> Tab2Mat(");

                         [ 3  6       ]
                         [            ]
                         [ 1  2  4  5 ]
```

SEE ALSO: `PermOnCTab`

5.15.14 `PermOnWord` — realize the action of a permutation on a word

Calling Sequence:

`PermOnWord`$(perm, w)$

`TAB[PermOnWord]`$(perm, w)$

Parameters:

$perm$ — any list denoting a permutation

w — any list of positive integers denoting a word

Synopsis:

- The `PermOnWord` function realizes the action of a permutation on a word. This action is compatible with plactic congruences and Schensted algorithm. The simple transposition s_i exchanges in a non-trivial manner the number of i and $i+1$ in the word.
- The image of a tableau (resp. contretableau) is a tableau (resp. contretableau).
- Whenever there is a conflict between the function name `PermOnWord` and another name used in the same session, use the long form `TAB['PermOnWord']`.

Examples:

```
> with(TAB):
> t:=[[2,2,3], [1,1,2,2,3]]:
> Tab2Word("):
> PermOnWord([1,3,2], "):
> Word2Tab(");

                  [[2, 3, 3], [1, 1, 2, 3, 3]]
```

SEE ALSO: `TYP[IsPerm]`, `TYP[IsWord]`, `Word2Tab`

5.15.15 `StdTab2Idemp` — build an idempotent from a standard tableau

Calling Sequence:

`StdTab2Idemp`(t)

`TAB[StdTab2Idemp]`(t)

Parameters:

t — any list of list(s) denoting a standard tableau

Synopsis:

- The `StdTab2Idemp` function computes an idempotent from a standard tableau. These idempotents (for all standard tableaux) are mutually orthogonal.

- Let t' be the tableau obtained from t by deleting the last letter n and let $t_1, t_2, \ldots$, be the other tableaux obtained from t' by adding n. Then, up to a normalization factor: `StdTab2Idemp`(t) = `StdTab2Idemp`(t') $\times \prod($`Jucis`$(n,n) - c(n,t_i))$ where `Jucis`(n,n) is the last Jucis-Murphy element and $c(n,t_i)$ the distance to the diagonal of the letter n in the tableau t_i.

- Whenever there is a conflict between the function name `StdTab2Idemp` and another name used in the same session, use the long form `TAB['StdTab2Idemp']`.

Examples:

```
> with(TAB):
> r1:=StdTab2Idemp([[3], [1,2]]):
> r2:=StdTab2Idemp([[3], [1,2,4]]);

       r2 := 5/48 [4, 2, 3, 1] + 5/48 [1, 4, 3, 2]  + 1/24 [1, 2, 4, 3]

            + 1/8 [1, 2, 3, 4] + 5/48 [4, 1, 3, 2]  - 1/48 [4, 2, 1, 3]

            + 5/48 [2, 4, 3, 1] - 1/48 [1, 4, 2, 3] - 1/48 [3, 2, 4, 1]

            - 1/48 [1, 3, 4, 2] - 1/16 [3, 2, 1, 4] - 1/12 [4, 3, 1, 2]

            - 1/48 [2, 4, 1, 3] - 1/12 [3, 4, 2, 1] - 1/48 [3, 1, 4, 2]

            - 1/16 [1, 3, 2, 4] - 1/48 [4, 1, 2, 3] - 1/48 [2, 3, 4, 1]

            + 1/24 [2, 1, 4, 3] + 1/8 [2, 1, 3, 4]  - 1/16 [3, 1, 2, 4]

            - 1/12 [4, 3, 2, 1] - 1/12 [3, 4, 1, 2] - 1/16 [2, 3, 1, 4]

> (r1 &!* (Jucis(4,4) &!* (Jucis(4,4) + 2) / 8)) &!- r2;

                                       0
```

SEE ALSO: `SGA[Jucis]`

5.15.16 `Schensted` — Schensted correspondence

✘

Calling Sequence:

`Schensted`(w)

`TAB[Schensted]`(w)

Parameters:

w — any list of positive integers denoting a word

Synopsis:

- The `Schensted` function gives the bijection between words and pairs of tableaux. The result is a list of two tableaux of the same shape, the second one being standard and called the (right) insertion tableau.
- Whenever there is a conflict between the function name `Schensted` and another name used in the same session, use the long form `TAB['Schensted']`.

✂ ....

Examples:

```
> with(TAB):
> Schensted([4,1,2,4,2]);

                 [[[4, 4], [1, 2, 2]], [[2, 5], [1, 3, 4]]]

> map(Tab2Mat, ");

                           [ 4  4    ]  [ 2  5    ]
                          [[         ], [         ]]
                           [ 1  2  2 ]  [ 1  3  4 ]
```

.... ✂

SEE ALSO: `InvSchensted`

5.15.17 `Tab2Mat` — convert a tableau into a matrix

Calling Sequence:

`Tab2Mat`(t)

`TAB[Tab2Mat]`(t)

Parameters:

t — any list of list(s) denoting a tableau

Synopsis:

- The `Tab2Mat` function converts a tableau t (read by rows) into a matrix which is its 2-dimensionnal representation.
- Whenever there is a conflict between the function name `Tab2Mat` and another name used in the same session, use the long form `TAB['Tab2Mat']`.

Examples:

```
> with(TAB):
> Tab2Mat([[5], [4,5], [1,2,4,6]]);

                              [ 5          ]
                              [            ]
                              [ 4  5       ]
                              [            ]
                              [ 1  2  4  6 ]
```

SEE ALSO: `Mat2Tab, TYP[Tableau]`

5.15.18 `Tab2Part` — compute the shape of a tableau

Calling Sequence:

`Tab2Part`(t)

`TAB[Tab2Part]`(t)

Parameters:

t — any list of list(s) denoting a tableau

Synopsis:

- The `Tab2Part` function computes the shape of a tableau t. This is the partition whose parts are the lengths of the rows of the tableau reordered in weakly decreasing order.
- Whenever there is a conflict between the function name `Tab2Part` and another name used in the same session, use the long form `TAB['Tab2Part']`.

Examples:

```
> with(TAB):
> t:=[[3], [2,3], [1,2,5,6]]:
> Tab2Part(t);

                                [4, 2, 1]

> Tab2Mat(t);

                              [ 3          ]
                              [            ]
                              [ 2  3       ]
                              [            ]
                              [ 1  2  5  6 ]
```

SEE ALSO: `CTab2Part, TYP[CTableau], TYP[Tableau]`

5.15.19 `Tab2Word` — convert a tableau into a word

Calling Sequence:

`Tab2Word`(t)

`TAB[Tab2Word]`(t)

Parameters:

t — any list of list(s) denoting a tableau

Synopsis:

- The `Tab2Word` function converts a tableau into a word. This is done by reading, from top to bottom, the successive rows of the tableau (read from left to right).
- Whenever there is a conflict between the function name `Tab2Word` and another name used in the same session, use the long form `TAB['Tab2Word']`.

Examples:

```
> with(TAB):
> t:=[[7], [2,4,6], [1,3,5]]:
> Tab2Word(t);

                          [7, 2, 4, 6, 1, 3, 5]

> Tab2Mat(t);

                              [ 7       ]
                              [         ]
                              [ 2  4  6 ]
                              [         ]
                              [ 1  3  5 ]
```

See Also: `Tab2Mat, Word2Tab`

5.15.20 `Word2StdWord` — standardize a word

Calling Sequence:

`Word2StdWord`(w)

`TAB[Word2StdWord]`(w)

Parameters:

w — any list of positive integers denoting a word

Synopsis:

- The `Word2StdWord` function standardizes a word, *i.e.* replaces the 1's by $1, \ldots, h$, the 2's by $h+1, \ldots, k$, etc, from left to right.
- Whenever there is a conflict between the function name `Word2StdWord` and another name used in the same session, use the long form `TAB['Word2StdWord']`.

Examples:

```
> with(TAB):
> Word2StdWord([3,4,3,1,2,1,1]);

                                [5, 7, 6, 1, 4, 2, 3]
```

SEE ALSO:

5.15.21 Word2Tab — convert a word into a tableau

Calling Sequence:

Word2Tab(w)

TAB[Word2Tab](w)

Parameters:

w — any list of positive integers denoting a word

Synopsis:

- The Word2Tab function inserts a word $[x_1, x_2, \ldots]$ into an empty tableau from the right. It first inserts x_1, then x_2, and so on.
- The insertion algorithm is due to Schensted.
- Whenever there is a conflict between the function name Word2Tab and another name used in the same session, use the long form TAB['Word2Tab'].

Examples:

```
> with(TAB):
> Tab2Mat( Word2Tab([5,6,2,4,3,7,1]) );

                                     [ 5       ]
                                     [         ]
                                     [ 4       ]
                                     [         ]
                                     [ 2  6    ]
                                     [         ]
                                     [ 1  3  7 ]
```

SEE ALSO: Bump, Tab2Mat, Tab2Word

5.15.22 Word2CoCharge — returns the cocharge of a word

Calling Sequence:

Word2CoCharge(w)

TAB[Word2CoCharge](w)

Parameters:

w — any list of positive integers denoting any word

Synopsis:

- The `Word2CoCharge` function computes the cocharge of the word w.
- When called with the second argument `'vector'`, it returns the cocharge vector, the sum of the components of this vector being the cocharge. This option is defined only when the word contains at least as many 1's as 2's, as many 2's as 3's and so on.
- Whenever there is a conflict between the function name `Word2CoCharge` and another name used in the same session, use the long form `TAB['Word2CoCharge']`.

Examples:

```
> with(TAB):
> Word2CoCharge([3,2,2,1,4,1,1,3], 'vector');

                        [2, 1, 1, 0, 2, 0, 0, 1]

> Word2CoCharge([3,3,2,2,4,1,3,3,1,3]);

                                   8
```

SEE ALSO: `Word2Charge`

5.15.23 `Word2CTab` — convert a word into a contretableau

Calling Sequence:

`Word2CTab`(w)

`TAB[Word2CTab]`(w)

Parameters:

w — any list of positive integers denoting a word

Synopsis:

- The `Word2Tab` function inserts a word $[x_1, x_2, \ldots]$ into an empty contretableau from the right. It first inserts x_1, then x_2, and so on.
- The insertion algorithm is due to Schensted.
- Whenever there is a conflict between the function name `Word2CTab` and another name used in the same session, use the long form `TAB['Word2CTab']`.

Examples:

```
> with(TAB):
> CTab2Mat( Word2CTab([5,6,2,4,3,7,1]) );

                              [ 5  6  7 ]
                              [         ]
                              [    2  4 ]
                              [         ]
                              [       3 ]
                              [         ]
                              [       1 ]
```

SEE ALSO: `BumpC`, `CTab2Mat`, `CTab2Word`

5.15.24 `Word2Charge` — returns the charge of a word

✓

Calling Sequence:

`Word2Charge`(w)

`TAB[Word2Charge]`(w)

Parameters:

w — any list of positive integers denoting any word

Synopsis:

- The `Word2Charge` function computes the charge of the word w.
- When called with the second argument `'vector'`, it returns the charge vector, the sum of the components of this vector being the charge. This option is defined only when the word contains at least as many 1's as 2's, as many 2's as 3's and so on.
- Whenever there is a conflict between the function name `Word2Charge` and another name used in the same session, use the long form `TAB['Word2Charge']`.

Examples:

```
> with(TAB):
> Word2Charge([3,2,2,1,4,1,1,3], 'vector');

                    [0, 0, 0, 0, 1, 0, 0, 1]

> Word2Charge([3,3,2,2,4,1,3,3,1,3]);

                               1
```

SEE ALSO: `Word2CoCharge`

5.15.25 `Word2LeftKey` — left key

✘

Calling Sequence:

`Word2LeftKey`(w)

`TAB[Word2LeftKey]`(w)

Parameters:

w — any list of positive integers denoting a word

Synopsis:

- The `Word2LefttKey` function returns the left key of a given word w, *i.e.* the set of left columns in the orbit of `PermOnCTab`($perm$, `Word2CTab`(w)).
- Whenever there is a conflict between the function name `Word2LeftKey` and another name used in the same session, use the long form `TAB['Word2LeftKey']`.

Examples:

```
> with(TAB):
> Word2LeftKey([5,6,2,4,3,7,1]);

                    [[5], [5, 2], [5, 4, 2, 1]]
```

SEE ALSO: PermOnCTab, PermOnTab, Word2RightKey

5.15.26 `Word2RightKey` — right key

✘

Calling Sequence:

`Word2RightKey`(w)

`TAB[Word2RightKey]`(w)

Parameters:

w — any list of positive integers denoting a word

Synopsis:

- The `Word2RightKey` function returns the right key of a given word w, *i.e.* the set of right columns in the orbit of `PermOnCTab`($perm$, `Word2CTab`(w)).
- Whenever there is a conflict between the function name `Word2RightKey` and another name used in the same session, use the long form `TAB['Word2RightKey']`.

Examples:

```
> with(TAB):
> Word2RightKey([5,6,2,4,3,7,1]);

                    [[7, 4, 3, 1], [7, 3], [7]]
```

SEE ALSO: PermOnCTab, PermOnTab, Word2LeftKey

5.15.27 `ListYama` — generate all Yamanouchi words corresponding to a given partition

Calling Sequence:

`ListYama`($part$)

`TAB[ListYama]`($part$)

Parameters:

$part$ — any list denoting a partition

Synopsis:

- The `ListYama` function computes all Yamanouchi words corresponding to a given partition $part$. These words are exactly the words which are plactically congruent to the word:

$$\ldots 3^{cpart[3]}\, 2^{cpart[2]}\, 1^{cpart[1]} \quad ,$$

  or, equivalently, give the tableaux $[\ldots,[3\ldots3],[2\ldots2],[1\ldots1]]$ by Schensted algorithm. $cpart$ denotes the conjugate partition of $part$.

- Whenever there is a conflict between the function name `ListYama` and another name used in the same session, use the long form `TAB['ListYama']`.

Examples:

```
> with(TAB):
> ListYama([2,1]);

                    [[1, 2, 1], [2, 1, 1]]

> ListYama([3,2]);

   [[2, 1, 3, 2, 1], [2, 3, 1, 2, 1], [3, 2, 1, 2, 1], [3, 2, 2, 1, 1],

       [2, 3, 2, 1, 1]]
```

SEE ALSO: `TYP[IsYama]`

5.15.28 `Yama2StdTab` — convert a Yamanouchi word into a standard tableau

Calling Sequence:

`Yama2StdTab`(*yama*)

`TAB[Yama2StdTab]`(*yama*)

Parameters:

yama — any list denoting a Yamanouchi word

Synopsis:

- The `Yama2StdTab` function converts any Yamanouchi word into a standard tableau.
- The letters in the Yamanouchi word indicate the heights of the letters in the standard tableau.
- Whenever there is a conflict between the function name `Yama2StdTab` and another name used in the same session, use the long form `TAB['Yama2StdTab']`.

Examples:

```
> with(TAB):
> Yama2StdTab([1,2,3,1,2,1]);

                    [[4], [2, 5], [1, 3, 6]]

> Tab2Mat(");
                        [ 4       ]
                        [         ]
                        [ 2  5    ]
                        [         ]
                        [ 1  3  6 ]
```

SEE ALSO: `TYP[IsYama]`, `Tab2Mat`, `TYP[Tableau]`

5.16 Manipulating Schubert polynomials with SP

✓

Calling Sequence:

`SP[<function>]`$(args)$

`<function>`$(args)$

Synopsis:

- This package provides functions to work on simple and double Schubert polynomials.
- To get more information on Schubert polynomials do either `?TYP,Schubert` or `?TYP[Schubert]`.
- All defined bases are stored in the global variable `SPBases` that is by default

  `{'x', 'y', 'X', 'Y', 'XX', 'YY'}`.

  Note that the y_i's are seen as special variables since they are the second alphabet for double Schubert polynomials.
- Whenever there is a conflict between a function name in `SP` and another name used in the same session, use the long form `SP[<function>]`.
- The available functions are:

Flag	LRrbyX	MaxTr	Monk
NewtonInterp	Rt	ScalarPol	Specialize
TableX	TableXX	TableY	TableYY
Tox	X2Dim	X2Dimq	ToX
ToXX	ToY	X2Y	X2x
XX2YY	XX2x	Y2X	Y2x
YY2XX	YY2x	x2X	x2XX

- For help with a particular function do `?SP[<function>]` or `?SP,<function>` where `<function>` is one from the above list.

Examples:

```
> with(SP):
> TableXX(3);

             table([
                 [2, 1, 3] = x1 - y1
                 [3, 2, 1] = (x2 - y1) (x1 - y2) (x1 - y1)
                 [1, 3, 2] = x2 - y2 + x1 - y1
                 [2, 3, 1] = (x2 - y1) (x1 - y1)
                 [1, 2, 3] = 1
                 [3, 1, 2] = (x1 - y2) (x1 - y1)
             ])
```

5.16.1 X2x — from X Schubert basis to the basis of monomials

Calling Sequence:

X2x(*schub*)

SP[X2x](*schub*)

Parameters:

schub — any expression in **X** Schubert basis

Synopsis:

- The X2x function converts an expression from **X** Schubert basis to the basis of monomials.
- The result is not expanded.
- Whenever there is a conflict between the function name X2x and another name used in the same session, use the long form SP['X2x'].

Examples:

```
> with(SP):
> X2x(X[4,1,3,2]);

                                    3        3
                              x3 x1  + x2 x1

> X2x(2*X[4,1,2,3] + 4*q^2*X[3,1,2]);

                                  3      2   2
                              2 x1  + 4 q  x1
```

SEE ALSO: x2X

5.16.2 Y2x — from Y Schubert basis to the basis of monomials

Calling Sequence:

Y2x(*schub*)

SP[Y2x](*schub*)

Parameters:

schub — any expression in **Y** Schubert basis

Synopsis:

- The Y2x function converts an expression from **Y** Schubert basis to the basis of monomials.
- The result is not expanded.
- Whenever there is a conflict between the function name Y2x and another name used in the same session, use the long form SP['Y2x'].

Examples:

```
> with(SP):
> Y2x(Y[3,0,1]);

                                          3
                              (x2 + x3) x1

> Y2x(2*Y[3] + 4*q^2*Y[2]);

                                 3       2   2
                             2 x1  + 4 q   x1
```

SEE ALSO: x2Y

5.16.3 Flag — fix the degree of the symmetric group

Calling Sequence:

Flag(n)

SP[Flag](n)

Parameters:

n — any (extra) positive integer

Synopsis:

- To compute in the ring of polynomials modulo the ideal of generated by symmetric polynomials in $x_1, x_2, \ldots, x_n$, use Flag(n). This ring can be interpreted as the cohomology ring of the flag manifold of rank n over the field of complex numbers.
- When called without any argument, it restores the default mode that is to compute without truncation. In that case, the function returns -1 and otherwise is returns n.
- Whenever there is a conflict between the function name Flag and another name used in the same session, use the long form SP['Flag'].

Examples:

```
> with(SP):
> x2X(x3);
                        - X[1, 3, 2] + X[1, 2, 4, 3]

> Flag(3):
> x2X(x3);

                                - X[1, 3, 2]
```

SEE ALSO: x2X

5.16.4 `LRrbyX` — multiply Schur functions

Calling Sequence:

`LRrbyX`($schur_1, schur_2$)

`SP[LRrbyX]`($schur_1, schur_2$)

Parameters:

$schur_1, schur_2$ — any Schur functions

Synopsis:

- The `LRrbyX` function expresses the product of Schur functions $schur_1 \times schur_2$ in the basis of Schur functions.
- Instead of the classical Littlewood-Richardson rule, it makes use of the Lascoux-Schützenberger rule involving Schubert polynomials.
- It can multiply any number of Schur functions given as a sequence of Schur functions indexed by partitions without 0 part.
- Whenever there is a conflict between the function name `LRrbyX` and another name used in the same session, use the long form `SP['LRrbyX']`.

Examples:

```
> with(SP):
> LRrbyX(s[2,1], s[1], s[2]);

    s[2, 2, 1, 1] + s[3, 1, 1, 1] + s[2, 2, 2] + 3 s[3, 2, 1] + s[3, 3]

         + 2 s[4, 1, 1] + 2 s[4, 2] + s[5, 1]
```

SEE ALSO: `Rt`

5.16.5 `MaxTr` — compute the maximal transition for a given permutation

Calling Sequence:

`MaxTr`($perm$)

`SP[MaxTr]`($perm$)

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The `MaxTr` function computes the maximal transition for the Schubert polynomial $\mathbf{X}_{perm}$, i.e. gives a canonical decomposition of $\mathbf{X}_{perm}$:

$$\mathbf{X}_{perm} = x_p\,\mathbf{X}_{perm\,s_k} + \sum \mathbf{X}_{\nu}.$$

- The result is a list of four elements: the first one is $perm$, the second one is the integer p, the third one is the product of $perm$ by the simple transposition s_k and the last is the list of the permutations occuring in the sum.

- The transition is displayed via a pretty printer.
- Whenever there is a conflict between the function name `MaxTr` and another name used in the same session, use the long form `SP['MaxTr']`.

Examples:

```
> with(SP):
> MaxTr([3,1,5,2,4]);

    X[3, 1, 5, 2, 4] = x3 X[3, 1, 4, 2] + X[3, 4, 1, 2] + X[4, 1, 3, 2]
```

SEE ALSO: `Rt`

5.16.6 `Monk` — Monk's formulae

Calling Sequence:

`Monk`($k, schub$)

`SP[Monk]`($k, schub$)

Parameters:

k	—	any positive integer
$schub$	—	any (simple or double) Schubert polynomial indexed by a permutation

Synopsis:

- The `Monk` function computes the product of the variable x_k by a (double) Schubert polynomial *schub*, applying Monk formula.
- Whenever there is a conflict between the function name `Monk` and another name used in the same session, use the long form `SP['Monk']`.

Examples:

```
> with(SP):
> Monk(3, X[3,1,4,2]);

        - X[3, 4, 1, 2] - X[4, 1, 3, 2] + X[3, 1, 5, 2, 4]

> Monk(2, XX[2,4,1,5,3]);

  - XX[4, 2, 1, 5, 3] + XX[2, 5, 1, 4, 3] + y4 XX[2, 4, 1, 5, 3]
```

SEE ALSO: `FM['Monkfix']`

5.16.7 `NewtonInterp` — Newton Interpolation formula

Calling Sequence:

`NewtonInterp`(f, n)

`SP[NewtonInterp]`(f, n)

Parameters:

f — any expression in the basis of monomials

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `NewtonInterp` function expresses the function $f(x_1, x_2, \ldots)$ in terms of $\mathbb{X}$ Schubert polynomials. It uses the Newton interpolation formula involving the permutations in the symmetric group of degree n.
- The function `x2XX` uses Monk formula instead of divided differences to decompose (exactly) a polynomial into the $\mathbb{X}$ basis.
- Don't forget it is an interpolation...
- When called with one argument, the second argument is set to the value of the variable `_Sn` that fixes the degree of the symmetric group.
- The result is not expanded.
- Whenever there is a conflict between the function name `NewtonInterp` and another name used in the same session, use the long form `SP['NewtonInterp']`.

Examples:

```
> with(SP):
> NewtonInterp(2*x2 + 4*a^2*x1^2, 4);

           2   2                                     2
 (2 y2 + 4 a  y1 ) XX[1, 2, 3, 4] + 2 XX[1, 3, 2, 4] + 4 a  XX[3, 1, 2, 4]

            2             2
      + (4 a  y2 - 2 + 4 a  y1) XX[2, 1, 3, 4]

> expand( XX2x(") );

                                           2   2
                                2 x2 + 4 a  x1
```

SEE ALSO: `Flag`, `x2XX`

5.16.8 `Rt` — compute the rooted tree of a permutation

Calling Sequence:

`Rt`$(perm)$

`SP[Rt]`$(perm)$

Parameters:

perm — any list denoting a permutation

Synopsis:

- The `Rt` function computes the rooted tree of a permutation which is obtained by iterating the `MaxTr` function. The leaves of the tree are grassmanniann permutations.
- The rooted tree is represented as a list of two elements: the root and the list of the sub-trees of this root. Is is displayed via a pretty printer.
- Whenever there is a conflict between the function name `Rt` and another name used in the same session, use the long form `SP['Rt']`.

Examples:

```
> with(SP):
> Rt([2,6,3,4,5,1]);
[2,6,3,4,5,1]
                            [[2, 6, 3, 4, 5, 1], []]

> Rt([2,1,4,3]);
[2,1,4,3]                   # the root
   [2,3,1,4]                # a sub-tree (a leaf)
   [3,1,2,4]                # a sub-tree (a leaf)

            [[2, 1, 4, 3], [[[2, 3, 1, 4], []], [[3, 1, 2, 4], []]]]

> MaxTr([2,1,4,3]);

   X[2, 1, 4, 3] = x3 X[2, 1, 3, 4] + X[2, 3, 1, 4] + X[3, 1, 2, 4]
```

SEE ALSO: `MaxTr`

5.16.9 `ScalarPol` — scalar product on polynomials

Calling Sequence:

`ScalarPol`(pol_1, pol_2, n)

`SP[ScalarPol]`(pol_1, pol_2, n)

Parameters:

pol_1 — any polynomial expression

pol_2 — any polynomial expression

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `ScalarPol` function computes the scalar product $\langle pol_1, pol_2\rangle$ corresponding to the symmetric group of degree n.
- $\langle pol_1, pol_2\rangle = \langle pol_1 \times pol_2, 1\rangle$.

- $\langle pol, 1\rangle$ is the symmetric function (in the basis of Schur functions) obtained by applying the maximal divided difference of $\mathfrak{S}_n$ to *pol*.
- When called without the third argument, it assumes that n equals `_Sn`.
- Whenever there is a conflict between the function name `ScalarPol` and another name used in the same session, use the long form `SP['ScalarPol']`.

Examples:

```
> with(SP):
> ScalarPol(x1^6*x2, x2^2*x3, 3);

                                s[4, 2, 1]
```

SEE ALSO:

5.16.10 `Specialize` — specialize a set of variables

Calling Sequence:

`Specialize`$(expr, gname, f, args)$

`SP[Specialize]`$(expr, gname, f, args)$

Parameters:

expr	—	any expression
gname	—	any string denoting a generic name of variable
f	—	any procedure
args	—	any (extra) paramaters for f

Synopsis:

- The `Specialize` function is devoted to specialize the set of variables generated by the generic name *gname* so that the variable $gname_i$ is sent onto $f(i, args)$.
- Whenever there is a conflict between the function name `Specialize` and another name used in the same session, use the long form `SP['Specialize']`.

Examples:

```
> with(SP):
> Specialize(x1 + y3 + xx2 + x3^3, 'x', i -> y.i);

                                             3
                        y1 + y3 + xx2 + y3

> degree:=3:               # you should use a map to specialize a table...
> map(Specialize, TableX(degree), 'x', proc(i,d) q^(d-i) end, degree);

                        table([
                                            2
```

```
                    [1, 3, 2] = q + q
                    [1, 2, 3] = 1
                                 4
                    [3, 1, 2] = q
                                 3
                    [2, 3, 1] = q
                                 2
                    [2, 1, 3] = q
                                 5
                    [3, 2, 1] = q
                ])

 # specialization of the yi's as  the permutation [3,1,2] of the  xi's...
> map(Specialize, TableXX(3), 'y', (i,perm) -> x.(perm[i]), [3,1,2]);

               table([
                   [3, 1, 2] = 0
                   [1, 3, 2] = x2 - x3
                   [2, 3, 1] = (x2 - x3) (x1 - x3)
                   [1, 2, 3] = 1
                   [2, 1, 3] = x1 - x3
                   [3, 2, 1] = 0
               ])
```

SEE ALSO: `TableXX`

5.16.11 `TableX` — table of all Schubert polynomials

Calling Sequence:

`TableX`(n)

`SP[TableX]`(n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `TableX` function returns the table of all Schubert polynomials indexed by permutations in $\mathfrak{S}_n$. The polynomials are expressed in the basis of monomials.
- Whenever there is a conflict between the function name `TableX` and another name used in the same session, use the long form `SP['TableX']`.

Examples:

```
> with(SP):
> t:=TableX(3);
                t := table([
                    [1, 2, 3] = 1
                                      2
                    [3, 1, 2] = x1
                    [1, 3, 2] = x2 + x1
                    [2, 1, 3] = x1
```

```
                         [2, 3, 1] = x2 x1
                                          2
                         [3, 2, 1] = x2 x1
                     ])
```

SEE ALSO: `TableXX`

5.16.12 `TableXX` — table of all double Schubert polynomials

Calling Sequence:

`TableXX`(n)

`TableXX`($n, perm$)

`SP[TableXX]`(n)

`SP[TableXX]`($n, perm$)

Parameters:

n — any positive integer denoting the degree of a symmetric group

$perm$ — any list denoting a permutation

Synopsis:

- The `TableXX` function returns the table of all double Schubert polynomials indexed by permutations in $\mathfrak{S}_n$. The polynomials are expressed in the basis of monomials.
- When the function is called with the second argument $perm$, the second alphabet (the y_i's) is specialized as the permutation $perm$ of the first one (the x_i's): in other words, y_i is sent to $x_{perm[i]}$.
- Whenever there is a conflict between the function name `TableXX` and another name used in the same session, use the long form `SP['TableXX']`.

Examples:

```
> with(SP):
> t:=TableXX(3);

             t := table([
                       [1, 2, 3] = 1
                       [3, 1, 2] = (x1 - y2) (x1 - y1)
                       [2, 1, 3] = x1 - y1
                       [1, 3, 2] = x2 - y2 + x1 - y1
                       [2, 3, 1] = (x2 - y1) (x1 - y1)
                       [3, 2, 1] = (x2 - y1) (x1 - y2) (x1 - y1)
                  ])

> t[ [3,1,2] ];
                              (x1 - y2) (x1 - y1)
> TableXX(3, [3,2,1]);

                table([
                     [1, 2, 3] = 1
                     [3, 1, 2] = (x1 - x2) (x1 - x3)
```

```
                [2, 1, 3] = x1 - x3
                [1, 3, 2] = x1 - x3
                [2, 3, 1] = (x2 - x3) (x1 - x3)
                [3, 2, 1] = (x2 - x3) (x1 - x2) (x1 - x3)
            ])
```

SEE ALSO: `TableX`

5.16.13 `TableY` — table of all Schubert polynomials

Calling Sequence:

`TableY`(n)

`SP[TableY]`(n)

Parameters:

n — any positive integer denoting the degree of a symmetric group

Synopsis:

- The `TableY` function returns the table of all Schubert polynomials indexed by codes of permutations in $\mathfrak{S}_n$. The polynomials are expressed in the basis of monomials.
- Whenever there is a conflict between the function name `TableY` and another name used in the same session, use the long form `SP['TableY']`.

Examples:

```
> with(SP):
> TableY(3);
                    table([
                        [0, 0, 0] = 1
                                      2
                        [2, 0, 0] = x1
                        [0, 1, 0] = x1 + x2
                        [1, 0, 0] = x1
                        [1, 1, 0] = x1 x2
                                      2
                        [2, 1, 0] = x1  x2
                    ])
```

SEE ALSO: `TableYY`

5.16.14 `TableYY` — table of all double Schubert polynomials

Calling Sequence:

`TableYY`(n)

`TableYY`($n, perm$)

`SP[TableYY]`(n)

`SP[TableYY]`($n, perm$)

Parameters:

n — any positive integer denoting the degree of a symmetric group

$perm$ — any list denoting a permutation

Synopsis:

- The `TableYY` function returns the table of all double Schubert polynomials indexed by codes of permutations in $\mathfrak{S}_n$. The polynomials are expressed in the basis of monomials.
- When called with the second argument *perm*, the second alphabet (the y_i's) is specialized as the permutation *perm* of the first one (the x_i's) : in other words, y_i is sent to $x_{perm[i]}$.
- Whenever there is a conflict between the function name `TableYY` and another name used in the same session, use the long form `SP['TableYY']`.

Examples:

```
> with(SP):
> t:=TableYY(3);

            t := table([
                    [0, 0, 0] = 1
                    [1, 0, 0] = x1 - y1
                    [2, 0, 0] = (x1 - y2) (x1 - y1)
                    [0, 1, 0] = x2 - y2 + x1 - y1
                    [2, 1, 0] = (x2 - y1) (x1 - y2) (x1 - y1)
                    [1, 1, 0] = (x2 - y1) (x1 - y1)
                ])

> t[ [0,1,0] ];

                              x2 - y2 + x1 - y1

> TableYY(3, [3,2,1]);

             table([
                 [0, 0, 0] = 1
                 [1, 0, 0] = x1 - x3
                 [2, 0, 0] = (x1 - x2) (x1 - x3)
                 [0, 1, 0] = x1 - x3
                 [2, 1, 0] = (x2 - x3) (x1 - x2) (x1 - x3)
                 [1, 1, 0] = (x2 - x3) (x1 - x3)
             ])
```

SEE ALSO: `TableY`

5.16.15 `ToX` — convert any expression to the X Schubert basis

Calling Sequence:

`ToX`$(expr)$

`ToX`$(expr, b)$

`SP[ToX]`$(expr)$

`SP[ToX]`$(expr, b)$

Parameters:

expr — any expression

b — any name of a known basis

Synopsis:

- The `ToX` function converts any expression *expr* to the **X** Schubert basis. *expr* may involve some x_i's, simple Schubert polynomials ($\mathbf{X}_{perm}, \mathbf{Y}_{code}$), double Schubert polynomials ($\mathbb{X}_{perm}, \mathbb{Y}_{code}$, the second alphabet being the y_i's), other terms being considered as coefficients.
- The expression *expr* is expanded and the result is not collected.
- One may specify by a second argument, say *b*, that *expr* is solely expressed in terms of the known basis *b* (`x, X, Y, XX, YY` and even `y` that is seen as a basis in the package).
- The call `ToX`(*expr*, `'X'`) does not affect the argument *expr*.
- One may add `'noexpand'` just after the argument *expr* to choose not to expand the expression *expr* before treating it.
- One may collect the result by adding a third argument: this is done by `ToX`(*expr*, *b*, `'collect'`). For instance, `ToX`(*expr*, `'X', 'collect'`) may be used to collect the argument *expr*.
- Whenever there is a conflict between the function name `ToX` and another name used in the same session, use the long form `SP['ToX']`.

Examples:

```
> with(SP):
> ToX((1+q)^500*x3*x4):                         # expands the input
> ToX((1+q)^500*x3*x4,noexpand):                # does not expand (1+q)^500
> ToX((1+q)^500*x3*x4,collect):                 # collects the result
> ToX((1+q)^500*x3*x4,noexpand,'x'):            # the most efficient
> ToX((1+q)^500*x3*x4,'x',collect):             # specifies a basis
> ToX((1+q)^500*x3*x4,noexpand,'x',collect):
> ToX(q^2*x3*XX[3,1,2]*Y[0,0,1], collect):

     2                        2
  - q  y2 y1 X[1, 4, 2, 3] + q  y2 y1 X[1, 2, 5, 3, 4]

     2                           2
  + q  (y1 + y2) X[4, 1, 2, 3] + q  (y1 + y2) X[2, 4, 1, 3]

     2                  2                  2
  - q  X[4, 2, 1, 3] - q  X[3, 4, 1, 2] - q  (y1 + y2) X[2, 1, 5, 3, 4]

     2
  + q  X[3, 1, 5, 2, 4]
```

SEE ALSO: `TYP[Schubert], Tox`

5.16.16 `ToXX` — convert any expression to the $\mathbb{X}$ Schubert basis

Calling Sequence:

`ToXX`(*expr*)

`ToXX`(*expr*, *b*)

`SP[ToXX]`(*expr*)

`SP[ToXX]`(*expr*, *b*)

Parameters:

expr — any expression

b — any name of a known basis

Synopsis:

- The `ToXX` function converts any expression *expr* to the $\mathbb{X}$ Schubert basis. *expr* may involve some x_i's, simple Schubert polynomials ($\mathbf{X}_{perm}, \mathbf{Y}_{code}$), double Schubert polynomials ($\mathbb{X}_{perm}, \mathbb{Y}_{code}$, the second alphabet being the y_i's), other terms being considered as coefficients.
- The expression *expr* is expanded and the result is not collected.
- One may specify by a second argument, say *b*, that *expr* is solely expressed in terms of the known basis *b* (`x, X, Y, XX, YY` and even `y` that is seen as a basis in the package).
- The call `ToXX`(*expr*, `'XX'`) does not affect the argument *expr*.
- One may add `'noexpand'` just after the argument *expr* to choose not to expand the expression *expr* before treating it.
- One may collect the result by adding a third argument: this is done by `ToXX`(*expr*, *b*, `'collect'`). For instance, `ToXX`(*expr*, `'XX'`, `'collect'`) may be used to collect the argument *expr*.
- Whenever there is a conflict between the function name `ToXX` and another name used in the same session, use the long form `SP['ToXX']`.

Examples:

```
> with(SP):
> ToXX((1+q)^500*x3*x4):                    # expands the input
> ToXX((1+q)^500*x3*x4,noexpand):           # does not expand (1+q)^500
> ToXX((1+q)^500*x3*x4,collect):            # collects the result
> ToXX((1+q)^500*x3*x4,noexpand,'x'):       # the most efficient
> ToXX((1+q)^500*x3*x4,'x',collect):        # specifies a basis
> ToXX((1+q)^500*x3*x4,noexpand,'x',collect):
> ToXX(q^2*x3*X[3,1,2], collect);

  2                    2                              2   2
 q  XX[3, 1, 4, 2] + q  (y2 + y1) XX[2, 1, 4, 3] + q  y1  XX[1, 2, 4, 3]

      2                 2                             2   2
   - q  XX[3, 2, 1] - q  (y2 + y1) XX[2, 3, 1] - q  y1  XX[1, 3, 2]

      2                    2                            2   2
   - q  y1 XX[3, 1, 2] + q  y3 (y2 + y1) XX[2, 1] + q  y1  y3 XX[1]
```

5.16.17 `ToY` — convert any expression to the $\mathbf{Y}$ Schubert basis

Calling Sequence:

`ToY`(*expr*)

`ToY`(*expr*, *b*)

`SP[ToY]`(*expr*)

`SP[ToY]`(*expr*, *b*)

Parameters:

expr — any expression

b — any name of a known basis

Synopsis:

- The `ToY` function converts any expression *expr* to the $\mathbf{Y}$ Schubert basis. *expr* may involve some x_i's, simple Schubert polynomials ($\mathbf{X}_{perm}, \mathbf{Y}_{code}$), double Schubert polynomials ($\mathbb{X}_{perm}, \mathbb{Y}_{code}$, the second alphabet being the y_i's), other terms being considered as coefficients.
- The expression *expr* is expanded and the result is not collected.
- One may specify by a second argument, say *b*, that *expr* is solely expressed in terms of the known basis *b* (`x, X, Y, XX, YY` and even `y` that is seen as a basis in the package).
- The call `ToY`(*expr*, `'Y'`) does not affect the argument *expr*.
- One may add `'noexpand'` just after the argument *expr* to choose not to expand the expression *expr* before treating it.
- One may collect the result by adding a third argument: this is done by `ToY`(*expr*, *b*, `'collect'`). For instance, `ToY`(*expr*, `'Y', 'collect'`) may be used to collect the argument *expr*.
- Whenever there is a conflict between the function name `ToY` and another name used in the same session, use the long form `SP['ToY']`.

Examples:

```
> with(SP):
> ToY((1+q)^500*x3*x4):                     # expands the input
> ToY((1+q)^500*x3*x4,noexpand):            # does not expand (1+q)^500
> ToY((1+q)^500*x3*x4,collect):             # collects the result
> ToY((1+q)^500*x3*x4,noexpand,'x'):        # the most efficient
> ToY((1+q)^500*x3*x4,'x',collect):         # specifies a basis
> ToY((1+q)^500*x3*x4,noexpand,'x',collect):
> ToY(q^2*x3*XX[3,1,2]*Y[0,0,1], collect):

         2                  2                  2
      - q  Y[3, 1, 0, 0] - q  Y[2, 2, 0, 0] + q  Y[2, 0, 2, 0, 0]

           2                             2
        + q  (y1 + y2) Y[3, 0, 0, 0] + q  (y1 + y2) Y[1, 2, 0, 0]
```

```
      2                            2
- q  (y1 + y2) Y[1, 0, 2, 0, 0] -  q  y2 y1 Y[0, 2, 0, 0]

      2
+ q  y2 y1 Y[0, 0, 2, 0, 0]
```

SEE ALSO: TYP[Schubert], Tox

5.16.18 Tox — express any expression in the basis of monomials

Calling Sequence:

Tox(*expr*)

Tox(*expr*, *b*)

SP[Tox](*expr*)

SP[Tox](*expr*, *b*)

Parameters:

expr — any expression

b — any name of a known basis

Synopsis:

- The Tox function expresses any expression *expr* in the basis of monomials. *expr* may involve some x_i's, simple Schubert polynomials ($\mathbf{X}_{perm}, \mathbf{Y}_{code}$), double Schubert polynomials ($\mathbb{X}_{perm}, \mathbb{Y}_{code}$, the second alphabet being the y_i's), other terms being considered as coefficients.
- The expression *expr* is expanded and the result is not collected.
- One may specify by a second argument, say *b*, that *expr* is solely expressed in terms of the known basis *b* (x, X, Y, XX, YY and even y that is seen as a basis in the package).
- The call Tox(*expr*, 'x') does not affect the argument *expr*.
- One may add 'noexpand' just after the argument *expr* to choose not to expand the expression *expr* before treating it.
- One may collect the result by adding a third argument: this is done by Tox(*expr*, *b*, 'collect'). For instance, Tox(*expr*, 'x', 'collect') may be used to collect the argument *expr*.
- Whenever there is a conflict between the function name Tox and another name used in the same session, use the long form SP['Tox'].

Examples:

```
> with(SP):
> Tox((1+q)^500*X[3,1,2]):                # expands the input
> Tox((1+q)^500*X[3,1,2],noexpand):       # does not expand (1+q)^500
> Tox((1+q)^500*X[3,1,2],collect):        # collects the result
> Tox((1+q)^500*X[3,1,2],noexpand,'X'):   # the most efficient
> Tox((1+q)^500*X[3,1,2],'X',collect):    # specifies a basis
```

```
> Tox((1+q)^500*X[3,1,2],noexpand,'X',collect):
> factor( Tox(q^2*x3*XX[3,1,2]*Y[0,0,1]) );

                                  2
                 (x3 + x1 + x2) q  x3 (x1 - y2) (x1 - y1)

> Tox(X[1,3,2]*XX[2,3,1] - q^2*YY[0,1], collect);

  2       2        2                    2        2         2    2
 q  y2 + q  y1 - x2  y1 - 2 x2 y1 x1 + x2  x1 + x1  x2 + (y1  - q ) x2

            2    2         2
      + (y1  - q ) x1 - x1  y1
```

SEE ALSO: ToX

5.16.19 X2Dim — specialization in X Schubert basis

Calling Sequence:

X2Dim(*schub*)

SP[X2Dim](*schub*)

Parameters:

schub — any expression in the X Schubert basis

Synopsis:

- The X2Dim function specializes in $x_i = 1$ all Schubert polynomials in schub, expressed in **X** Schubert basis.
- Whenever there is a conflict between the function name X2Dim and another name used in the same session, use the long form SP['X2Dim'].

Examples:

```
> with(SP):
> X2Dim(q^2*X[3,1,2] - 2*q*X[2,3,1,5,4] + q^3);

                                2          3
                               q  - 8 q + q
```

SEE ALSO: X2Dimq

5.16.20 X2Dimq — q-specialization in X Schubert basis

Calling Sequence:

X2Dimq(*schub*)

SP[X2Dimq](*schub*)

Parameters:

schub — any expression in the X Schubert basis

Synopsis:

- The `X2Dimq` function specializes in $x_i = q^{(i-1)}$ all Schubert polynomials in *schub*, expressed in **X** Schubert basis.
- Whenever there is a conflict between the function name `X2Dimq` and another name used in the same session, use the long form `SP['X2Dimq']`.

Examples:

```
> with(SP):
> X2Dimq(X[2,3,1,5,4] - a*X[3,1,4,2]);

                            4    3    2            2
                           q  + q  + q  + q - a (q  + q)
```

SEE ALSO: `X2Dim`

5.16.21 `X2Y` — from X Schubert basis to Y Schubert basis

Calling Sequence:

`X2Y`(*schub*)

`SP[X2Y]`(*schub*)

Parameters:

schub — any expression in the **X** Schubert basis

Synopsis:

- The `X2Y` function converts any expression *schub* from **X** Schubert basis (Schubert polynomials indexed by permutations) to **Y** Schubert basis (Schubert polynomials indexed by sequences of integers interpreted as codes of permutations).
- The result is not expanded.
- One may add `'noexpand'` just after the argument *schub* to choose not to expand the expression *schub* before treating it.
- One may add `'collect'` just after the argument *schub* or just after the argument `'noexpand'` to collect the result.
- Whenever there is a conflict between the function name `X2Y` and another name used in the same session, use the long form `SP['X2Y']`.

Examples:

```
> with(SP):
> X2Y((1+q)^500*X[3,1,2]):                     # expands the input
> X2Y((1+q)^500*X[3,1,2],noexpand):            # does not expand (1+q)^500
> X2Y((1+q)^500*X[3,1,2],collect):             # collects the result
> X2Y((1+q)^500*X[3,1,2],noexpand,collect):
> X2Y((X[3,1,2] - q*X[3,1,2])*(1 + q) + X[3,1,2]*q^4, collect);
```

```
                              2    4
                        (1 - q  + q ) Y[2, 0, 0]

> X2Y(q^2*X[3,1,2] - 2*q*X[2,3,1,5,4] + q^3, noexpand);

                 2                                      3
                q  Y[2, 0, 0] - 2 q Y[1, 1, 0, 1, 0] + q
```

SEE ALSO: Y2X

5.16.22 XX2YY — from $\mathbb{X}$ Schubert basis to $\mathbb{Y}$ Schubert basis

Calling Sequence:

XX2YY(*schub*)

SP[XX2YY](*schub*)

Parameters:

schub — any expression in $\mathbb{X}$ Schubert basis

Synopsis:

- The XX2YY function converts any expression *schub* from $\mathbb{X}$ Schubert basis (double Schubert polynomials indexed by permutations) to $\mathbb{Y}$ Schubert basis (double Schubert polynomials indexed by sequences of integers interpreted as codes of permutations).
- The result is not expanded.
- One may add 'noexpand' just after the argument *schub* to choose not to expand the expression *schub* before treating it.
- One may add 'collect' just after the argument *schub* or just after the argument 'noexpand' to collect the result.
- Whenever there is a conflict between the function name XX2YY and another name used in the same session, use the long form SP['XX2YY'].

Examples:

```
> with(SP):
> XX2YY((1+q)^500*XX[3,1,2]):                    # expands the input
> XX2YY((1+q)^500*XX[3,1,2],noexpand):           # does not expand (1+q)^500
> XX2YY((1+q)^500*XX[3,1,2],collect):            # collects the result
> XX2YY((1+q)^500*XX[3,1,2],noexpand,collect):
> XX2YY(q^2*XX[3,1,2] - 2*q*XX[2,3,1,5,4] + q^3);

                2                                        3
               q  YY[2, 0, 0] - 2 q YY[1, 1, 0, 1, 0] + q
```

SEE ALSO: YY2XX

5.16.23 XX2x — from $\mathbb{X}$ Schubert basis to the basis of monomials

Calling Sequence:

XX2x(*schub*)

XX2x(*schub*, *perm*)

SP[XX2x](*schub*)

SP[XX2x](*schub*, *perm*)

Parameters:

schub — any expression in $\mathbb{X}$ Schubert basis

perm — any list denoting a permutation

Synopsis:

- The XX2x function converts an expression from $\mathbb{X}$ Schubert basis to the basis of monomials.
- The result is not expanded.
- When the function is called with the second argument *perm*, the second alphabet (the y_i's) is specialized as the permutation *perm* of the first one (the x_i's): in other words, y_i is sent to $x_{perm[i]}$.
- Whenever there is a conflict between the function name XX2x and another name used in the same session, use the long form SP['XX2x'].

Examples:

```
> with(SP):
> XX2x(XX[4,1,3,2]);

                (x3 - y2) (x1 - y3) (x1 - y2) (x1 - y1)

              + (x2 - y1) (x1 - y3) (x1 - y2) (x1 - y1)

> XX2x(2*XX[4,1,2,3] + 4*q^2*XX[3,1,2]);

                                                  2
        2 (x1 - y3) (x1 - y2) (x1 - y1) + 4 q  (x1 - y2) (x1 - y1)

> XX2x(y3^2*XX[1,2,3,5,4], [5,4,3,2,1]);

                                2
                              x3  (x1 - x5)
```

SEE ALSO:

5.16.24 Y2X — from $\mathbf{Y}$ Schubert basis to $\mathbf{X}$ Schubert basis

Calling Sequence:

Y2X(*schub*)

SP[Y2X](*schub*)

Parameters:

schub — any expression in the **Y** Schubert basis

Synopsis:

- The `Y2X` function converts any expression from **Y** Schubert basis (Schubert polynomials indexed by sequences of integers interpreted as codes of permutations) to **X** Schubert basis (Schubert polynomials indexed by permutations).
- The result is not expanded.
- One may add `'noexpand'` just after the argument *schub* to choose not to expand the expression *schub* before treating it.
- One may add `'collect'` just after the argument *schub* or just after the argument `'noexpand'` to collect the result.
- Whenever there is a conflict between the function name `Y2X` and another name used in the same session, use the long form `SP['Y2X']`.

Examples:

```
> with(SP):
> X2Y((1+q)^500*X[3,1,2]):                    # expands the input
> X2Y((1+q)^500*X[3,1,2],noexpand):           # does not expand (1+q)^500
> X2Y((1+q)^500*X[3,1,2],collect):            # collects the result
> X2Y((1+q)^500*X[3,1,2],noexpand,collect):
> Y2X((Y[1,2,0,1]-q*Y[1,2,0,1])*(1+q) + Y[1,2,0,1]*q^4, collect);

                                 2    4
                           (1 - q  + q ) X[2, 4, 1, 5, 3]

> Y2X(q^2*Y[2] - 2*q*Y[1,0,0] + Y[4,0,2], noexpand);

               2
              q  X[3, 1, 2] - 2 q X[2, 1, 3] + X[5, 1, 4, 2, 3]
```

SEE ALSO: `X2Y`

5.16.25 `YY2XX` — from $\mathbb{Y}$ Schubert basis to $\mathbb{X}$ Schubert basis

Calling Sequence:

`YY2XX`(*schub*)

`SP[YY2XX]`(*schub*)

Parameters:

schub — any expression in $\mathbb{Y}$ Schubert basis

Synopsis:

- The `XX2YY` function converts any expression *schub* from $\mathbb{Y}$ Schubert basis (double Schubert polynomials indexed by sequences of integers interpreted as codes of permutations) to $\mathbb{X}$ Schubert basis (double Schubert polynomials indexed by permutations).
- The result is not expanded.
- One may add `'noexpand'` just after the argument *schub* to choose not to expand the expression *schub* before treating it.
- One may add `'collect'` just after the argument *schub* or just after the argument `'noexpand'` to collect the result.
- Whenever there is a conflict between the function name `YY2XX` and another name used in the same session, use the long form `SP['YY2XX']`.

Examples:

```
> with(SP):
> YY2XX((1+q)^500*YY[3,1,2]):                # expands the input
> YY2XX((1+q)^500*YY[3,1,2],noexpand):       # does not expand (1+q)^500
> YY2XX((1+q)^500*YY[3,1,2],collect):        # collects the result
> YY2XX(q^2*YY[2] - 2*q*YY[1,0,0] + YY[4,0,2]);

          2
         q  XX[3, 1, 2] - 2 q XX[2, 1, 3] + XX[5, 1, 4, 2, 3]
```

SEE ALSO: `YY2XX`

5.16.26 `YY2x` — from $\mathbb{Y}$ Schubert basis to the basis of monomials

Calling Sequence:

`YY2x`(*schub*)

`YY2x`(*schub*, *perm*)

`SP[YY2x]`(*schub*)

`SP[YY2x]`(*schub*, *perm*)

Parameters:

schub — any expression in $\mathbb{Y}$ Schubert basis

perm — any list denoting a permutation

Synopsis:

- The `YY2x` function converts an expression from $\mathbb{Y}$ Schubert basis to the basis of monomials.
- The result is not expanded.
- When the function is called with the second argument *perm*, the second alphabet (the y_i's) is specialized as the permutation *perm* of the first one (the x_i's): in other words, y_i is sent to $x_{perm[i]}$.
- Whenever there is a conflict between the function name `YY2x` and another name used in the same session, use the long form `SP['YY2x']`.

Examples:

```
> with(SP):
> YY2x(YY[3,0,1,0]);

                (x3 - y2) (x1 - y3) (x1 - y2) (x1 - y1)

              + (x2 - y1) (x1 - y3) (x1 - y2) (x1 - y1)

> YY2x(2*YY[3] + 4*q^2*YY[2,0]);

                                                2
        2 (x1 - y3) (x1 - y2) (x1 - y1) + 4 q  (x1 - y2) (x1 - y1)

> YY2x(y3^2*YY[0,0,0,1], [5,4,3,2,1]);

                              2
                            x3  (x1 - x5)
```

SEE ALSO:

5.16.27 x2X — from the basis of monomials to X Schubert basis

Calling Sequence:

`x2X`(*pol*)

`SP[x2X]`(*pol*)

Parameters:

pol — any expression in the basis of monomials

Synopsis:

- The `x2X` function converts any expression from the basis of monomials to **X** Schubert basis.
- The result is not expanded.
- One may add `'noexpand'` just after the argument *pol* to choose not to expand the expression *pol* before treating it.
- One may add `'collect'` just after the argument *pol* or just after the argument `'noexpand'` to collect the result.
- Whenever there is a conflict between the function name `x2X` and another name used in the same session, use the long form `SP['x2X']`.

Examples:

```
> with(SP):
> x2X((1+q)^500*x1*x3^3):                      # expands the input
> x2X((1+q)^500*x1*x3^3,noexpand):             # does not expand (1+q)^500
> x2X((1+q)^500*x1*x3^3,collect):              # collects the result
> x2X((1+q)^500*x1*x3^3,noexpand,collect):
> x2X(2*x1^3 + 4*q^2*x1^2);
                                            2
                     2 X[4, 1, 2, 3] + 4 q  X[3, 1, 2]
```

SEE ALSO: `X2x`

5.16.28 x2XX — from the basis of monomials to $\mathbb{X}$ Schubert basis

Calling Sequence:

x2XX(*pol*)

SP[x2XX](*pol*)

Parameters:

pol — any expression on monomial basis

Synopsis:

- The x2XX function converts any expression from the basis of monomials to $\mathbb{X}$ Schubert basis.
- The result is not expanded.
- One may add 'noexpand' just after the argument *pol* to choose not to expand the expression *pol* before treating it.
- One may add 'collect' just after the argument *pol* or just after the argument 'noexpand' to collect the result.
- Whenever there is a conflict between the function name x2XX and another name used in the same session, use the long form SP['x2XX'].

Examples:

```
> with(SP):
> x2XX((1+q)^500*x1*x3^3):                      # expands the input
> x2XX((1+q)^500*x1*x3^3,noexpand):             # does not expand (1+q)^500
> x2XX((1+q)^500*x1*x3^3,collect):              # collects the result
> x2XX((1+q)^500*x1*x3^3,noexpand,collect):
> x2XX(2*x1^3 + 4*q^2*x1^2, collect);

          3      2   2
     (2 y1  + 4 q  y1 ) XX[1]

                         2        2       2          2
      + (2 y2 y1 + 2 y1  + 2 y2  + 4 q  y2 + 4 q  y1) XX[2, 1]

                                     2
      + (2 y1 + 2 y3 + 2 y2 + 4 q ) XX[3, 1, 2] + 2 XX[4, 1, 2, 3]

> expand( XX2x(") );

                                          3      2   2
                                    2 x1  + 4 q  x1
```

SEE ALSO: NewtonInterp, XX2x

5.17 Computations in the ring $\mathbb{Z}[x_1, \ldots, x_n]$ with FM

✓

Calling Sequence:

`FM[<function>]`(*args*)

`<function>`(*args*)

Synopsis:

- This package provides functions to compute in the polynomial ring in several, say n, variables, seen as a free module of rank $n!$ over the ring of symmetric polynomials in n variables. We choose as basis of this module Schubert polynomials indexed by permutations in the symmetric group of degree n.
- By default, one computes in the polynomial ring in 4 variables x_1, x_2, x_3 and x_4. In this case, $x_5, x6, \ldots$ are considered as scalars.
- Whenever there is a conflict between a function name in `FM` and another name used in the same session, use the long form `FM[<function>]`.
- The available functions are:

FM_n	Monkfix	ToXfix	ToXfixScal
x2Xfix	x2XfixScal	X2Xfix	Xfix2x
Xfix2X			

- For help with a particular function do either `?FM[<function>]` or `?FM,<function>` where **<function>** is one from the above list.
- The functions defined in the package provide a type checking on their arguments. All these functions may be called by their names in the package.

Examples:

```
> with(FM):
> x2XfixScal(q^2*x2^3*x4);

    2                             2
   q  s[1, 1, 1] X[2, 1, 3, 4] - q  s[1, 1] X[2, 3, 1, 4]

    2                          2                           2
 - q  s[1, 1] X[3, 1, 2, 4] + q  s[1, 1] X[1, 4, 2, 3] + q  X[2, 4, 3, 1]

        2                  2                  2
     + q  X[3, 4, 1, 2] + q  X[4, 1, 3, 2] - q  s[1, 1, 1] X[1, 2, 4, 3]

        2                          2
     + q  s[1, 1] X[1, 3, 4, 2] - q  s[1] X[1, 4, 3, 2]

> Xfix2x(");

                                  2   3
                                 q  x2  x4
```

5.17.1 `FM_n` — fix the number of variables

✘

Calling Sequence:

`FM_n`(n)

`FM_n`(n, `'load'`)

`FM[FM_n]`(n)

`FM[FM_n]`(n, `'load'`)

Parameters:

n — any positive integer

Synopsis:

- The `FM_n` function is used to fix the number of variables. This value is stored in the `_FMn` global variable.
- When called without any argument, it returns the current value of `_FMn`.
- When called with a second paramater `'load'`, it loads all pre-computed tables of available data for the given n. If one table is not available, it computes it so that it can take some time.
- Whenever there is a conflict between the function name `FM_n` and another name used in the same session, use the long form `FM['FM_n']`.

✄...

Examples:

```
> with(FM):
> FM_n();

                                        4

> FM_n(3, 'load');

                                        3
```

....✄

SEE ALSO: `ACE[BuildData]`, `ACE[LoadData]`

5.17.2 `Monkfix` — Monk's formula

✘

Calling Sequence:

`Monkfix`($k, schub$)

`FM[Monkfix]`($k, schub$)

Parameters:

k — any positive integer ($1 \leq k \leq$ `_FMn`)

$schub$ — any Schubert polynomial indexed by a permutation

Synopsis:

- The `Monkfix` function computes the product of the variable x_k by a simple Schubert polynomial $schub$, applying Monk formula. The result is expressed in the free module.

- Schubert polynomial schub should be indexed by a permutation of the symmetric group of degree `_FMn`.
- Whenever there is a conflict between the function name `Monkfix` and another name used in the same session, use the long form `FM['Monkfix']`.

Examples:

```
> with(FM):
> FM_n();

                                        4

> Monkfix(3, X[3,1,4,2]);

      - X[3, 4, 1, 2] - X[4, 1, 3, 2] + s[1, 1, 1, 1] X[1, 2, 3, 4]

          - s[1, 1, 1] X[2, 1, 3, 4] + s[1] X[3, 1, 4, 2]
```

SEE ALSO: `FM_n`, `SP['Monk']`

5.17.3 `X2Xfix` — from X Schubert basis to X Schubert basis of the free module

Calling Sequence:

`X2Xfix`(*schub*)

`FM[X2Xfix]`(*schub*)

Parameters:

schub — any expression in **X** Schubert basis

Synopsis:

- The `X2Xfix` function converts an expression from **X** Schubert basis to the **X** Schubert basis of the free module over symmetric polynomials.
- All Schubert polynomials are indexed by permutations in the symmetric group of degree `_FMn`. Coefficients are symmetric polynomials expressed in the Schur basis.
- The result is not expanded.
- One may add `'noexpand'` just after the argument *schub* to choose not to expand the expression *schub* before treating it.
- One may add `'collect'` just after the argument *schub* or just after the argument `'noexpand'` to collect the result.
- Whenever there is a conflict between the function name `X2Xfix` and another name used in the same session, use the long form `FM['X2Xfix']`.

Examples:

```
> with(FM):
> FM_n();

                                    4

> X2Xfix((1+q)^500*X[3,5,2,4,1]):              # expands the input
> X2Xfix((1+q)^500*X[3,5,2,4,1],noexpand):     # does not expand (1+q)^500
> X2Xfix((1+q)^500*X[3,5,2,4,1],collect):      # collects the result
> X2Xfix((1+q)^500*X[3,5,2,4,1],noexpand,collect):
> X2Xfix(X[3,2,1] + q*X[3,2,1] + z*X[4,5,1,3,2], collect);

           (1 + q) X[3, 2, 1, 4] - z s[1, 1, 1, 1] X[2, 4, 1, 3]

              + z s[1, 1, 1] X[3, 4, 1, 2]
```

SEE ALSO: Xfix2X

5.17.4 ToXfix — convert any expression to the X Schubert basis of the free module ✘

Calling Sequence:

ToXfix(*expr*)

ToXfix(*expr*, *b*)

FM[ToXfix](*expr*)

FM[ToXfix](*expr*, *b*)

Parameters:

expr — any expression

b — any name of a known basis

Synopsis:

- The ToXfix function converts any expression *expr* to the **X** Schubert basis of the free module over symmetric polynomials. *expr* may involve some x_i's, simple Schubert polynomials ($\mathbf{X}_{perm}, \mathbf{Y}_{code}$), other terms being considered as coefficients.

- The expression *expr* is expanded and the result is not collected.

- One may specify by a second argument, say *b*, that *expr* is solely expressed in terms of the known basis *b* (x, X, Y).

- The call ToXfix(*expr*, 'X') does not affect the argument *expr*.

- One may add 'noexpand' just after the argument *expr* to choose not to expand the expression *expr* before treating it.

- One may collect the result by adding a third argument: ToXfix(*expr*, *b*, 'collect'). For instance, the following instruction ToXfix(*expr*, 'X', 'collect') may be used to collect the argument *expr*.

- Whenever there is a conflict between the function name ToXfix and another name used in the same session, use the long form FM['ToXfix'].

Examples:

```
> with(FM):
> FM_n();

                                        4

> ToXfix((1+q)^500*x3*x4):                    # expands the input
> ToXfix((1+q)^500*x3*x4,noexpand):           # does not expand (1+q)^500
> ToXfix((1+q)^500*x3*x4,collect):            # collects the result
> ToXfix((1+q)^500*x3*x4,noexpand,'x'):       # the most efficient
> ToXfix((1+q)^500*x3*x4,'x',collect):        # specifies a basis
> ToXfix((1+q)^500*x3*x4,noexpand,'x',collect):
> ToXfix(q^2*x3*X[3,1,4,2]*Y[0,1], collect);

    2              2                 2
 - q  X[1, 3, 2, 4]  X[1, 2, 3, 4] + q  s[1] X[3, 4, 1, 2]

    2                      2                      2
 + q  s[1] X[3, 2, 4, 1] + q  s[1] X[4, 1, 3, 2] + q  X[3, 4, 2, 1]

    2                 2
 + q  X[4, 2, 3, 1] - q  s[1, 1, 1] X[2, 3, 1, 4]

    2                            2
 - q  s[1, 1, 1] X[3, 1, 2, 4] + q  s[1, 1, 1, 1] X[1, 3, 2, 4]
```

SEE ALSO: ToXfixScal

5.17.5 ToXfixScal — convert any expression to the $\mathbf{X}$ Schubert basis of the free module

Calling Sequence:

ToXfixScal($expr$)

ToXfixScal($expr, b$)

FM[ToXfixScal]($expr$)

FM[ToXfixScal]($expr, b$)

Parameters:

$expr$ — any expression

b — any name of a known basis

Synopsis:

- The ToXfixScal function converts any expression $expr$ to the $\mathbf{X}$ Schubert basis of the free module over symmetric polynomials, by using scalar products instead of Monk formula when possible. This function is useful for _FMn ≤ 5 especially when pre-computed data tables are loaded. $expr$ may involve some x_i's, simple Schubert polynomials ($\mathbf{X}_{perm}, \mathbf{Y}_{code}$), other terms being considered as coefficients.

- The expression *expr* is expanded and the result is not collected.
- One may specify by a second argument, say b, that *expr* is solely expressed in terms of the known basis b (`x, X, Y`).
- The call `ToXfixScal`(*expr*, `'X'`) does not affect the argument *expr*.
- One may add `'noexpand'` just after the argument *expr* to choose not to expand the expression *expr* before treating it.
- One may collect the result by adding a third argument: `ToXfixScal`(*expr*, b, `'collect'`). For instance, the next instruction `ToXfixScal`(*expr*, `'X'`, `'collect'`) may be used to collect its argument.
- Whenever there is a conflict between the function name `ToXfixScal` and another name used in the same session, use the long form `FM['ToXfixScal']`.

Examples:

```
> with(FM):
> FM_n();

                                     4

> ToXfixScal((1+q)^500*x3*x4):                # expands the input
> ToXfixScal((1+q)^500*x3*x4,noexpand):       # does not expand (1+q)^500
> ToXfixScal((1+q)^500*x3*x4,collect):        # collects the result
> ToXfixScal((1+q)^500*x3*x4,noexpand,'x'):   # the most efficient
> ToXfixScal((1+q)^500*x3*x4,'x',collect):    # specifies a basis
> ToXfixScal((1+q)^500*x3*x4,noexpand,'x',collect):
> ToXfixScal(q^2*x3*X[3,1,4,2]*Y[0,1], collect);

    2             2                  2
 - q  X[1, 3, 2, 4]  X[1, 2, 3, 4] + q  s[1] X[3, 4, 1, 2]

    2                    2                        2
 + q  s[1] X[3, 2, 4, 1] + q  s[1] X[4, 1, 3, 2] + q  X[3, 4, 2, 1]

    2                2
 + q  X[4, 2, 3, 1] - q  s[1, 1, 1] X[2, 3, 1, 4]

    2                          2
 - q  s[1, 1, 1] X[3, 1, 2, 4] + q  s[1, 1, 1, 1] X[1, 3, 2, 4]
```

SEE ALSO: `ToXfix`

5.17.6 `Xfix2X` — from X Schubert basis of the free module to X Schubert basis ✘

Calling Sequence:

`Xfix2X`(*schub*)

`FM[Xfix2X]`(*schub*)

Parameters:

schub — any expression in **X** Schubert basis

Synopsis:

- The `Xfix2x` function converts an expression from **X** Schubert basis of the free module over symmetric polynomials to the **X** Schubert basis.
- The result is not expanded.
- One may add `'noexpand'` just after the argument *schub* to choose not to expand the expression *schub* before treating it.
- One may add `'collect'` just after the argument *schub* or just after the argument `'noexpand'` to collect the result.
- We recommend to use the `collect` option to have the result simplified.
- Whenever there is a conflict between the function name `Xfix2X` and another name used in the same session, use the long form `FM['Xfix2X']`.

Examples:

```
> with(FM):
> FM_n();

                                  4

> x2Xfix(q*x2^2*x4 - x3^3*x4, collect):
> Xfix2X(", noexpand, collect);

  - X[1, 2, 6, 4, 3, 5] + q X[4, 1, 2, 3] - q X[1, 4, 3, 2]

  + q X[2, 4, 1, 3] + X[1, 3, 5, 4, 2] + q X[2, 3, 4, 1] + X[2, 4, 3, 1]

  - X[2, 3, 4, 5, 1] + X[1, 5, 2, 4, 3] - X[2, 4, 1, 5, 3]

  + X[3, 4, 1, 2] + q X[1, 4, 2, 5, 3] - q X[2, 3, 1, 5, 4]

  - q X[3, 1, 2, 5, 4] + q X[3, 1, 4, 2] + X[1, 3, 6, 2, 4, 5]

  - X[2, 3, 5, 1, 4] - X[1, 5, 3, 2, 4]
```

SEE ALSO: `X2Xfix`

5.17.7 `Xfix2x` — from X Schubert basis of the free module to the basis of monomials

Calling Sequence:

`Xfix2x`(*schub*)

`FM[Xfix2x]`(*schub*)

Parameters:

schub — any expression in **X** Schubert basis

Synopsis:

- The `Xfix2x` function converts an expression from **X** Schubert basis of the free module over symmetric polynomials to the basis of monomials.

- The result is not expanded.
- Whenever there is a conflict between the function name `Xfix2x` and another name used in the same session, use the long form `FM['Xfix2x']`.

Examples:

```
> with(FM):
> FM_n();

                                     4

> x2Xfix(q*x2^2*x4 - x3^3*x4, collect):
> Xfix2x(");

                                 2        3
                            q x2  x4 - x3  x4
```

SEE ALSO: `x2Xfix`

5.17.8 `x2Xfix` — from the basis of monomials to X Schubert basis of the free module

Calling Sequence:

`x2Xfix`(*pol*)

`FM[x2Xfix]`(*pol*)

Parameters:

pol — any expression in the basis of monomials

Synopsis:

- The `x2Xfix` function converts any expression from the basis of monomials to the **X** Schubert basis of the free module over symmetric polynomials.
- All Schubert polynomials are indexed by permutations in the symmetric group of degree `_FMn`. Coefficients are symmetric polynomials expressed in the Schur basis.
- The result is not expanded.
- One may add `'noexpand'` just after the argument *pol* to choose not to expand the expression *pol* before treating it.
- One may add `'collect'` just after the argument *pol* or just after the argument `'noexpand'` to collect the result.
- Whenever there is a conflict between the function name `x2Xfix` and another name used in the same session, use the long form `FM['x2Xfix']`.

Examples:

```
> with(FM):
> FM_n();

                                    4

> x2Xfix((1+q)^500*x1*x3^3):                   # expands the input
> x2Xfix((1+q)^500*x1*x3^3,noexpand):          # does not expand (1+q)^500
> x2Xfix((1+q)^500*x1*x3^3,collect):           # collects the result
> x2Xfix((1+q)^500*x1*x3^3,noexpand,collect):
> x2Xfix(q*x2^2*x4 - x3^3*x4, collect);

   (s[2, 2] - q s[1, 1, 1]) X[1, 2, 3, 4] - X[1, 2, 4, 3] s[2, 1]

 + (- q s[1] - s[1, 1]) X[2, 3, 1, 4] + (s[2] + s[1, 1]) X[1, 3, 4, 2]

 + s[1, 1] X[1, 4, 2, 3] + (- s[1] + q) X[2, 3, 4, 1]

 - q s[1] X[3, 1, 2, 4] + X[3, 4, 1, 2] + (- q - s[1]) X[1, 4, 3, 2]

 + q X[3, 1, 4, 2] + X[2, 4, 3, 1] + q s[1, 1] X[1, 3, 2, 4]

 + q X[4, 1, 2, 3] + q X[2, 4, 1, 3]
```

SEE ALSO: `FM_n`, `x2XfixScal`, `Xfix2x`

5.17.9 `x2XfixScal` — from the basis of monomials to X Schubert basis of the free module

✘

Calling Sequence:

`x2XfixScal`(*pol*)

`FM[x2XfixScal]`(*pol*)

Parameters:

pol — any expression in the basis of monomials

Synopsis:

- The `x2XfixScal` function converts any expression from the basis of monomials to the **X** Schubert basis of the free module over symmetric polynomials by computing scalar products with the adjoint basis of the **X** Schubert basis.
- All Schubert polynomials are indexed by permutations in the symmetric group of degree `_FMn`. Coefficients are symmetric polynomials expressed in the Schur basis.
- This function is quite efficient for small values of `_FM`: for instance, `_FM` ≤ 5.
- The result is not expanded but collected.
- Whenever there is a conflict between the function name `x2XfixScal` and another name used in the same session, use the long form `FM['x2XfixScal']`.

Examples:

```
> with(FM):
> FM_n();

                                       4

> x2XfixScal(q*x2^2*x4 - x3^3*x4);

   (s[2, 2] - q s[1, 1, 1]) X[1, 2, 3, 4] - X[1, 2, 4, 3] s[2, 1]

 + (- q s[1] - s[1, 1]) X[2, 3, 1, 4] + (s[2] + s[1, 1]) X[1, 3, 4, 2]

 + s[1, 1] X[1, 4, 2, 3] + (- s[1] + q) X[2, 3, 4, 1]

 - q s[1] X[3, 1, 2, 4] + X[3, 4, 1, 2] + (- q - s[1]) X[1, 4, 3, 2]

 + q X[3, 1, 4, 2] + X[2, 4, 3, 1] + q s[1, 1] X[1, 3, 2, 4]

 + q X[4, 1, 2, 3] + q X[2, 4, 1, 3]
```

SEE ALSO: FM_n, x2Xfix, Xfix2x

5.18 TEX output

✓

Calling Sequence:

TEX[<function>]($args$)

<function>($args$)

Synopsis:

- This package provides an environment of functions that produce output which is suitable with a TEX processor. It knows how to format almost all the types defined in *Maple*, while it knows how to traduce all standard internal Maple functions or objects such as integrals, limits, sums, products, and even arrays and tables.
- The mathematical format is taken, in general, from the command `latex` so that it is said to be as in the CRC handbook or the Handbook of Mathematical functions.
- Almost all the mathematical objects manipulated by `ACE` have a default corresponding function describing the output format. This is the case for symmetric function bases, noncommutative symmetric function bases, quasi-symmetric function bases, Schubert polynomials, ..., but it is still possible to modify these default formats.
- Whenever there is a conflict between a function name in `TEX` and another name used in the same session, use the long form `TEX[<function>]`.
- The available functions are:

TeX	TeXconcat	TeXer	TeXsequence
TeXstring			

- It is possible to extend the abilities of `TEX` to format other objects by defining procedures named by:

  `‘TeX/UserdefinedFun/`$< name >$`‘` to define a format for the function whose name is $< name >$,

  `‘TeX/UserdefinedName/`$< name >$`‘` to define a format for the string whose name is $< name >< number >$.

  `‘TeX/UserdefinedIndexed/`$< name >$`‘` to define a format for the indexed object whose name is $< name >$.

- When it encounters a call to `<name>`(...), the `TeX` function will call the procedure (if it exists) `‘TeX/UserdefinedFun/`$< name >$`‘`.

- When it encounters a name $< name >< number >$, the `TeX` function will call the procedure (if it exists) `‘TeX/UserdefinedName/`$< name >$`‘` with the parameter $< number >$.

- When it encounters a name $< name >$ [*sequence*], the `TeX` function will call the procedure (if it exists) `‘TeX/UserdefinedIndexed/`$< name >$`‘` with the sequence of parameters *sequence*.

- Note that the output may include the commands to invoke the TeX math environment: this can be done by positionning some variables...

- For help with a particular function do `?TEX[<function>]` or `?TEX,<function>` where `<function>` is one from the above list.

Examples:

```
> with(TEX):
> TeX(x^2 + y^2 = z^2);
x^{2}+y^{2}=z^{2}
> TeX(Int(1/(x^2+1), x) = int(1/(x^2+1), x),'TEX_MODE'='MATHS');
$\\int \\! {{1\\over x^{2}+1}}{dx}=\\arctan (x) $
> TeXstring(\\end);
\\end
```

SEE ALSO: `TEX[advanced]`, `TEX[functions]`, `TEX[indexed]`, `TEX[names]`, `TEX[variables]`

5.18.1 `TeX` — produce TeX output

Calling Sequence:

`TeX`(*expr*)

`TeX`(*expr*, *args*)

`TEX[TeX]`(*expr*)

`TEX[TeX]`(*expr*, *args*)

Parameters:

expr — any expression

args — (optional) sequence of expressions $< var >=< val >$

Synopsis:

- The `TeX` function produces ouput suitable for TEX printing for the expression *expr*.
- When called with other parameters, it provides a way to control the output. Options have the Maple type `=` so that the left operand is a known variable (see `TEX[variables]`) and the right operand is a value, compatible with this variable.
- Whenever there is a conflict between the function name `TeX` and another name used in the same session, use the long form `TEX['TeX']`.

Examples:

```
> with(TEX):
> TeX(-b+abc+'cos'(4*Pi/3));
-b+{\\it abc}+\\cos (4/3\\pi )
> TeX(Int(x,y),'TEX_MODE'='DISPLAY');
$$\\int \\! {x}{dy}$$
```

SEE ALSO: `TeXer`

5.18.2 `TeXer` — produce TEX output

Calling Sequence:

`TeXer`(*expr*)

`TEX[TeXer]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `TeXer` function produces ouput suitable for TEX printing for the expression *expr*.
- This is the same function as `TEX[TeX]` without options.
- One may only use this function in a user defined function called within a call to the `TeX` function since the `TeX` function sets some variables !
- This is a function that does not reset the execution. It means that we do not start a new expression: it is very important to understand this since you have to use this function when you write your own `TeX/Userdefined`.../... functions. Indeed, in such functions you want to `TeX` a sub expression and not a new one. So, take care to use the appropriate function !
- Whenever there is a conflict between the function name `TeXer` and another name used in the same session, use the long form `TEX['TeXer']`.

Examples:

```
> with(TEX):
> TeXer(b);
Error, (in ERR[TEXPerror]) problem occured in procedure TeXer,
Trying to call TeXer() out of a TeX() call...
```

SEE ALSO: `TEX[advanced], TeX`

5.18.3 `TEX[advanced]` —- advanced TEX output

Synopsis:

- This help presents some technics and examples not to have too many problems when adding your own formatting functions.
- We first give the content of the tables used by the `TEX` package.

```
`TEX/special`:=table([Beta=\beta, GAMMA=\Gamma,
                      E=e, I=i, infinity =\infty, Pi=\pi,
                      Zeta=\zeta]):

`TEX/greek`:=table([alpha,         iota,          varrho,
                    beta,          kappa,         sigma,
                    gamma,         lambda,        varsigma,
                    delta,         mu,            tau,
                    epsilon,       nu,            upsilon,
                    varepsilon,    xi,            phi,
                    zeta,                         varphi,
                    eta,           pi,            chi,
                    theta,         varpi,         psi,
                    vartheta,      rho,           omega,
                    Gamma,         Xi,            Phi,
                    Delta,         Pi,            Psi,
                    Theta,         Sigma,         Omega,
                    Lambda,        Upsilon]):

`TEX/mathops`:=table([sin,         cos,          tan,
                      cot,         sec,          csc,
                      arcsin,      arccos,       arctan,
                      sinh,        cosh,         tanh,
                      coth,        liminf,       limsup,
                      arg,         deg,
                      dim,         hom,          inf,
                      ker,         lg,           Pr,
                      sup,         gcd]):
```

- Now, we present some internal functions that are default values of some variables used in the `TEX` package. These variables are detailed in the help `TEX[variables]`. It is just to show you how it is programmed so that it is not assumed that you should know all functions appearing in the following lines...

```
`TeXer/Identity`:=proc(i)
  i
end:

# This function is devoted to print something whenever there is an
# unassigned entry in an array.
#
`TeXer/UnknownEntryArray`:=proc()
```

```
  if (nargs=1) then
    `TeXer/cat`(`??_{`);
    `TeXer/tex`(args[1]);
    `TeXer/cat`(`}`)
  elif (nargs=2) then
    `TeXer/cat`(`??_{`);
    `TeXer/tex`(args[1]);
    `TeXer/cat`(`,`);
    `TeXer/tex`(args[2]);
    `TeXer/cat`(`}`)
  fi
end:

# This function is devoted to print something whenever there is an
# unassigned entry in a table.
#
`TeXer/UnknownEntryTable`:=proc()
  if (nargs=1) then
    `TeXer/cat`(`??_{`);
    `TeXer/tex`(args[1]);
    `TeXer/cat`(`}`)
  elif (nargs=2) then
    `TeXer/cat`(`??_{`);
    `TeXer/tex`(args[1]);
    `TeXer/cat`(`,`);
    `TeXer/tex`(args[2]);
    `TeXer/cat`(`}`)
  fi
end:

`TEX/OrderOnTableIndices`:=proc(l)
  sort(l)
end;
```

- The next function structure is very important if you want to make a new function to do the same job. The fact is that it has to treat the case of one argument from which it will extract the two lists of index for rows and columns, and also the case of two arguments so that it will rebuild the index to access to an element of the table.

```
`TeXer/IdentityTwoDim`:=proc(l)
  # we assume that l is the list of the indices of a table
  # obtained by [ indices(table) ]
  # the aim is to separate the index of rows and columns...
  # For this simple function, we assume that the table index
  # is made of a list of two elements that are the row index
  # and the column index.
  # THIS WORK MAY BE DONE WHEN ONLY ONE ARGUMENT IS GIVEN.
  # IN THE CASE OF TWO ARGUMENTS, THE FUNCTION MAY CONSTRUCT
  # THE INDEX FOR THE TABLE FROM A ROW AND A COLUMN INDEX.
```

```
  local
    sri, # set of rows indices...for a table with TWODIM
    sci, # set of columns indices...for a table with TWODIM
    n,   # nb indices...
    i;   # variable for loop...
  if (nargs=1) then
    n:=nops(l);
    sri:={};
    sci:={};
    for i from 1 to n do
      if (not member(op(1,op(op(i,l))),sri)) then
        sri:=sri union {op(1,op(op(i,l)))}
      fi;
      if (not member(op(2,op(op(i,l))),sci)) then
        sci:=sci union {op(2,op(op(i,l)))}
      fi
    od;
    sci:=(TEX_ORDER_ON_INDICES_TABLE_FUN)(convert(sci,list));
    sri:=(TEX_ORDER_ON_INDICES_TABLE_FUN)(convert(sri,list));
    RETURN([sri,sci])
  elif (nargs=2) then
    # this is the inverse operation that consists in rebuilding
    # the index from the row index and the column index...
    [args[1],args[2]]
  fi;
end:
```

- The following example shows how it is possible to define your own formatting functions for objects dealing with the symmetric group and the symmetric functions and Schubert polynomials.
- We first define a function called `SymmetricGroup`. When encountering a function call

`SymmetricGroup(4)`

`TeX` will produce `S_{4}` in TEX format.

```
# Symmetric group
`TeX/UserdefinedFun/SymmetricGroup`:=proc(n)
  texconcat(`S_{`);
  texer(n);
  texconcat(`}`)
end:
```

- Note that we use the `TeXer` function to call to `TeX` within a userdefined function.
- We may define then a function to treat expressions as `e5` or `e23` by

```
# Elementary functions...
`TeX/UserdefinedName/e`:=proc(i)
  texconcat(`\\Lambda_{`);
```

```
    texconcat(i);
    texconcat(`}`)
  end:
```

- Finally, you may define a function to treat Schubert polynomials that are indexed by permutations.

```
  # Schubert polynomials...indexed by permutations...
  `TeX/UserdefinedIndexed/X`:=proc()
    texconcat(`X_{`):
    texsequence(op(args)):
    texconcat(`}`)
  end:
```

5.18.4 TeXconcat — concat a string to the yet generated output

Calling Sequence:

TeXconcat(s)

TEX[TeXconcat](s)

Parameters:

s — any string

Synopsis:

- The TeXconcat concats a string s to the yet generated output. It is a useful function when defining your own formatting functions.
- One may only use this function in a user defined function called within a call to the TeX function since the TeX function sets some variables !
- Whenever there is a conflict between the function name TeXconcat and another name used in the same session, use the long form TEX['TeXconcat'].

Examples:

```
> with(TEX):
> TeXconcat(\\end );
Error, (in ERR[TEXPerror]) problem occured in procedure TeXconcat,
Trying to call TeXconcat() out of a TeX() call...
```

See Also: TEX[advanced], TeXstring

5.18.5 TEX[functions] — how TeX formats functions

Synopsis:

- When TeX processes a *Maple* object of type function that is in fact an unevaluated function call, it checks to see whether it exists a procedure which name looks like `TeX/UserdefinedFun/$< fun\_name >$` where $< fun\_name >$ is the name of the function. If such a procedure exists, it is used to format the function call. Since, it looks first for user defined functions, it provides a way to override the TeX internal formatting functions.

- Then, it checks to see whether it exists a procedure having the name ‘`TeX/`< *name* >‘. If so, it uses it to format the function call. This allows `TeX` to produce standard mathematical output in most cases.
- If such procedures do not exist, `TeX` formats the function call by recursively formatting the operands of the function call and inserting parentheses and commas at the appropriate points.
- The pre-defined functions are:

abs	det	factorial	binomial
@	@@	D	(diff)
(Diff)	int	Int	sum
Sum	exp	log	ln
log10	log2	limit	max
min	Limit	O	

- Moreover, the function `TeXsequence()` may be used to treat a sequence since the `TeX` command treats its first argument and see the others as options !!!

Examples:

```
> with(TEX):
> TeX(examples1);
{\\it examples1}
> TeX/UserdefinedName/examples1:=proc() TeXconcat(Name) end:
> TeX(examples1);
Name
> TeX/UserdefinedName/examples:=proc(x) TeXconcat(Generic);
                                           TeXconcat(x); end:
> TeX(examples1);
Name
> TeX(examples2);
Generic2
> TeX/UserdefinedFun/examples:=proc(x) TeXconcat(Fun);
                                           TeXconcat(x); end:
> TeX(examples1 + examples2 + examples(2));
Name+Generic2+Fun2
> TeX(examples1(2));
Name(2)
> TeX/UserdefinedFun/examples1:=proc(x) TeXconcat(Function);
                                          TeXconcat(x); end:
> TeX(examples1(2));
Function2
```

SEE ALSO: `TEX[advanced]`, `TeXconcat`, `TEX[indexed]`, `TEX[names]`, `TeXstring`, `TEX[variables]`

5.18.6 TEX[indexed] — how TeX formats indexed names

Synopsis:

- When TeX processes a *Maple* object of type indexed it makes various tests to see if special processing is desirable. This allows TeX to produce standard mathematical output in most situations.
- On encountering an indexed name, TeX first tests to see if the given name refers to a function `TeX/UserdefinedIndexed/` $<$ *name* $>$ `. If so, it uses this function to format the name, calling it with a list that is the index.
- If this function does not exist, it formats the name with its index on the right, a little bit under the name.

Examples:

```
> with(TEX):
> TeX(cosinus[1, 2, 3]);
{\\it cosinus}_{1, 2, 3}
> TeX(cos[1, 2, 3]);
\\cos _{1, 2, 3}
```

SEE ALSO: TEX[advanced], TEX[functions], TEX[names], TEX[variables]

5.18.7 TEX[names] — how TeX formats names

Synopsis:

- When TeX processes a *Maple* object of type string it makes various tests to see if special processing is desirable. This allows TeX to produce standard mathematical output in most situations.
- On encountering a string, TeX first tests to see if the given string refers to an entry in the table `TEX/special`. In such a case, that entry's value is used. For instance, infinity is converted in \infty.
- Next, TeX checks to see if the given string is contained in the table `TEX/greek`. In such a case, a backslash is prepended to its value. This allows names like "alpha" or "beta" to be printed as Greek symbols.
- Then, TeX checks to see if the given string is contained in the `TEX/mathops` table. If so, a backslash is prepended to its value. This allows function names like "sin" or "log" to be printed correctly by TeX.
- Finally, if the given string does not end by a number, then TeX looks for a function

  `TeX/UserdefinedName/` $<$ *the given string* $>$ `

  and calls it if it is assigned, while if it is not, it prints the given string. If the given string ends by a number then, TeX checks if a function whose name is

  `TeX/UserdefinedName/`*str*`

(where *str* is the given string without the numbers at the end) is assigned. If so, it calls this function with the extracted number as argument. For instance, if you want to see `h1, h2` as functions, define the function

```
`TeX/UserdefinedName/h`:=proc(n)
end;
```

so that when encountering either `h1` or `h2` for instance, the function `TeX` will simply call either `` `TeX/UserdefinedName/h`(1) `` or `` `TeX/UserdefinedName/h`(2) ``.

- The font used by `TeX` is controlled by the variable `TEX_FONT_NAME` which is initially set to `\it`. It is used for printing multi-character.
- The `TeX` command searches for special function formatting rules before searching for special name formatting rules. See `TEX[functions]` for more information and even `TEX[advanced]` for advanced use of `TeX`.

Examples:

```
> with(TEX):
> TeX(examples1);
{\\it examples1}
> TeX/UserdefinedName/examples1:=proc() TeXconcat(Name) end:
> TeX(examples1);
Name
> TeX/UserdefinedName/examples:=proc(x) TeXconcat(Generic);
                                        TeXconcat(x); end:
> TeX(examples1);
Name
> TeX(examples2);
Generic2
```

SEE ALSO: `TEX[advanced]`, `TEX[functions]`, `TEX[indexed]`, `TEX[variables]`

5.18.8 `TeXsequence` — `TeX` a sequence

Calling Sequence:

`TeXsequence`(s)

`TEX[TeXsequence]`(s)

Parameters:

s — any sequence

Synopsis:

- The `TeXsequence` function analyses a sequence of expressions by TeXing each one, and separating them by commas. It is a useful function when defining your own function.
- One may only use this function in a user defined function called within a call to the `TeX` function since the `TeX` function sets some variables !
- Whenever there is a conflict between the function name `TeXsequence` and another name used in the same session, use the long form `TEX['TeXsequence']`.

Examples:

```
> with(TEX):
> TeXsequence(a,b,c);
Error, (in ERR[TEXPerror]) problem occured in procedure TeXsequence,
Trying to call TeXsequence() out of a TeX() call...
```

SEE ALSO: `TEX[advanced], TeX, TeXer, TeXconcat`

5.18.9 `TeXstring` — dump a string

Calling Sequence:

`TeXstring`(s)

`TEX[TeXstring]`(s)

Parameters:

s — any string

Synopsis:

- The `TeXstring` function dumps a string, without using any buffer. The result is directly printed on the output so that it may be used every time.
- The problem is when using it inside your own functions, the results may be controlled to be sure that the output is done at the right time.
- Whenever there is a conflict between the function name `TeXstring` and another name used in the same session, use the long form `TEX['TeXstring']`.

Examples:

```
> with(TEX):
> TeXstring(\\end );
\\end
> TeXstring(a_b);
a_b
> TeX(a_b);
{\\it a\\_b}
```

SEE ALSO: `TEX[advanced], TeX, TeXer, TeXconcat`

5.18.10 `TEX[variables]` — how to control the `TeX` command

Synopsis:

- There are a lot of variables that may be modified to control the `TeX` *Maple* command. All these variables have default values so that the user who do not want to understand precisely how the `TEX` package commands work, has not to modify them. This help is to explain how the commands use the various variables, and how to modify the default values of such variables.
- Just remember that all names of variables begin with `TEX_`.

- `TEX_VERBOSE` : this variable indicates the verbose mode level.
  - Type : integer
  - 0 : only TEX expressions are dumped.
  - 1 : 0 + informations about the program.
  - 2 : 1 + informations about the author.
  - Default : 0
- `TEX_MODE` : this variable sets the way, the expressions are dumped.
  - Type : string
  - NONE : the expression is dumped without anything else.
  - MATHS : the expression is dumped between the TEX commands that invoke the math environment.
  - DISPLAY : the expression is dumped between the TEX commands that invoke the display style environment.
  - Default : NONE
- `TEX_OVER_MODE` : this variable specifies whether the TEX display mode may or not be used in a TEX expression as $\frac{numerator}{denominator}$ since in such expressions, the default TEX mode is simply the mathematical one.
  - Type : boolean
  - true : to set the display mode.
  - false : to unset the display mode.
  - Default : true
- `TEX_MATRIX_MODE` : this variable specifies whether the TEX display mode may or not be used in a matrix TEX expression since in such expressions, the default TEX mode is simply the mathematical one.
  - Type : boolean
  - true : to set the display mode.
  - false : to unset the display mode.
  - Default : true
- `TEX_INDEXED_COLUMNS` : to index or not matrix columns.
  - Type : string
  - NONE : no index on columns.
  - UP : an index is set over matrices columns.
  - Default : `NONE`
- `TEX_INDEXED_ROWS` : to index or not matrix rows.
  - Type : string
  - NONE : no index on rows.
  - LEFT : an index is set to the right of matrices rows.

  - Default : `NONE`

- `TEX_MAX_LENGTH` : whenever the length of the string created from a *Maple* expression is longer than the value of this variable, the string is cut (when it is possible), and the rest will appear from the next line. It is useful since text editors do not all accept too long lines...
  - Type : integer
  - Default : 60

- `TEX_UNKNOWN_ENTRY_ARRAY_FUN` : this variable specifies what has to be done whenever an array entry is not assigned.
  - Type : function
  - Default : `` `TeXer/UnknownEntryArray` `` that produces ?? indexed by the corresponding row and column.

- `TEX_UNKNOWN_ENTRY_TABLE_FUN` : this variable specifies what has to be done whenever a table entry is not assigned.
  - Type : function
  - Default : `` `TeXer/UnknownEntryTable` `` that produces ?? indexed by the corresponding row and column.

- `TEX_INDEX_ARRAY_FUN` : this variable is used to index arrays by special objects as for instance partitions, permutations, or simply integers...
  - Type : function
  - Default : `` `TeXer/Identity` `` that returns its argument !

- `TEX_TABLE_MODE` : this variable is used to control the way tables are dumped, that is how the command understand a table.
  - Type : string
  - `NONE` : the table is seen as a structure that contains various informations so that it is dumped as it is seen...
  - `ONEDIM` : the table is seen as a one dimensional array.
  - `TWODIM` : the table is seen as a two dimensional array, so that the result is a matrix indexed by a certain function of the table indices. This function is in fact specified by the `TEX_TWODIM_TABLE_FUN` variable.
  - Default : `TWODIM`

- `TEX_TWODIM_TABLE_FUN` : this variable is used to set the function whose goal is to extract from a table indice, two indices that will respectively correspond to the row and the column indices of a matrix.
  - Type : function
  - Default : `` `TeXer/IdentityTwoDim` `` that can be used when table indices are list of two arguments that are the row and the column.

- `TEX_ORDER_ON_INDICES_TABLE_FUN` : this variable sets the function that defines an order on the indices of a table. As a matter of fact, *Maple* has its own order on the entries of a table so that we have to reorder the entries.

  - Type : function
  - Default : `‘TeXer/OrderOnTableIndices‘` that sorts in increasing ordering a list of integers. This default value assume that the table is indexed by integers (list of 2 integers) so be careful when not using `TeX` with such a table.

- `TEX_INDEX_TABLE_FUN` : this variable is used to index tables by special objects as for instance partitions, permutations, or simply integers...
  - Type : function
  - Default : `‘TeXer/Identity‘` that returns its argument !

- `TEX_FONT_NAME` : this variable specifies the font for the names in mathematical expressions. This may be set to a TEX font.
  - Type : string
  - Default : `\it` that is the italic font.

- `TEX_CR` : this variable sets the carriage return that is used for instance in matrices,...
  - Type : string
  - Default : `\cr\cr` that means a quite big vertical skip !

- `TEX_OUTPUT` : this variable specifies the name of the file where the output will be done.
  - Type : string
  - Default : `NULL` to say that the output should be done on the terminal.

- `TEX_OUTPUT_MODE` : this variable specifies, the way the output is done.
  - `WRITE` : the file whose name is `TEX_OUTPUT` is erased whenever a new expression is dumped (by `TeX` command).
  - `APPEND` : the result is added at the end of the file whose name is specified by `TEX_OUTPUT`.
  - Default : `APPEND`

- `TEX_ORDERED` : this variable specifies whether `TeX` has the right to reorder the components of a sum (separating positive and negative ones).
  - `false` : provide the reordering.
  - `true` : forbid the reordering so that it provides a way to extract results in your own ordering.
  - Default : `true`

- `TEX_LEFT_RIGHT` : this variable specifies whether `TeX` should insert a `\left` and a `\right` command around the mathematical expressions.
  - `false` : no left and right delimiters.
  - `true` : add a left and right delimiter saying to TEX that it should adjust the size of the delimiters such as (), [], . . .
  - Default : false

- All these variables are global ones. The fact is that you may set some of them for all expressions, while you may also set some of them when TEXing only one expression. This can be done since the `TeX` command accepts as arguments (from the second argument, to . . .) expressions as `X=Y` were `X` is a name of variable and `Y` a value for this variable.

Examples:

```
> with(TEX):
> TeX(name);
{\\it name}
> TeX(name,'TEX_MODE'='MATHS','TEX_FONT_NAME'=\\bf );
${\\bf name} $
```

SEE ALSO: `TEX[advanced]`, `TEX[functions]`, `TEX[indexed]`, `TEX[names]`

5.19 Type checking package

Calling Sequence:

`TYP[<function>]`(*args*)

`<function>`(*args*)

Synopsis:

- This package provides functions to check that an object is well-formed.
- Whenever there is a conflict between a function name in `TYP` and another name used in the same session, use the long form `TYP[<function>]`.
- The available functions are:

IsBnBar	IsBnCode	IsBnPalin	IsCTab
IsCompo	IsO	IsPart	IsPe
IsPerm	IsSp	IsStdCTab	IsStdTab
IsTab	IsWord	IsX	IsXX
IsY	IsYY	IsYama	Iscl
Isc	Ise	Ish	Ism
Isp	Iss	Isx	Isy

- For help with a particular function do `?TYP[<function>]` or `?TYP,<function>` where `<function>` is one from the above list.

Examples:

```
> with(TYP):
> IsPart([3,3,2,1]);
                                true
> IsPerm([1,5,2,4,3]);
                                true
```

5.19.1 IsBnBar — test whether the argument is a signed permutation

Calling Sequence:

`IsBnBar`(*expr*)

`TYP[IsBnBar]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsBnBar` function tests whether *expr* is a signed permutation, that is a list of n distinct positive or negative integers, with distinct absolute values between 1 and n.

- Whenever there is a conflict between the function name `IsBnBar` and another name used in the same session, use the long form `TYP['IsBnBar']`.

Examples:

```
> with(TYP):
> IsBnBar([-2, 1, 2]);
                                    false
> IsBnBar([-2, 3, -1]);
                                    true
```

SEE ALSO: `IsBnCode, IsBnPalin`

5.19.2 IsBnCode — test whether the argument is a code

Calling Sequence:

`IsBnCode`(*expr*)

`TYP[IsBnCode]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsBnCode` function tests whether *expr* is a code in the sense of a B_n Coxeter group, that is a list of n non negative integers, lesser than $[1, 3, 5, 7, \ldots]$.

- Whenever there is a conflict between the function name `IsBnCode` and another name used in the same session, use the long form `TYP['IsBnCode']`.

Examples:

```
> with(TYP):
> IsBnCode([-2, 3, 4]);
```

false

```
> IsBnCode([1, 0, 5]);
```

true

SEE ALSO: `IsBnBar`, `IsBnPalin`

5.19.3 `IsBnPalin` — test whether the argument is a palindromic permutation ✘

Calling Sequence:

`IsBnPalin`(*expr*)

`TYP[IsBnPalin]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsBnPalin` function tests whether *expr* is a palindromic permutation of the symmetric group of degree $2n$, that is a permutation p for which $2n + 1 - p[2n + 1 - i] = p[i]$ for $i = 1, \ldots, n$, where n is $\texttt{nops}(p)/2$.
- Whenever there is a conflict between the function name `IsBnPalin` and another name used in the same session, use the long form `TYP['IsBnPalin']`.

Examples:

```
> with(TYP):
> IsBnPalin([3, 2, 6, 1, 5, 4]);
```

true

SEE ALSO: `IsBnBar`, `IsBnCode`

5.19.4 `IsO` — test whether the argument is an element of the orthogonal Schur basis ✘

Calling Sequence:

`IsO`(*expr*)

`TYP[IsO]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsO` function tests whether the argument *expr* is an element of the orthogonal Schur basis, that is `O`[*part*] where *part* denotes a partition in non-increasing order.
- Whenever there is a conflict between the function name `IsO` and another name used in the same session, use the long form `TYP['IsO']`.

Examples:

```
> with(TYP):
> IsO(O[2,1,1]);
```

```
true
```

SEE ALSO: `IsSp`

5.19.5 `IsSp` — test whether the argument is an element of the symplectic Schur basis

Calling Sequence:

`IsSp`(*expr*)

`TYP[IsSp]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsSp` function tests whether the argument *expr* is an element of the symplectic Schur basis, that is `Sp`[*part*] where *part* denotes a partition in non-increasing order.
- Whenever there is a conflict between the function name `IsSp` and another name used in the same session, use the long form `TYP['IsSp']`.

Examples:

```
> with(TYP):
> IsSp(Sp[2,1,1]);
```

```
true
```

SEE ALSO: `IsO`

5.19.6 `IsCompo` — test whether the argument is a composition

Calling Sequence:

`IsCompo`(*expr*)

`TYP[IsCompo]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsCompo` function tests whether *expr* is a composition that is a list of positive integers.
- Whenever there is a conflict between the function name `IsCompo` and another name used in the same session, use the long form `TYP['IsCompo']`.

Examples:

```
> with(TYP,IsCompo):
> IsCompo([1,2,1,3,2]);
```

```
true
```

SEE ALSO:

5.19.7 `IsPerm` — test whether the argument is a permutation

Calling Sequence:

`IsPerm`(*expr*)

`TYP[IsPerm]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsPerm` function tests whether *expr* is a permutation. A permutation of the symmetric group $\mathfrak{S}_n$ is a list of n distinct positive integers between 1 and n.
- Whenever there is a conflict between the function name `IsPerm` and another name used in the same session, use the long form `TYP['IsPerm']`.

Examples:

```
> with(TYP):
> IsPerm([3,1]);
```

```
false
```

```
> IsPerm([3,2,1]);
```

```
true
```

SEE ALSO: `Permutation`

5.19.8 How permutations are represented in the SG package

Synopsis:

- This part describes how the elements of the symmetric group of degree n ($n \geq 1$) are represented in the `SG` package.
- A permutation is denoted by the list of the images of i (for $i \in [1, n]$) under the permutation. In other words, a permutation *perm* is represented by the list $[perm(1), \ldots, perm(n)]$ where $perm(i)$ is the image of i under the permutation *perm*.

Examples:

```
> with(TYP):
                    # this is an element of the symmetric group of degree 3
> IsPerm([2,3,1]);

                                    true

                                # this is not a permutation of [1,2,3,4]
> IsPerm([2,4,1]);

                                   false
```

SEE ALSO: `IsPerm`

5.19.9 `IsPe` — test whether the argument is an element of the $\mathcal{Pe}$-basis

Calling Sequence:

`IsPe`(*expr*)

`TYP[IsPe]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsPe` function tests whether *expr* is an element of the basis of products of elementary symmetric functions on a flag of alphabets. It is $\mathcal{Pe}_I$ where I is a sequence of non-negative integers contained in the staircase $[n, n-1, \ldots, 2, 1]$ in which n is the length of *expr*.
- The coding is such that we do not add a 0 at the end of the I's since it is implicit.
- Whenever there is a conflict between the function name `IsPe` and another name used in the same session, use the long form `TYP['IsPe']`.

Examples:

```
> with(TYP):
> IsPe(Pe[3,0,1]);

                                    true
```

```
> IsPe(Pe[3,0,2]);
```

false

SEE ALSO:

5.19.10 IsX — test whether the argument is a Schubert polynomial

Calling Sequence:

`IsX`(*expr*)

`TYP[IsX]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsX` function tests whether *expr* is a Schubert polynomial indexed by a permutation.
- Whenever there is a conflict between the function name `IsX` and another name used in the same session, use the long form `TYP['IsX']`.

Examples:

```
> with(TYP):
> IsX(X[3,1]);
```

false

```
> IsX(X[3,1,2]);
```

true

SEE ALSO: `TYP[IsPerm], IsXX, IsY, IsYY`

5.19.11 IsXX — test whether the argument is a double Schubert polynomial

Calling Sequence:

`IsXX`(*expr*)

`TYP[IsXX]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsXX` function tests whether *expr* is a double Schubert polynomial indexed by a permutation.
- Whenever there is a conflict between the function name `IsXX` and another name used in the same session, use the long form `TYP['IsXX']`.

Examples:

```
> with(TYP):
> IsXX(XX[3,1]);
```

```
                    false
```

```
> IsXX(XX[3,1,2]);
```

```
                    true
```

SEE ALSO: `TYP[IsPerm]`, `IsX`, `IsY`, `IsYY`

5.19.12 `IsY` — test whether the argument is a Schubert polynomial

Calling Sequence:

`IsY`(*expr*)

`TYP[IsY]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsY` function tests whether *expr* is a Schubert polynomial indexed by a sequence of non-negative integers (interpreted as the code of a permutation).
- Whenever there is a conflict between the function name `IsY` and another name used in the same session, use the long form `TYP['IsY']`.

Examples:

```
> with(TYP):
> IsY(Y[0,1,2,0]);
```

```
                    true
```

```
> IsY(Y[0,-1]);
```

```
                    false
```

SEE ALSO: `IsX`, `IsXX`, `IsYY`

5.19.13 `IsYY` — test whether the argument is a Schubert polynomial

Calling Sequence:

`IsYY`(*expr*)

`TYP[IsYY]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsYY` function tests whether *expr* is a double Schubert polynomial indexed by a sequence of non-negative integers (interpreted as the code of a permutation).
- Whenever there is a conflict between the function name `IsYY` and another name used in the same session, use the long form `TYP['IsYY']`.

Examples:

```
> with(TYP):
> IsYY(YY[0,1,2,0]);
```

```
                    true
```

SEE ALSO: `IsX, IsXX, IsY`

5.19.14 `Isx` — test whether the argument is an x_i indeterminate

Calling Sequence:

`Isx`(*expr*)

`TYP[Isx]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `Isx` function tests whether *expr* is an x_i indeterminate where i denotes an integer.
- Whenever there is a conflict between the function name `Isx` and another name used in the same session, use the long form `TYP['Isx']`.

Examples:

```
> with(TYP):
> Isx(x3);
```

```
                    true
```

SEE ALSO: `Isy`

5.19.15 `Isy` — test whether the argument is an y_i indeterminate

Calling Sequence:

`Isy`(*expr*)

`TYP[Isy]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `Isy` function tests whether *expr* is an y_i indeterminate where i denotes an integer.
- Whenever there is a conflict between the function name `Isy` and another name used in the same session, use the long form `TYP['Isy']`.

Examples:

```
> with(TYP):
> Isy(y3);
                              true
```

SEE ALSO: `Isx`

5.19.16 How Schubert polynomials are represented in the SP package

Synopsis:

- This part describes how (double) Schubert polynomials are represented in the SP package.
- Schubert polynomials may be indexed by permutations or integer vectors. In the first case, they are denoted by the expression `X`[*permutation*] and in the second one, by `Y`[*code*] where *code* is a sequence of non-negative integers (interpreted as the code of a permutation).
- In the case of double Schubert polynomials we denote by `XX`[*permutation*] the double Schubert polynomial indexed by a permutation and by `YY`[*code*] the corresponding double Schubert polynomial indexed by a vector of non-negative integers (interpreted as the code of a permutation).

Examples:

```
> with(TYP):
> IsXX(XX[3,1,2,4]);

                              true
```

SEE ALSO: `Isx, Isy, IsX, IsXX, IsY, IsYY`

5.19.17 `IsPart` — test whether the argument is a partition

Calling Sequence:

`IsPart`(*expr*)

`TYP[IsPart]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsPart` function tests whether *expr* is a partition that is a list of positive integers in non-increasing order.

- Whenever there is a conflict between the function name `IsPart` and another name used in the same session, use the long form `TYP['IsPart']`.

Examples:

```
> with(TYP,IsPart):
> IsPart([1,2]);
```

false

```
> IsPart([3,2,2,1]);
```

true

SEE ALSO:

5.19.18 Isc — test whether the argument is an element of the cycle index basis

Calling Sequence:

`Isc`(*expr*)

`TYP[Isc]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `Isc` function tests whether the argument *expr* is an element of the cycle index basis, that is `c`[*part*] where *part* denotes a partition in non-increasing order.
- Each c_{part} is the cycle index of the conjugacy class indexed by *part*.
- Whenever there is a conflict between the function name `Isc` and another name used in the same session, use the long form `TYP['Isc']`.

Examples:

```
> with(TYP):
> Isc(c[2,1,1]);
```

true

SEE ALSO: `Ise, Ish, Ism, Isp, Iss, Sf`

5.19.19 Iscl — test whether the argument is like cl[*part*]

Calling Sequence:

`Iscl`(*expr*)

`TYP[Iscl]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `Iscl` function tests whether the argument *expr* is like `cl`[*part*] where *part* denotes a partition in non-increasing order.
- Whenever there is a conflict between the function name `Iscl` and another name used in the same session, use the long form `TYP['Iscl']`.

Examples:

```
> with(TYP):
> Iscl(cl[2,1,1]);
```

```
true
```

SEE ALSO: `IsPart`

5.19.20 `Ise` — test whether the argument is a generator of the e-basis

Calling Sequence:

`Ise`(*expr*)

`TYP[Ise]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `Ise` function tests whether the argument *expr* is a generator of the e-basis.
- Whenever there is a conflict between the function name `Ise` and another name used in the same session, use the long form `TYP['Ise']`.

Examples:

```
> with(TYP):
> Ise(e0);
```

```
true
```

```
> Ise(e12);
```

```
true
```

SEE ALSO: `Isc, Ish, Ism, Isp, Iss, Sf`

5.19.21 `Ish` — test whether the argument is a generator of the h-basis

Calling Sequence:

`Ish`(*expr*)

`TYP[Ish]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The Ish function tests whether the argument *expr* is a generator of the h-basis.
- Whenever there is a conflict between the function name Ish and another name used in the same session, use the long form TYP['Ish'].

Examples:

```
> with(TYP):
> Ish(h0);
```

true

```
> Ish(h12);
```

true

SEE ALSO: Isc, Ise, Ism, Isp, Iss, Sf

5.19.22 Ism — test whether the argument is an element of the monomial basis

✘

Calling Sequence:

Ism(*expr*)

TYP[Ism](*expr*)

Parameters:

expr — any expression

Synopsis:

- The Ism function tests whether the argument *expr* is an element of the monomial basis, that is m[*part*] where *part* denotes a partition in non-increasing order.
- Whenever there is a conflict between the function name Ism and another name used in the same session, use the long form TYP['Ism'].

Examples:

```
> with(TYP):
> Ism(m[2,1,1]);
```

true

SEE ALSO: Isc, Ise, Ish, Isp, Iss, Sf

5.19.23 Isp — test whether the argument is a generator of the p-basis

Calling Sequence:

Isp(*expr*)

TYP[Isp](*expr*)

Parameters:

expr — any expression

Synopsis:

- The `Isp` function tests whether the argument *expr* is a generator of the p-basis.

- Whenever there is a conflict between the function name `Isp` and another name used in the same session, use the long form `TYP['Isp']`.

Examples:

```
> with(TYP):
> Isp(p0);
```

true

```
> Isp(p12);
```

true

SEE ALSO: `Isc, Ise, Ish, Ism, Iss, Sf`

5.19.24 `Iss` — test whether the argument is an element of the Schur basis

Calling Sequence:

`Iss`(*expr*)

`TYP[Iss]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `Iss` function tests whether the argument *expr* is an element of the Schur basis, that is `s`[*part*] where *part* denotes a partition in non-increasing order.

- Whenever there is a conflict between the function name `Iss` and another name used in the same session, use the long form `TYP['Iss']`.

Examples:

```
> with(TYP):
> Iss(s[2,1,1]);
```

true

SEE ALSO: `Isc, Ise, Ish, Ism, Isp, Sf`

5.19.25 How contretableaux are represented in the TAB package

Synopsis:

- This part describes how contretableaux are represented in the `TAB` package.
- First of all, the alphabet is the set of the positive integers. The order on such an alphabet is the natural one, $1 < 2 < \cdots$
- A contretableau is represented as the list of its columns read from left to right and from top to bottom. Hence, lengths of these lists (columns) are in weakly increasing order. Moreover, each list (column) must be in decreasing order while if we read the contretableau row by row from left to right, values must be in weakly increasing order.
- $\begin{array}{ccc} 2 & 5 & 7 \\ 1 & 4 & 4 \\ & & 3 \end{array}$ is represented by the list `[[2,1], [5,4], [7,4,3]]`.
- The empty contretableau is denoted by the empty list.

Examples:

```
> with(TAB):
> ct:=[[2,1], [5,4], [7,4,3]]:
> CTab2Mat(ct);
```

```
                    [ 2  4  7 ]
                    [         ]
                    [ 1  3  6 ]
                    [         ]
                    [       5 ]
```

SEE ALSO: `TAB[CTab2Mat], IsCTab, Tableau`

5.19.26 `IsCTab` — test whether the argument is a contretableau

Calling Sequence:

`IsCTab`(*expr*)

`TYP[IsCTab]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsCTab` function tests whether *expr* is a contretableau that is a list of list(s) of positive integers with some conditions on the list(s), and on these integers.
- Whenever there is a conflict between the function name `IsCTab` and another name used in the same session, use the long form `TYP['IsCTab']`.

Examples:

```
> with(TAB):
> ct:=[[2], [3], [5,3], [5,4,2,1]]:
> IsCTab(ct);

                                   true

> CTab2Mat(ct);

                              [ 2  3  5  5 ]
                              [            ]
                              [       3  4 ]
                              [            ]
                              [          2 ]
                              [            ]
                              [          1 ]
```

SEE ALSO: `TAB[CTab2Mat]`, `CTableau`

5.19.27 `IsStdCTab` — test whether the argument is a standard contretableau

Calling Sequence:

`IsStdCTab`(*expr*)

`TYP[IsStdCTab]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsStdCTab` function tests whether *expr* is a contretableau that is contretableau filled with n distinct letters from 1 to n.
- Whenever there is a conflict between the function name `IsStdCTab` and another name used in the same session, use the long form `TYP['IsStdCTab']`.

Examples:

```
> with(TAB):
> ct:=[[1], [4,2], [6,5,3]]:
> IsStdCTab(ct);

                                  true

> CTab2Mat(ct);

                              [ 1  4  6 ]
                              [         ]
                              [    2  5 ]
                              [         ]
                              [       3 ]
```

SEE ALSO: `CTableau`, `IsCTab`

5.19.28 `IsStdTab` — test whether the argument is or not a standard tableau

Calling Sequence:

`IsStdTab`(t)

`TYP[IsStdTab]`(t)

Parameters:

t — any expression

Synopsis:

- The `IsStdTab` function tests whether *expr* is a tableau that is tableau filled with n distinct letters from 1 to n.
- Whenever there is a conflict between the function name `IsStdTab` and another name used in the same session, use the long form `TYP['IsStdTab']`.

Examples:

```
> with(TAB):
> t:=[[4], [3,6], [1,2,5]]:
> IsStdTab(t);

                                   true

> Tab2Mat(t);

                                [ 4       ]
                                [         ]
                                [ 3   6   ]
                                [         ]
                                [ 1   2   5 ]
```

SEE ALSO: `IsTab, Tableau`

5.19.29 `IsTab` — test whether the argument is a tableau

Calling Sequence:

`IsTab`(*expr*)

`TYP[IsTab]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsTab` function tests whether *expr* is a tableau that is a list of list(s) of positive integers with some conditions on the list(s) and on these integers.
- Whenever there is a conflict between the function name `IsTab` and another name used in the same session, use the long form `TYP['IsTab']`.

Examples:

```
> with(TAB):
> t:=[[5], [3,6], [1,2,7,8]]:
> IsTab(t);

                                  true

> Tab2Mat(t);

                            [ 5          ]
                            [            ]
                            [ 3  6       ]
                            [            ]
                            [ 1  2  7  8 ]
```

SEE ALSO: `TAB[CTab2Mat]`, `Tableau`

5.19.30 `IsWord` — test whether the argument is a word

Calling Sequence:

`IsWord`(*expr*)

`TYP[IsWord]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsWord` function tests whether *expr* is a word on the alphabet 1, 2, and so on. A word is a list of positive integers.
- Whenever there is a conflict between the function name `IsWord` and another name used in the same session, use the long form `TYP['IsWord']`.

Examples:

```
> with(TYP):
> IsWord([1,4,1,2,2,2,1]);

                                  true
```

SEE ALSO:

5.19.31 `IsYama` — test whether the argument is a Yamanouchi word

Calling Sequence:

`IsYama`(*expr*)

`TYP[IsYama]`(*expr*)

Parameters:

expr — any expression

Synopsis:

- The `IsYama` function tests whether *expr* is a Yamanouchi word on the alphabet 1, 2, and so on.
- We recall that a word is said to be a Yamanouchi one if and only if all its right subwords are such that $\#1 \geq \#2 \geq \#3 \geq \cdots$ where $\#i$ denotes the number of i's.
- Whenever there is a conflict between the function name `IsYama` and another name used in the same session, use the long form `TYP['IsYama']`.

Examples:

```
> with(TYP):
> IsYama([2,1]);

                              true

> IsYama([1,2]);

                              false

> IsYama([4,3,4,4,3,3,2,1,2,2,1,1]);

                              true
```

See Also: `IsWord`

5.19.32 How symmetric functions are represented in the SYMF package

Synopsis:

- This part describes how symmetric functions are represented in the `SYMF` package.
- Symmetric function names are compatible with Macdonald's conventions, as well as the conventions used in the `SF` package of Stembridge.
- First of all, one may consider two types of bases: multiplicative bases and non-multiplicative ones.
- The bases declared by default are the following:

  `e` : multiplicative basis of elementary symmetric functions. The generators of such a basis are `e0, e1, e2, ...`, and the elements of the basis are the monomials in these variables.

  `h` : multiplicative basis of complete homogeneous symmetric functions. The generators of such a basis are `h0, h1, h2, ...`, and the elements of the basis are the monomials in these variables.

  `m` : non-multiplicative basis of monomial symmetric functions. The elements of such a basis are indexed by partitions. For instance, `m[3,3,1]` and `m[]` are elements of the m-basis.

  `p` : multiplicative basis of power sum symmetric functions. The generators of such a basis are `p0, p1, p2, ...`, and the elements of the basis are the monomials in these variables.

  `s` : non-multiplicative basis of Schur functions. The elements of such a basis are indexed by partitions. For instance, `s[3,3,1]` and `s[]` are elements of the s-basis.

  `c` : non-multiplicative basis of cycle indices. The elements of such a basis are indexed by partitions. For instance, `c[3,3,1]` and `c[]` are elements of the c-basis.

Examples:

```
  # h3 is a generator of the basis of complete symmetric functions
> Ish(h3);

                                    true

> Iss(s[2,1]);

                                    true
```

SEE ALSO: `Ise, Ish, Ism, Isp, Iss, IsPart`

5.19.33 How tableaux are represented in the `TAB` package

Synopsis:

- This part describes how tableaux are represented in the `TAB` package.
- First of all, the alphabet is the set of the positive integers. The order on such an alphabet is the natural one, $1 < 2 < \cdots$
- A tableau is represented as the list of its rows read from top to bottom and from left to right. Hence, lengths of the lists (rows) are in weakly increasing order. Furthermore, each list (row) must be in weakly increasing order while if we read the tableau column by column from bottom to top, values must be in increasing order.
- $\begin{matrix} 4 & & & \\ 3 & 4 & & \\ 1 & 1 & 5 & 7 \end{matrix}$ is represented by the list `[[4], [3,4], [1,1,5,7]]`.
- The empty tableau is denoted by the empty list.

Examples:

```
> with(TAB):
> t:=[[4], [3,4], [1,1,5,7]]:
> Tab2Mat(t);

                              [ 4          ]
                              [            ]
                              [ 3  4       ]
                              [            ]
                              [ 1  1  5  7 ]
```

SEE ALSO: `CTableau, IsTab, TAB[Tab2Mat]`

5.20 Error printing package ✓

Calling Sequence:

ERR[<function>]($args$)

<function>($args$)

Synopsis:

- This package provides some functions to print a quite explicit error message whenever there is a problem. There is one function by package, member of the global environment ACE.
- Whenever there is a conflict between a function name in ERR and another name used in the same session, use the long form ERR[<function>].
- The available functions are:

ACEPerror	BNPerror	BNAPerror	CLGPerror
CGPerror	FMPerror	FOCKPerror	HEKAPerror
IDCAPerror	NCAPerror	NCSFPerror	SGAPerror
SGPerror	SHRIPerror	SPPerror	SYMFPerror
TABPerror	TEXPerror		

6

Bugs report

Etre grand, c'est aujourd'hui (sauf en mathématique pure et en poésie)
être à la tête d'une équipe dont les préoccupations sont tournées vers l'avenir.

Louis Armand.

We are proud of the fact that ACE is supported from people outside our university. There are several people who helped us by pointing out bugs. In particular, Alain Lascoux has made a major contribution to the source code by offering bug reports and also bug fixes. We give some[(1)] of the fixed bugs for information:

```
✂.....................................................................................
date: 19.06.95
file: SOURCES/SGA/src/SgaOnX
ACE> SgaOnX([3,2,1], X[2,1]);
bug:

                                    X[2, 1]

fixed:

                          - X[1, 3, 2] + X[1, 2, 4, 3]

####################################################################################
date: 22.06.95
file: SOURCES/GL/help/GL_n
ACE> ?GL,GL_n
bug:
          No exact match;  displaying help for   GL

fixed:
```

[(1)]Note that some functions involved in this bugs report have their names changed or have changed of package, in ACE 2.0. Moreover, some informations may be out of date.

```
gives the right help.

###########################################################################################
date: 06.07.95
file: SOURCES/HEQA/src/TrepT
ACE> HeqaRepT([1,2,4,3],[6]);
bug:

                                     [ -1 ]

fixed:

                                     [ q ]

ACE> HeqaRepT([1,2,4,3],[1,1,1,1,1]);
bug:

                                     [ q ]

fixed:

                                     [ -1 ]

###########################################################################################
date: 06.07.95
file: SOURCES/HEQA/src/TrepCT
ACE> HeqaRepCT([1,2,4,3],[6]);
bug:

                                     [ -1 ]

fixed:

                                     [ q ]

ACE> HeqaRepCT([1,2,4,3],[1,1,1,1,1]);
bug:

                                     [ q ]

fixed:

                                     [ -1 ]
###########################################################################################
date: 06.12.95
file: SOURCES/GL/src/x2m
ACE> x2m_n(x1^2*x2 + x1*x2^2);
bug:

                                      0

fixed:

                                    m[2, 1]
```

```
###########################################################################################
date: 08.12.95
file: SOURCES/SP/src/x2XX
ACE> x2XX(XX2x(XX[2,3,1]));
bug:

   - y2 XX[2, 1] + XX[2, 3, 1] + y1 XX[1, 3, 2] + y2 (XX[2, 1] + y1 XX[1])

        - y1 (- XX[2, 1] + XX[1, 3, 2] + y2 XX[1]) - y1 (XX[2, 1] + y1 XX[1])

            2
        + y1  XX[1]

fixed:

                                    XX[2, 3, 1]

###########################################################################################
date: 12.12.95
file: SOURCES/SP/help/SP
ACE> ?SP
bug:

     Flag              LRrbyX            MaxTr             Monk
     NewtonInterp      Rt                Specialize        TableX
     TableXX           TableY            TableYY           Tox
     ToX               ToY               X2Y               X2x
     XX2YY             XX2x              Y2X               Y2x
     YY2XX             YY2x              x2X               x2XX

fixed:

     Flag              LRrbyX            MaxTr             Monk
     NewtonInterp      Rt                ScalarPol         Specialize
     TableX            TableXX           TableY            TableYY
     Tox               ToX               ToY               X2Y
     X2x               XX2YY             XX2x              Y2X
     Y2x               YY2XX             YY2x              x2X
     x2XX

###########################################################################################
date: 12.12.95
file: SOURCES/SYMF/help/SYMF
ACE> ?SYMF
bug:

     Char2Sf           EvalSf            Hooks             ITensor
     JtMatrix          ListPart          ListPartIn        LRrSf
     Omega             MuirSf            PieriSf           Plethysm
     RedressSchur      Sf2Char           ScalarSf          Theta
     Toe               Toh               Top               Tos
     e2h               e2p               e2s               h2e
     h2p               h2s               p2e               p2h
     p2s               s2e               s2h               s2p
```

```
     Zee

fixed:

     Char2Sf            Conjugate          EvalSf             Hooks
     ITensor            JtMatrix           ListPart           ListPartIn
     LRrSf              Omega              MuirSf             PieriSf
     Plethysm           RedressSchur       ScalarSf           Sf2Char
     Theta              Toe                Toh                Top
     Tos                e2h                e2p                e2s
     h2e                h2p                h2s                p2e
     p2h                p2s                s2e                s2h
     s2p                Zee

###########################################################################################
date: 20.12.95
file: SOURCES/SYMF/src/Toe
file: SOURCES/SYMF/src/Toh
file: SOURCES/SYMF/src/Top
ACE> Top((1-q)^2*s[2,1], noexpand);
bug:

              3         3            3  2                              2
        1/3 p1  - 2/3 p1  q + 1/3 p1  q  - 1/3 p3 + 2/3 p3 q - 1/3 p3 q

fixed:

                                   2        3
                             (1 - q)  (1/3 p1  - 1/3 p3)

###########################################################################################
date: 02.01.96
file: SOURCES/GL/src/Monomial
file: SOURCES/GL/src/Toe
ACE> GL_n(3):
ACE> Toh_n(h4);
bug:
                                       0

fixed:

                                       h4

###########################################################################################
date: 02.01.96
file: SOURCES/GL/src/JtMatrix
ACE> GL_n(5):
ACE> `GL/JtMatrix`([2,1,1,1,1]);
bug:
                              [ h2  h3  h4  h5   0 ]
                              [                    ]
                              [  1  h1  h2  h3  h4 ]
                              [                    ]
                              [  0   1  h1  h2  h3 ]
                              [                    ]
```

```
                          [ 0   0   1  h1  h2 ]
                          [                   ]
                          [ 0   0   0   1  h1 ]

fixed:
                          [ h2  h3  h4  h5  h6 ]
                          [                    ]
                          [  1  h1  h2  h3  h4 ]
                          [                    ]
                          [  0   1  h1  h2  h3 ]
                          [                    ]
                          [  0   0   1  h1  h2 ]
                          [                    ]
                          [  0   0   0   1  h1 ]

###########################################################################################
date: 02.01.96
file: SOURCES/GL/src/PieriSf
ACE> GL_n(5):
ACE> PieriSf_n(h6, s[3,2,2,1]);
bug:
                                        0

fixed:

    s[6, 3, 2, 2, 1] + s[7, 2, 2, 2, 1] + s[7, 3, 2, 1, 1] + s[7, 3, 2, 2]

         + s[8, 2, 2, 1, 1] + s[8, 2, 2, 2] + s[8, 3, 2, 1] + s[9, 2, 2, 1]

###########################################################################################
date: 02.01.96
file: SOURCES/GL/src/Toe
file: SOURCES/GL/src/Toh
file: SOURCES/GL/src/Top
ACE> Top_n((1-q)^2*s[2,1], noexpand);
bug:

              3         3           3  2                                2
        1/3 p1  - 2/3 p1  q + 1/3 p1  q  - 1/3 p3 + 2/3 p3 q - 1/3 p3 q


fixed:

                                  2        3
                           (1 - q)  (1/3 p1  - 1/3 p3)

###########################################################################################
date: 04.01.96
file: PACKAGES/SG/SGhlp
ACE> ?SG,IsVexillar
bug:
No exact match;  displaying help for   SG

fixed:
```

```
ok

###########################################################################################
date: 04.01.96
file: PACKAGES/SG/SGhlp
ACE> ?SG,RandGrassmannian
bug:
No exact match;  displaying help for   SG

fixed:

ok

###########################################################################################
date: 04.01.96
file: SOURCES/SYMF/help/Hooks
ACE> ?SYMF,Hooks
bug:
No exact match;  displaying help for   SYMF

fixed:

ok

###########################################################################################
date: 04.01.96
file: SOURCES/SYMF/help/ListPart
ACE> ?SYMF,ListPart
bug:
No exact match;  displaying help for   SYMF

fixed:

ok

###########################################################################################
date: 05.01.96
file: SOURCES/GL/src/x2m
ACE> x2m now works and depends on _GLn i.e. it assumes that the input
     symmetric function is symmetric in the variables x1, ..., x.(_GLn).

###########################################################################################
date: 05.02.96
file: SOURCES/SP/src/Develop
ACE> x2X(x1^5 + x1*x2^3*x3);

     X[2, 5, 3, 1, 4] - X[3, 4, 2, 1] - X[4, 2, 3, 1] + X[6, 1, 2, 3, 4, 5]

bug:
ACE> X2x(");

                  2           2          3          2   2            3     5
       x3 (x2 (x2  x1 + x2 x1 ) + x2 x1 ) - x3 x2  x1  - x3 x2 x1  + x1
```

```
fixed:
ACE> X2x(");

                                     5         3
                                   x1  + x1 x2  x3

############################################################################################
date: 05.02.96
file: SOURCES/SP/help/MaxTr
ACE> MaxTr([3,1,5,2,4]);

bug:

    X[3, 1, 5, 2, 4] = x3 X[3, 1, 4, 2] + X[3, 4, 1, 2] + X[4, 1, 3, 2]

fixed:

  X[3, 1, 5, 2, 4] = x3 X[3, 1, 4, 2, 5] + X[3, 4, 1, 2, 5] + X[4, 1, 3, 2, 5]

############################################################################################
date: 06.02.96
file: SOURCES/SP/src/XX2x
ACE> p:=x1^2: NewtonInterp(p, 2):  XX2x(");

bug:
                                2
                              y1  + (y1 + y2) (x1 - y1)

fixed:

                                y1 x1 + y2 x1 - y2 y1

###########################################################################################
date: 22.05.96
file: SOURCES/SG/help/Rd2Perm
file: SOURCES/SG/src/Rd2Perm
ACE> Rd2Perm([1,2,1], 2);

bug:

                                       [3, 2]

fixed:
Error, (in ERR[SGPerror]) problem occured in procedure Rd2Perm,
Wrong range for second argument...
```

7

About ACE

ACE is an environment for computations in algebraic combinatorics. It has been developped by several persons and is distributed for several systems. This chapter clusters some informations about `ACE`. The first table gives for each author, the corresponding package(s), whereas the second table presents technical informations about `ACE`.

Author(s)	**Package(s)**	**Email**
Renaud Eppstein	`CG`	`eppstein@univ-mlv.fr`
Éric Laugerotte	`SHRI`	`laugerot@lir.dir.univ-rouen.fr`
Bernard Leclerc and Jean-Yves Thibon	`FOCK`	`bl@litp.ibp.fr`, `jyt@univ-mlv.fr`
Bun-Chan-Vorac Ung	`NCSF`	`ung@univ-mlv.fr`

Sébastien Veigneau	`ACE`
	`SG, BN, SGA, NCA, IDCA, HEKA, BNA, SYMF, CLG, TAB, SP, FM`
	`ERR, TYP, TEX`
		`sv@univ-mlv.fr`

Packages in `ACE` 2.0, by author.

Project Name	`ACE`	Algebraic Combinatorics Environment.
Version	2.0 (1996)	Available since July 1996.
Main contact	Sébastien Veigneau	*Email*: `sv@univ-mlv.fr`
Anonymous Ftp	`weyl.univ-mlv.fr`	directory `pub/lang/maple`
Available for	Maple V	Release 1, 2 and 3.
Systems	UNIX, MS-DOS and MACINTOSH	From MS-DOS 3.3.
Number of lines	about 50000 lines for sources and 20000 lines for on-line help	7000 lines for installation programs
Ms-Dos, Maple V Release 1	`ACE1DOS.tgz, DAT1DOS.tgz, DOC1DOS.tgz`	gziped `tar` files.
Ms-Dos, Maple V Release 2	`ACE2DOS.tgz, DAT2DOS.tgz, DOC2DOS.tgz`	gziped `tar` files.
Ms-Dos, Maple V Release 3	`ACE3DOS.tgz, DAT3DOS.tgz, DOC3DOS.tgz`	gziped `tar` files.
Unix, Maple V Release 1	`ACE1UNIX.tgz, DAT1UNIX.tgz, DOC1UNIX.tgz`	gziped `tar` files.
Unix, Maple V Release 2	`ACE2UNIX.tgz, DAT2UNIX.tgz, DOC2UNIX.tgz`	gziped `tar` files.
Unix, Maple V Release 3	`ACE3UNIX.tgz, DAT3UNIX.tgz, DOC3UNIX.tgz`	gziped `tar` files.
Macintosh, Maple V Release 1	`ACE1MAC.sit.hqx, ACE+DAT1MAC.sit.hqx, DOC1MAC.sit.hqx`	BinHexed archive files.
Macintosh, Maple V Release 2	`ACE2MAC.sit.hqx, ACE+DAT2MAC.sit.hqx, DOC2MAC.sit.hqx`	BinHexed archive files.
Macintosh, Maple V Release 3	`ACE3MAC.sit.hqx, ACE+DAT3MAC.sit.hqx, DOC3MAC.sit.hqx`	BinHexed archive files.
Postscript documentation	`manual.ps`	500 pages, 3Mb
Text documentation	`*.txt`	464 printable files

Bibliography

Le nombre de livres écrits sur l'électron en cinquante ans
est sans doute plus grand que le nombre de livres écrits sur la
lune en cinq cents ans. Exister par le livre, c'est déjà une existence,
une existence si humaine, si solidement humaine!

L'activité rationaliste de la physique contemporaine, Bachelard.

[1] G.Duchamp, D.Krob, A.Lascoux, B.Leclerc, T.Scharf and J.-Y.Thibon, *Euler-Poincare characteristic and polynomial representations of Iwahori-Hecke algebras*, Publications of the Research Institute for Mathematical Sciences (Kyoto), **31**, (2), 179–201, 1995.

[2] A.Kerber, *Algebraic combinatorics via finite group actions*, Mannheim:BI-Wissenschaftsverlag, 1991.

[3] A.Kerber, A.Kohnert and A.Lascoux, *SYMMETRICA, an object oriented computer algebra system for the symmetric group*, J. Symbolic Comp. **14** (1992), 195–203.

[4] J.-Y.Thibon and S.Veigneau, *Using ACE, an Algebraic Combinatorics Environment for MAPLE*, Proceedings of the 8-*th* Conference "Formal Power Series and Algebraic Combinatorics", University of Minnesota, 1996.

[5] B.C.V.Ung and S.Veigneau, *ACE, un environnement de calcul en Combinatoire Algébrique*, Proceedings of the 7-*th* Conference "Formal Power Series and Algebraic Combinatorics", Université de Marne-la-Vallée 1995, 557–562.

[6] S.Veigneau, *ACE, an Algebraic Combinatorics Environment for the computer algebra system MAPLE*, *User's Reference Manual*, *Version 1.0*, Publ. IGM **95-14**, Université de Marne-la-Vallée, 1995.

Characters of classical groups

[7] G.R.E.Black, R.C.King and B.G.Wybourne, *Kronecker products for compact semisimple Lie groups*, J. Phys. A: Math. Gen., 1983.

[8] H.Georgi, *Lie algebras in particle physics*, Benjamin, Reading Mass., 1982.

[9] R.C.King, *S-functions and characters of Lie algebras and Lie groups*, Topics in Algebra, Banach Center Publication, **26**, PWN, Warsaw, 1990.

[10] R.C.King, *Modification rules and products of irreducible representations of the unitary, orthogonal and symplectic groups*, J. Math. Phys. **12**, 1971, 1588–1598.

[11] K.Koike and I.Terada, *Young-diagrammatic methods for the representation theory of the classical group of type B_n, C_n, D_n*, J. Algebra **107**, 1987, 466–511.

[12] C.Kostka, *Über den Zusammenhang zwischen einingen Formen von symmetrischen Funktionen*, Crelle's Journal **93**, 89–123.

[13] D.E.Littlewood, *On certain symmetric functions*, J. London. Math. Soc. **11**, 1961, 485–498.

[14] D.E.Littlewood, *Products and plethysms of characters with orthogonal, symplectic and symmetric groups*, J. Can. Math. **10**, 1958, 17–32.

[15] D.E.Littlewood and A.R.Richardson, *Group characters and algebra*, Phil. Trans. R. Soc. **233**, 99–141.

[16] G.Lusztig, *Singularities, character formulas and a q-analog of weight multiplicities*, Analyse et topologie des espaces singuliers (II-III), Astérisque **101–102**, 1983.

[17] R.V.Moody and J.Patera, *Fast recursion formula for weight multiplicities*, Bull. American Math. Soc. **7**, 237–242.

[18] D.H.Sattinger and O.L.Weaver, *Lie groups and algebras with applications to physics, geometry and mechanics*, Springer-Verlag **61**, New York, 1986.

[19] T.Scharf and J.-Y.Thibon, *A Hopf-algebra approach to inner plethysm*, Advances in Math. **104**, 1994.

[20] J.-P.Serre, *Complex semisimple Lie algebras*, Springer-Verlag, New York, 1987.

[21] D.Stanton, *Invariant theory and tableaux*, Springer-Verlag, New York, 1986.

[22] H.Weyl, *Classical groups*, Princeton University Press, Princeton, NJ, 1939.

[23] B.G.Wybourne, *Symmetry principles and atomic spectroscopy*, Wiley, 1969.

[24] B.G.Wybourne, *SCHUR an interactive programme for calculating properties of Lie groups*.

Fock space representation of $U_q(\widehat{sl}_n)$

[25] S.Ariki, *On the decomposition numbers of the Hecke algebra of $G(m,1,n)$*, preprint 1996.

[26] I.Grojnowski, *Affine Hecke algebras (and affine quantum GL_n) at roots of unity*, Internat. Res. Math. Notices **5** (1994), 215–217.

[27] T.Hayashi, *q-analogues of Clifford and Weyl algebras – spinor and oscillator representations of quantum enveloping algebras*, Commun. Math. Phys., **127** (1990), 129–144.

[28] G.James, *The decomposition matrices of $GL_n(q)$ for $n \leq 10$*, Proc. London Math. Soc., **60** (1990), 225–265.

[29] G.James, A.Kerber, *The representation theory of the symmetric group*, Addison-Wesley, 1981.

[30] M.Kashiwara, *On crystal bases of the q-analogue of universal enveloping algebras*, Duke Math. J. **63** (1991), 465–516.

[31] M.Kashiwara, *Crystal bases of modified quantized enveloping algebra*, Duke Math. J. **73** (1994), 383–413.

[32] M.Kashiwara, T.Miwa, E.Stern, *Decomposition of q-deformed Fock spaces*, preprint RIMS-1035 (1995), Selecta Mathematica (New series), (1996).

[33] A.S.Kleshchev, *Branching rules for the modular representations of symmetric groups III; some corollaries and a problem of Mullineux*, J. London Math. Soc., **2** (1995).

[34] A.Lascoux, B.Leclerc, J.-Y. Thibon, *Hecke algebras at roots of unity and crystal bases of quantum affine algebras*, Commun. Math. Phys. (to appear).

[35] A.Lascoux, B.Leclerc, J.-Y.Thibon, *Ribbon tableaux, Hall-Littlewood functions, quantum affine algebras and unipotent varieties*, preprint, 1995.

[36] B.Leclerc, J.-Y.Thibon, *Canonical bases of q-deformed Fock spaces*, preprint q-alg/9602025, Internat. Res. Math. Notices, (to appear).

[37] K.C.Misra, T.Miwa, *Crystal base of the basic representation of* $U_q(\widehat{sl}_n)$, Commun. Math. Phys., **134** (1990), 79–88.

Noncommutative symmetric functions

[38] F.Bergeron and D.Krob, *Acyclic complexes related to noncommutative symmetric functions*, preprint, 1995, to appear in J. Alg. Comb.

[39] R.W.Carter, *Representation theory of the 0-Hecke algebra*, J. Algebra, **15**, (1986), 89–103.

[40] G.Duchamp, A.Klyachko, D.Krob and J.-Y.Thibon, *Noncommutative symmetric functions III: Deformations of Cauchy and convolution algebras*, Publ. IGM **96-1**, Université de Marne-la-Vallée, 1996.

[41] G.Duchamp, D.Krob, B.Leclerc and J.-Y.Thibon, *Déformations de projecteurs de Lie*, C.R. Acad. Sci. Paris **319** (1994), 909–914.

[42] G.Duchamp, D.Krob, B.Leclerc and J.-Y.Thibon, *Fonctions quasi-symétriques, fonctions symétriques non commutatives et algèbre de Hecke à q=0*, to appear in C.R. Acad. Sci. Paris.

[43] A.M.Garsia and C.Reutenauer, *A decomposition of Solomon's descent algebra*, Adv. in Math, **77** (1989), 189–262.

[44] I.M.Gelfand, D.Krob, A.Lascoux, B.Leclerc, V.S.Retakh and J.-Y.Thibon, *Noncommutative symmetric functions*, Adv. in Math. **112** (1995), 218–348.

[45] I.M.Gelfand and V.S.Retakh, *Determinants of matrices over noncommutative rings*, Funct. Anal. Appl. **25**, (1991), 91–102.

[46] I.M.Gelfand and V.S.Retakh, *A theory of noncommutative determinants and characteristic functions of graphs*, Funct. Anal. Appl. **26** (1992), 1–20; Publ. LACIM, UQAM, Montreal, **14**, 1–26.

[47] I.Gessel, *Multipartite P-partitions and inner product of skew Schur functions*, Contemporary Math. **34** (1984), 289–301.

[48] A.Klyachko, *Lie elements in the tensor algebra*, Siberian Math. J. **15** (1974), 1296–1304.

[49] D.Krob, B.Leclerc and J.-Y.Thibon, *Noncommutative symmetric functions II: Transformations of alphabets*, Publ. IGM **95-5**, Université de Marne-la-Vallée, 1995.

[50] D.Krob and J.-Y.Thibon, *Noncommutative symmetric functions IV*, preprint LITP, Université Paris 7, 1996.

[51] B.Leclerc, T.Scharf and J.-Y.Thibon, *Noncommutative cyclic characters of symmetric groups*, preprint, 1994, to appear in J. Comb. Theory A.

[52] P.A.MacMahon, *Combinatory analysis*, Cambridge University Press, 1915, 1916; Chelsea reprint, 1960.

[53] C. Malvenuto, *Produits et coproduits des fonctions quasi-symmétriques et de l'algèbre des descentes (Thesis)*, LACIM, UQAM, Montréal, 1993.

[54] C.Malvenuto and C.Reutenauer, *Duality between quasi-symmetric functions and the Solomon descent algebra*, preprint UQAM, Montréal, 1993.

[55] B.Mielnik and J.Plebański, *Combinatorial approach to Baker-Campbell-Hausdorff exponents*, Ann. Inst. Henri Poincaré, Section A vol. **XII** (1970), 215–254.

[56] C.Reutenauer, *Free Lie algebras*, Oxford University Press, 1993.

[57] T.Scharf and J.-Y. Thibon, *On Witt vectors and symmetric functions*, to appear in Algebra Colloquium.

[58] L.Solomon, *A Mackey formula in the group ring of a Coxeter group*, J. Algebra **41** (1976), 255–268.

Schubert polynomials

[59] I.N.Bernstein, I.M.Gelfand, S.I.Gelfand, *Schubert cells and cohomology of the spaces G/P*, Uspekhi.Mat.Nauk, **28** (1973), 3–26.

[60] I.V.Cherednik, *A new interpretation of Gelfand-Tzetlin bases*, Duke. Math. J. **54** (1987), 563–577.

[61] M.Demazure, *Désingularisation des variétés de Schubert*, Annales E.N.S. **6** (1974), 163–172.

[62] S.Fomin and A.N.Kirillov, *Combinatorial B_n-analogues of Schubert polynomials*, preprint, 1994.

[63] W.Fulton, *Flags, Schubert polynomials, degeneracy loci and determinantal formulas*, Duke Math.J. **65** (1991), 381-420.

[64] F.Hirzebruch, *Topological Methods in Algebraic Geometry*, 3rd ed., Springer-Verlag, 1978.

[65] A.Kohnert, *Weintrauben, Polynome, Tableaux*, Bayreuther Math. Schr. **38** (1991), 1–97.

[66] A.Kohnert and S.Veigneau, *Using Schubert toolkit to compute with polynomials in several variables*, Proceedings of the 8-*th* Conference "Formal Power Series and Algebraic Combinatorics", Minneapolis.

[67] A.Lascoux, *Polynômes de Schubert. Une approche historique*, Actes du 4^e colloque "Séries formelles et combinatoire algébrique", Montréal 1992, Publ. LACIM, UQAM, 1992.

[68] A.Lascoux, B.Leclerc and J.-Y.Thibon, Flag Varieties and the Yang-Baxter Equation, preprint, 1995.

[69] A.Lascoux, B.Leclerc and J.-Y.Thibon, *Twisted action of the symmetric group on the cohomology of a flag manifold*, Banach Center Publ. (to appear).

[70] A.Lascoux and M.P.Schützenberger, *Schubert polynomials and the Littlewood-Richardson Rule*, Lett. Math. Phys. **10** (1985), 111–124.

[71] A.Lascoux and M.P.Schützenberger, *Symmetry and Flag manifolds*, Springer L.N. **996** (1983), 118–144.

[72] A.Lascoux, *Classes de Chern des variétés de drapeaux*, C.R. Acad. Sci. Paris **295** (1982), 393–398.

[73] A.Lascoux and M.P.Schützenberger, *Polynômes de Schubert*, C.R. Acad. Sci. Paris **294** (1982), 447–450.

[74] A.Lascoux and M.P.Schützenberger, *Structure de Hopf de l'anneau de cohomologie et de l'anneau de Grothendieck d'une variété de drapeaux*, C.R. Acad. Sci. Paris **295** (1982), 629–633.

[75] I.G.Macdonald, *Notes on Schubert polynomials*, Publ. LACIM **6**, UQAM, Montréal, 1991.

[76] D.Monk, *the Geometry of flags manifolds*, Proc. London M.S. (3)**9** (1959), 253–286.

[77] P.Pragacz and J.Ratajski, *Formulas for Lagrangian and orthogonal degeneraci loci; The $\tilde{Q}$-Polynomials Approach*, to appear in Compositio Math.

[78] S.Veigneau, *SP, a package on Schubert polynomials realized with the computer algebra system MAPLE*, Publ. IGM **95-1**, Université de Marne-la-Vallée, 1995, submitted to the Journal of Symbolic Computation.

Schubert polynomials and computations in $\mathbb{Z}[x_1, \ldots, x_n]$

[79] S.Fomin and C.Greene, *Noncommutative Schur functions and their applications*, preprint, 1995.

[80] S.Fomin and A.Kirillov, *Yang-Baxter equation, symmetric functions and Schubert polynomials*, Proceedings of the 5-*th* Conference "Formal Power Series and Algebraic Combinatorics", Florence 1993, 215–229.

[81] W.Fulton and A.Lascoux, *A Pieri formula in the Grothendieck ring of a flag bundle*, Duke Math.J. **76** (1994), 711–729.

Symmetric functions

[82] I.G.Macdonald, *Symmetric functions and Hall polynomials*, 2_{nd} ed., Oxford: Clarendon Press, 1995.

[83] D.Knutson, *λ-rings and the representation theory of the symmetric group*, Springer L.N.M. **308**, 1973.

[84] D.E.Littlewood, *The theory of group characters*, 2_{nd} ed., Oxford, 1950.

[85] J.Stembridge, *SF, a maple package for symmetric functions*, University of Michigan, 1993.

Symmetric group Hecke algebra representations and isomorphisms

[86] C.W.Curtis and I.Reiner, *Methods of representation theory with applications to finite groups and orders*, Vol. **1**, Wiley, 1990.

[87] R.Dipper and G.James, *Blocks and idempotents of Hecke algebras of general linear groups*, Proc. London Math. Soc., **54**, 57–82, 1987.

[88] W.Fulton and J.Harris, *Representation theory, a first course*, Springer-Verlag, 1991.

[89] J.E.Humphreys, *Reflection groups and Coxeter groups*, Cambridge University Press, Springer-Verlag, 1990.

[90] N.Iwahori, *On the structure of a Hecke ring of a Chevalley group over a finite field*, J. Fac. Sci. Univ. Tokyo, **10**, 215–236, 1964.

Tableaux

[91] D.E.Knuth, *Permutations, matrices and generalized young tableaux*, Pacific J. Math. **34** (1970), 709–727.

[92] A.Lascoux and M.P.Schützenberger, *Keys and standard bases, in invariant theory and tableaux*, D. Stanton ed., Springer-Verlag (1990), 125–144.

[93] A.Lascoux and M.P.Schützenberger, *Le monoïde plaxique*, Quaderni de la Ricerca Scientifica, **109** (1981), 129–156.

[94] C.Schensted, *Longest increasing and decreasing subsequences*, Canad. J. Math. **13** (1961), 179–191.

Index

A

B

C

E

F

G

H