ACE

an Algebraic Combinatorics Environment

for the computer algebra system MAPLE

User's Reference Manual

Version 1.0

Sébastien Veigneau

June 6, 1995

ACE, an Algebraic Combinatorics Environment for the computer algebra system MAPLE

People having participated to the 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@dir.univ-rouen.fr`

Alain Lascoux[1]

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

Jean-Yves Thibon[1]

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`

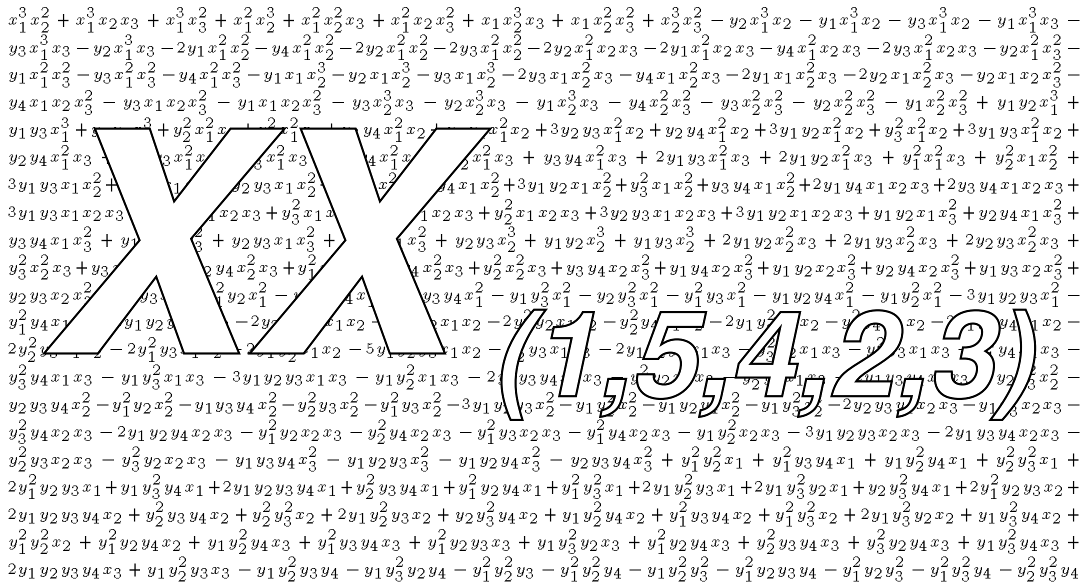
[1] Supported by the EC network "Algebraic Combinatorics".

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: veigneau@univ-mlv.fr



[2]Supported by the EC network “Algebraic Combinatorics”.

Copyright notice

Institut Gaspard Monge,
Université de Marne-la-Vallée


```
        __/__/__/   __/__/   __/__/__/
       __/    __/  __/       __/
      __/__/__/  __/       __/__/
     __/    __/  __/       __/
    __/    __/   __/__/   __/__/__/
```

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.

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 chose *Maple* because this algebra 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, while it may be more interesting to use some specific programs for large scale computations: we refer the reader to the `SYMMETRICA` system that is well suited for such computations.

Hence, `ACE` "was born" in 1994. It is 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. It is actually available on any MS-DOS or UNIX system, for *Maple V* Release 1, 2 and 3.

This documentation gives a brief description how `ACE` can be installed and how the new user can make his first steps towards an efficient use of the environment. Then it gives a user's documentation about all the functions available with `ACE`.

The environment will be extended by packages dealing with classical groups, Lie algebras and certain affine Lie algebras as well as their quantum analogues. Grothendieck polynomials, which permit efficient computations in the equivariant K-theory of flag manifolds will be added to the already existing `SP` package, to create a complete toolbox for intersection theory.

Contents

Chapter 1

Introduction

The Algebraic Combinatorics Environment is composed of various packages dealing with specific objects: `SG` (*symmetric group*), `SGA` (*symmetric group algebra*), `NCA` (*nilCoxeter algebra*), `IDCA` (*idCoxeter algebra*), `HEKA` (*generic Hecke algebra*), `HEQA` (*Hecke algebra*), `SYMF` (*symmetric functions*), `GL` (*linear group*), `NCSF` (*noncommutative symmetric functions*), `TAB` (*tableaux*) and `SP` (*Schubert polynomials*).

We give a brief presentation of these libraries.

1.1 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

$$\sum q^k \dim \mathrm{H}^{2k}(X_{3,4,1,2}, \mathbb{Q})$$

of the Schubert variety $X_{3,4,1,2}$ of the manifold $\mathcal{F}_4$ of complete flags of $\mathbb{C}^4$.

1.2 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]
```

1.3 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]
```

1.4 The IDCA library

It provides functions to work in the idCoxeter algebra corresponding to the symmetric group. This is the algebra generated by the simple π_i operators 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
```

1.5 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 gives a way to compute a special linear basis of the 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
```

1.6 The HEQA library

This package is devoted to the representation theory of the Hecke algebra of the symmetric group. The chosen normalization is $q_1 = q$, $q_2 = -1$ that is: $\tau_i^2 = (q-1)\tau_i + q$.

For instance, the `HeqaRepT` function returns the representation matrix of a given element of the Hecke algebra in the irreducible representation (as an orbit of q-deformation of the Vandermonde determinant) indexed by a given weakly decreasing shape:

```
ACE> HeqaRepT(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 )   ]
```

1.7 The SYMF library

It provides functions to manipulate symmetric functions. 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 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]);                              # Pieri formula.

 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.

1.8 The GL 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 Schur function basis to the basis of monomials (using an algorithm on Schubert polynomials):

```
ACE> GL_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
```

1.9 The NCSF library

This package provides functions to work with (integral) 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, that we check 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
```

1.10 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]):
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 `PermOnWord` function that realizes the action of a permutation on a noncommutative 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> t:=[[2,2,3], [1,1,2,2,3]]:          # t  is  a  two-rows  tableau,
ACE> Tab2Word("):                        # that we convert into a word.
ACE> PermOnWord([1,3,2], "):             # We act  on  this  word  and
ACE> Word2Tab(");                        # we   rebuild   the  tableau.

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

1.11 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 a lot of algorithms as for instance to express a Schubert polynomial in the basis of monomials: this is done by the `X2x` function that uses a 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:

```
ACE> TableX(3);             # computes  the  table  of  simple Schubert
                            # polynomials indexed by permutations in S3.

                       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 it works on codes using another algorithm.

The same combinatorics may be used to define Schubert polynomials on 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 an 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 the universal coefficients in the Newton interpolation of the functions in several variables. Their specializations (specialization of the second alphabet as a permutation of the first one) gives a way to express divided differences in the basis of permutations:

```
ACE> TableXX(3, [3,1,2]);            # computes  the  table  of  double
                                     # Schubert polynomials specialized
                table([               # to  y1 = x3,  y2 = x1,   y3 = x2.
                    [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 $\mathbb{Z}[x_1, \ldots, x_n]/I^+$ where I^+ is the ideal generated by the symmetric polynomials in $x_1, \ldots, x_n$ without constant terms. A linear geometrical basis of the cohomology ring is constituted of 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. Indeed, the variety $\mathcal{F}_n$ is equipped with n tautological line bundles $\mathrm{L}_1, \ldots, \mathrm{L}_n$ whose respective first *Chern classes* are equal to the classes of $x_1, \ldots, x_n$ modulo the ideal I^+.

The tangent bundle has the same Chern classes as the direct sum $\bigoplus_{i>j} \mathrm{L}_i \mathrm{L}_j^*$, that is

$$c(\mathcal{F}_n) = \prod_{i<j} (1 + x_i - x_j).$$

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> ToX( chern(4) );

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]
```

1.12 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.

The next chapter gives a brief exposition of the underlying theory.

Chapter 2

Background

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

Mein Weltbild, Albert Einstein.

It is not our purpose to develop the complete theory of the symmetric group, tableaux, or Schubert polynomials. We want in fact to touch a few aspects of the underlying theories to introduce some notions, set up notations as well as to establish some relations between the mathematical objects manipulated by `ACE`.

2.1 Symmetric group

2.1.1 Definitions and notations

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:

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

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 $1 \le i < j \le n$ 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, we will talk about a rise of a given permutation w as a position i such that $w(i) < w(i + 1)$.

The length $l(w)$ (`Perm2Length`) of a given permutation w is the number of inversions of w, *i.e.* the cardinal of the set $I(w)$ (`Inversions`) of inversions of w:

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

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 a vector of non-negative integers, the i-th component being the numbers 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 $c(w) := [1,4,2,1,0,1,0]$.

Given a permutation $w \in \mathfrak{S}_n$, we can fill a $n \times n$ square matrix with 0's and 1's so that the sub-matrix of 1's exactly represents the permutation w: this is done by having a 1 in all positions $(r, w(r))$ for $r = 1, \ldots, n$ where r denotes a row index and $w(r)$ stands for a column index. All other values being set to 0. For instance, we show the 7×7 square matrix representing 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 a quite 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. Consider the element of $\mathfrak{S}_7$, say $w := [2,6,4,3,1,7,5]$, its diagram $D(w)$ is the following:

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

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. In other words, $s_i := [1, \ldots, i+1, i, \ldots, n]$ where $i+1$ lies at position i, or equivalently $c(s_i) := [0, \ldots, 0, 1, 0, \ldots, 0]$ in which the unique 1 lies at position i. Elementary transpositions satisfy the following relations:

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

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

known as braid relations, together with the extra relation:

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

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

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

This decomposition is far from being unique (`ListRd`): the number of reduced decompositions is given by the function `NbRd`.

Let us fix the convention used to multiply a permutation w by an elementary transposition s_i:

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

where $w(i+1)$ lies at position i, and

$$s_i\, w := \begin{cases} [w(1), \ldots, i+1, \ldots, i, \ldots, w(n)] & \text{if } l(s_i\, w) > l(w), \\ [w(1), \ldots, i, \ldots, i+1, \ldots, w(n)] & \text{if } l(s_i\, w) < l(w). \end{cases}$$

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. 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

$$\begin{bmatrix} 1 & . & . & . & . & . & . \\ 2 & . & 3 & 4 & 5 & . & . \\ 3 & . & 4 & . & . & . & . \\ 4 & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & 6 & . & . \\ . & . & . & . & . & . & . \end{bmatrix}$$

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 reduced decompositions ranked according to their lengths, with edges corresponding to simple transpositions, is called the *permutohedron*, the order being the *weak order*: this 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*, and is the natural order for the idCoxeter algebra (`IDCA`).

The interval of permutations below a given one is obtained with the function `Interval`; `Betti`(*perm*) gives the number of permutations in the interval corresponding to the permutation *perm* according to their lengths.

2.1.2 Symmetric group algebra

The symmetric group algebra $\mathbb{K}[\mathfrak{S}_n]$ (`SGA`) in which $\mathbb{K}$ 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: $\texttt{Jucis}(k) := \sum_{i \leq k-1} t_{ik}$, sum on all transpositions exchanging i and k, $i = 1, \ldots, k-1$.

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`).

$\texttt{Carre}(perm)$ and $\texttt{Nabla}(perm)$) are other distinguished linear bases of $\mathbb{K}[\mathfrak{S}_n]$. One can obtain idempotents from them: $\texttt{Carre[LastPerm}(n)\texttt{]}$ is the sum of all permutations in $\mathfrak{S}_n$ and $\texttt{Nabla[LastPerm}(n)\texttt{]}$ is the alternated 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:

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

2.2 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

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

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

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

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

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

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

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

equation (2.3) being replaced by

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

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

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

Considering a polynomial $f \in \mathbb{K}[x_1]$ of degree lesser 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

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

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, e.g.

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

involving extra parameters.

2.3 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

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

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

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

This operator still satisfy the braid relations but now

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

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.

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

2.4 Generic Hecke algebra

The simple operators now act as follows:

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

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

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

Usually, one takes $q_1 = q$, $q_2 = -1$ (`HEQA`).

By product, 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`).

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.

2.5 Hecke algebra

Let $\mathbb{K}$ be a ring of coefficients and $q \in \mathbb{K}$. The usual Hecke algebra of type A (`HEQA`), denoted $\mathcal{H}_n$ is a deformation of the symmetric group algebra. With $(\tau_i)_{1\leq i<n}$ as set of generators, its presentation is

$$\begin{cases} \tau_i\tau_{i+1}\tau_i = \tau_{i+1}\tau_i\tau_{i+1} & i < n, \\ (\tau_i + 1)(\tau_i - q) = 0 & i \leq n. \end{cases}$$

As the braid relations appear in the presentation, the product $\tau_w := \tau_{i_1}\tau_{i_2}\cdots\tau_{i_k}$ is independent of the reduced decomposition $s_{i_1}s_{i_2}\cdots s_{i_k}$ of the permutation w. It is well known that $(\tau_w)_{w\in\mathfrak{S}_n}$ 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

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

The typical element of $\mathcal{H}_\infty$ then is the sum $\sum \lambda_w \tau_w$. There is a unique scalar product (`HeqaScal`) on $\mathcal{H}_n$ satisfying

$$\begin{cases} \langle 1|1\rangle = 1,\ \langle 1|\tau_w\rangle = 0, \\ \langle e\tau_i|f\rangle = \langle e|\tau_i f\rangle. \end{cases}$$

It will be called *invariant* and, if $\mathbb{K}$ is a field and q not a root of unity, it makes $\mathcal{H}_n$ into a symmetric algebra.

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

$$\sum_{i=1}^{k-1} q^{i-k}\tau_{t_{ik}}.$$

In case $\mathcal{H}_n$ is semi-simple, they generate a (self-adjoint) maximal commutative subalgebra. Minimal idempotents are polynomials in these elements (polynôme gallois computed by the `HeqaGall` function) indexed by tableaux.

We can construct a complete family of irreducible representations (`HeqaRepT` and `HeqaRepCT`) 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

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

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

$$\begin{cases} \tau_i.(\cdots x_i^h x_{i+1}^k \cdots) := (\cdots x_i^k x_{i+1}^h \cdots), \\ \tau_i.(\cdots x_i^h x_{i+1}^h \cdots) := q(\cdots x_i^h x_{i+1}^h \cdots), \\ \tau_i.(\cdots x_i^k x_{i+1}^h \cdots) := (q-1)(\cdots x_i^h x_{i+1}^k \cdots) + q(\cdots x_i^k x_{i+1}^h \cdots). \end{cases}$$

When I is increasing (resp. decreasing) a natural basis of the *Specht* module is indexed by contretableaux (resp. tableaux). In the semi-simple case, inverses in $\mathcal{H}_n$ can be computed by means of the minimal polynomials (`HeqaMinT` and `HeqaMinCT`). 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 for the matrix of e in a representation of shape I. The minimal polynomial Q of e in $\mathcal{H}_n$ is then

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

which can be written $Q = xR + a_0$. If $a_0 \neq 0$, we get $e^{-1} = \frac{1}{a_0} R(e)$ (these elements being computed by the use of the `HeqaInv` function).

2.6 Symmetric functions

The basic reference for this section is Macdonald's book [31].

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

$$\lambda_t(X) := \sum_{k \geq 0} t^k e_k(X) = \prod_{i \geq 1} (1 + x_i t). \tag{2.16}$$

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

$$\sigma_t(X) := \sum_{k \geq 0} t^k h_k(X) = \prod_{i \geq 1} (1 - x_i t)^{-1}. \tag{2.17}$$

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

$$\psi_t(X) := \sum_{k \geq 1} t^{k-1} p_k(X) = \sum_{i \geq 1} x_i (1 - x_i t)^{-1}. \tag{2.18}$$

These generating series satisfy the following relations

$$\psi_t(X) = \frac{d}{dt} \log \sigma_t(X) = -\frac{d}{dt} \log \lambda_{-t}(X), \tag{2.19}$$

$$\frac{d}{dt} \sigma_t(X) = \sigma_t(X)\, \psi_t(X), \tag{2.20}$$

$$-\frac{d}{dt} \lambda_{-t}(X) = \psi_t(X)\, \lambda_{-t}(X). \tag{2.21}$$

Formula (2.21) 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 $\lambda = (1^{\alpha_1}\, 2^{\alpha_2} \cdots)$, 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

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

These are linear bases of **Sym** (`Top`, `Toe`, `Toh`). For $\lambda \in \mathbb{Z}^r$, not necessarily a partition, the *Schur function* s_λ is defined by

$$s_\lambda = \det\left(h_{\lambda_i - i + j}\right)_{1\leq i,j\leq r} \tag{2.22}$$

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 $z_\lambda := \langle p_\lambda, p_\lambda\rangle = 1^{\alpha_1}\, \alpha_1!\, 2^{\alpha_2}\, \alpha_2! \cdots$ (`Zee`, `ScalarSf`). 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

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

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

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

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

$$\mathcal{F}([\rho]\otimes[\eta]) = \mathcal{F}(\chi_\rho\chi_\eta) = \mathcal{F}([\rho]) * \mathcal{F}([\eta]), \tag{2.23}$$

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

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

$$(F_1F_2\cdots F_r) * G = \mu_r[(F_1\otimes F_2\otimes\cdots\otimes F_r) * \Delta^r G], \tag{2.24}$$

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

$$\Delta^r(F * G) = \Delta^r(F) * \Delta^r(G). \tag{2.25}$$

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

$$\lambda_t(-X) := \sigma_{-t}(X) = [\lambda_t(X)]^{-1} \tag{2.26}$$

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

$$\lambda_t(X-Y) := \lambda_t(X)\ \lambda_t(-Y) = \lambda_t(X)\ \sigma_{-t}(Y). \tag{2.27}$$

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

$$\delta(F) = F(XY). \tag{2.28}$$

One can check that its adjoint is the internal product:

$$\langle \delta F\, ,\ P\otimes Q\rangle = \langle F\, ,\ P * Q\rangle. \tag{2.29}$$

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 iff

$$\sigma_1(XY) = \sum_I U_\lambda(X)\, V_\lambda(Y). \tag{2.30}$$

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.

2.7 Noncommutative symmetric functions

Recently, starting from the quasi-determinants of Gelfand and Retakh [7, 8] a noncommutative theory of symmetric functions has been developed [6], 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 rôle of the character ring is played by Solomon's descent algebra [17].

The *algebra of formal noncommutative symmetric functions* is the free associative algebra $\mathbf{Sym} = \mathbb{C}\langle\Lambda_1, \Lambda_2, \dots\rangle$ generated by an infinite sequence of noncommuting indeterminates Λ_k, called the *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

$$\lambda(t) = \sum_{k\geq 0} t^k \Lambda_k \ , \qquad \sigma(t) = \sum_{k\geq 0} t^k S_k = \lambda(-t)^{-1},$$

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

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

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

defined by the equation

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

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

$$F^I = F_{i_1}F_{i_2}\cdots F_{i_r}. \tag{2.31}$$

Then, (Λ^I), (S^I), (Φ^I) and (Ψ^I) are homogeneous bases of $\mathbf{Sym}$ (`ToL`, `ToS`, `ToPh`, `ToPs`).

The set of all compositions (`ListCompo`) of a given integer n is equipped with the *reverse refinement order* (`ListCompoFiner`, `IsFiner`), denoted $\preceq$. 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 basis (R_I) of *ribbon Schur functions*, originally defined in terms of quasi-determinants in [6], can also be defined by either of the two equivalent equations

$$S^I = \sum_{J\preceq I} R_J, \quad R_I = \sum_{J\preceq I} (-1)^{\ell(I)-\ell(J)} S^J, \tag{2.32}$$

$\ell(I)$ being the length of the composition I (`ToR`).

The commutative image of a noncommutative symmetric function is given by the algebra morphism $\Lambda_n \mapsto e_n$. Then, $S_n \mapsto h_n$, $\Psi_n \mapsto p_n$, $\Phi_n \mapsto p_n$, and R_I is sent to an ordinary ribbon Schur function, which is denoted by r_I. Ribbon Schur functions have been defined by McMahon (*see* [12], t. 1, p. 200), and are denoted in his book by h_I.

Let $\sigma \in \mathfrak{S}_n$ be a permutation with descent set $A = \{d_1 < \cdots < d_k\} \subseteq \underline{n-1} := \{1, \ldots, n-1\}$. The *descent composition* (`Perm2Desc`) $I = C(\sigma)$ is the composition $I = (i_1, \ldots, i_{k+1})$ of n defined by $i_s = d_s - d_{s-1}$, where $d_0 := 0$ and $d_{k+1} := n$. We also set $I = C(A)$, and conversely, the subset A of $\underline{n-1}$ associated to a composition I of n is denoted by $A = E(I)$. The sum in the group algebra of all permutations with descent composition I is denoted by D_I. The D_I with $|I| = n$ form a basis of a subalgebra $\mathbb{Z}[\mathfrak{S}_n]$, called the *descent algebra* of $\mathfrak{S}_n$ [17]. We denote by Σ_n the same algebra, with scalars extended to $\mathbb{C}$. There is an isomorphism of graded vector spaces

$$\alpha : \Sigma = \bigoplus_{n \geq 0} \Sigma_n \longrightarrow \mathbf{Sym} = \bigoplus_{n \geq 0} \mathbf{Sym}_n \tag{2.33}$$

such that

$$\alpha(D_I) = R_I \tag{2.34}$$

for any composition I.

The direct sum Σ can be given an algebra structure, by extending the natural product of its components Σ_n by setting $xy = 0$ for $x \in \Sigma_p$ and $y \in \Sigma_q$ with $p \neq q$. The *internal product* (`Internal`), denoted $*$, on **Sym** is defined by requiring that α be an *anti*-isomorphism. That is, we set

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

One also defines on **Sym** a coproduct Δ by any of the following equivalent conditions:

$$\Delta S_n = \sum_{k=0}^{n} S_k \otimes S_{n-k}, \quad \Delta\Lambda_n = \sum_{k=0}^{n} \Lambda_k \otimes \Lambda_{n-k}, \tag{2.35}$$

$$\Delta\Phi_n = \Phi_n \otimes 1 + 1 \otimes \Phi_n, \quad \Delta\Psi_n = \Psi_n \otimes 1 + 1 \otimes \Psi_n. \tag{2.36}$$

The fundamental property for computing with the internal product is the following splitting formula:

Theorem 2.7.1 ([6]) *Let $F_1, F_2, \ldots, F_r, G \in$ **Sym**. Then,*

$$(F_1 F_2 \cdots F_r) * G = \mu_r \left[(F_1 \otimes \cdots \otimes F_r) * \Delta^r G \right]$$

where in the right-hand side, μ_r denotes the r-fold ordinary multiplication and $*$ stands for the operation induced on $\mathbf{Sym}^{\otimes n}$ by $*$.

In the commutative case, this formula can be regarded as a particular case of the Mackey tensor product theorem.

With this coproduct, the involution $S_n \mapsto (-1)^n \Lambda_n$, and the natural unit and counit, **Sym** is a Hopf algebra, whose graded dual is Gessel's algebra of quasisymmetric functions [13].

It has been shown by Klyachko [10] that for a primitive n-th root of unity ε, the element

$$\kappa_n = \frac{1}{n} \sum_{\sigma \in \mathfrak{S}_n} \varepsilon^{\mathrm{maj}(\sigma)} \sigma$$

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

$$\sum_{n\geq 0} \frac{K_n(q)}{(q)_n} = \overleftarrow{\prod_{k\geq 0}} \sigma(q^k).$$

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

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

where A is just a label.

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.

As an application, one can show that the element

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

is a Lie idempotent [5, 11], interpolating between several known ones.

As shown in [11], this leads to a q-deformation of the Eulerian idempotents.

2.8 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 striclty 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 tableaux 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.

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);
```

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 ($i < j < k$):

$$kij \equiv ikj, \quad jki \equiv jik, \quad jii \equiv iji, \quad jji \equiv jij$$

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`.

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`.

Given a partition, the function `Part2ListStdTab` enumerates all the standard tableaux of shape this partition. The equivalent enumeration of Yamanouchi words is given by `ListYama`.

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 give 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`).

These two vectors are compatible with 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 a rank poset by the operation of deleting the first letter of a tableau and inserting from the right:

$$t = [[xu][v]\cdots[w]] \rightarrow t' = \texttt{Word2Tab}([u, v, \ldots, w, x])$$

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}.

2.9 Schubert polynomials

Schubert polynomials were introduced in 1982 by Lascoux and Schützenberger as a combinatorial tool for certain questions of algebraic geometry. Given a symmetric group $\mathfrak{S}_n$, they are the images by divided differences of the monomial $x^\delta = x_1^{n-1} x_2^{n-2} \cdots x_n^0$.

They are polynomials in n variables, indexed by permutations $w \in \mathfrak{S}_n$ —since divided differences are so—, which provide a convenient basis of the polynomial ring regarded as a free module of dimension $n!$ on the ring of symmetric polynomials. They are non-negative integral linear combinations of the monomials of the obvious basis $\{x_1^{i_1} \cdots x_n^{i_n} : i_k \leq n-k \text{ for } k = 1, \ldots, n\}$, over which they present many advantages.

2.9.1 Simple Schubert polynomials

Schubert polynomials verify a lot of interesting combinatorial properties related to the symmetric group. They contain as a subfamily Schur functions.

Schubert polynomials are compatible with the embedding $\mathfrak{S}_n \hookrightarrow \mathfrak{S}_{n+1}$ given by

$$[w(1), w(2), \ldots, w(n)] \longmapsto [w(1), w(2), \ldots, w(n), n+1].$$

If one identifies $\mathfrak{S}_n$ 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.

Algebraic calculation on Schubert polynomials may be described by manipulations of some combinatorial objects (permutations, codes, diagrams, tableaux, ...) or by the use of a scalar product and of self-adjoint operators. This combinatorics includes the combinatorics of Schur functions —partitions being considered as the Lehmer codes of special permutations, *Grassmannian permutations*— and moreover is very similar to it (Pieri's formula, scalar product). Essentially, the combinatorics of partitions (which is the backbone of computations with Schur functions) is replaced by the combinatorics of permutations.

Fix the symmetric group to be $\mathfrak{S}_n$ and let $x^\delta := x_1^{n-1} x_2^{n-2} \cdots x_n^0$. For each $w \in \mathfrak{S}_n$, the (simple) Schubert polynomial $\mathbf{X}_w$ is defined as

$$\mathbf{X}_w := \partial_{w^{-1}\omega}(x^\delta), \tag{2.37}$$

where ω denotes the longest element (maximal permutation) of $\mathfrak{S}_n$.

Schubert polynomials, for all permutations $w \in \mathfrak{S}_n$, are homogeneous polynomials in $x_1, x_2, \ldots, x_{n-1}$ of degree $l(w)$, symmetric in x_i, x_{i+1} if and only if the position i is a rise of w.

They verify the following interesting property: let w be a permutation that factorizes (`FactorPerm`) in a direct product $w' \, w''$. Then the Schubert polynomial $\mathbf{X}_w$ factorizes too. In other words, whenever $w \in \mathfrak{S}_n$ belongs to a Young diagonal subgroup $\mathfrak{S}_i \times \mathfrak{S}_j$ so that $w = w'w''$, we have also $\mathbf{X}_{w'w''} = \mathbf{X}_{w'} \mathbf{X}_{w''}$.

Schubert polynomials may also be indexed by sequences $I = (i_1, i_2, \ldots, i_n) \subset \mathbb{N}^n$ interpreted as the code of a permutation instead of the permutation itself. We denote by $\mathbf{Y}_I$ the Schubert polynomial corresponding to the sequence $I \subset \mathbb{N}^n$. On account of the above properties, one sees that $\mathbf{Y}_I, I \subset \mathbb{N}^n$ is homogeneous of degree $\sum_{k=1}^n i_k$.

The Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_n$ form a $\mathbb{Z}-$basis (`ToX`) of the subspace of $\mathbb{Z}[x_1, \ldots, x_n]$ spanned by the monomials $x^\alpha, \alpha \subseteq (n-1, n-2, \ldots, 0)$, but the crucial fact is that Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_\infty$ form a $\mathbb{Z}-$basis of the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates.

The Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_n$ form a basis for the algebra of polynomials $\mathbb{Z}[x_1, \ldots, x_n]$ as a module over the ring of symmetric polynomials $Sym[x_1, \ldots, x_n]$. The multiplicative structure in $Sym[x_1, \ldots, x_n]$ is determined by Pieri's formula that expresses in the basis of Schur functions the product $h_i s_I$ (resp. $e_i s_I$), $i \geq 0, I$ being a partition, where h_i (respectively e_i) denotes the i-th complete (respectively elementary) symmetric function. In the Schubert basis, the multiplicative structure is based on the multiplication by one variable (`Monk`). It involves a certain combinatorics on permutations that is described by Monk's formula:

$$x_r \mathbf{X}_w = \sum_{\nu = w\, t_{ir},\, i > r,\, l(\nu) = l(w)+1} \mathbf{X}_\nu - \sum_{\eta = w\, t_{ir},\, i < r,\, l(\eta) = l(w)+1} \mathbf{X}_\eta, \tag{2.38}$$

where t_{ij} denotes the transposition that interchanges i and j.

For the time being, Monk's formula appears as the fundamental tool for calculation on Schubert polynomials. For instance, one may choose r such that only one positive term appears in the product $x_r \mathbf{X}_w$. This provides a way to develop a Schubert polynomial $\mathbf{X}_\nu$ on the basis of monomials (`X2x`) since we have

$$\mathbf{X}_\nu = x_r \mathbf{X}_w + \sum \mathbf{X}_\eta. \tag{2.39}$$

The `TableX` function gives the table of all Schubert polynomials for $\mathfrak{S}_n$.

In relation with the cohomology ring of flag manifolds, it is natural to define a scalar product on $\mathbb{Z}[x_1, x_2, \ldots, x_n]$ for which elements of the ring of symmetric polynomials $Sym[x_1, x_2, \ldots, x_n]$ are scalars. The use of the ∂_ω operator seems to be the best adapted so that one defines the scalar product (corresponding to $\mathfrak{S}_n$) as

$$\langle f, g \rangle := \langle fg, 1 \rangle, \tag{2.40}$$

that is (in the basis of Schur functions) the symmetric function obtained by applying the maximal divided difference (`ScalarSf`) of $\mathfrak{S}_n$ to the product fg.

2.9.2 Double Schubert polynomials

We now consider two independent sets of indeterminates $x := \{x_1, \ldots, x_n\}$ and $y := \{y_1, \ldots, y_n\}$. One defines for each $w \in \mathfrak{S}_n$ the double Schubert polynomial $\mathbb{X}_w$ to be

$$\mathbb{X}_w(x, y) := \partial_{w^{-1}\omega}(\Delta(x, y)), \tag{2.41}$$

where ω denotes the longest element of $\mathfrak{S}_n$, divided differences act only on the x variables and

$$\Delta(x, y) := \prod_{i+j \leq n} (x_i - y_j). \tag{2.42}$$

These polynomials which belong to the polynomial ring $\mathbb{Z}[x_1, \ldots, x_{n-1}, y_1, \ldots, y_{n-1}]$ satisfy the following properties:

1.. For $w \in \mathfrak{S}_n$, $\mathbb{X}_w$ is a non-zero homogeneous polynomial in $x_1, \ldots, x_{n-1}, y_1, \ldots, y_{n-1}$ of degree $l(w)$,

2.. Setting all y_i's to 0, $\mathbb{X}_w = \mathbf{X}_w$ since $\Delta(x, 0) = x^\delta$,

3.. $\mathbb{X}_\omega = \Delta(x, y)$ and $\mathbb{X}_1 = 1$,

4.. For $w \in \mathfrak{S}_n$, $\mathbb{X}_{w^{-1}}(x, y) = \mathbb{X}_w(-y, -x)$.

Double Schubert polynomials generalize the coefficients $(x_1 - y_1) \cdots (x_1 - y_j)$ in the Newton interpolation formula (2.10) to the case of any number of variables. In other words, considering one set of indeterminates $x := \{x_1, x_2, \ldots\}$ and $y := \{y_1, \ldots, y_{n-1}\}$ denoting any set of elements of a field $\mathbb{K}$, the Newton interpolation formula (2.10) becomes for $f \in \mathbb{K}[x_1, x_2, \ldots]$ the equation

$$f(x) = \sum_w \partial_w(f)(y) \mathbb{X}_w(x, y), \tag{2.43}$$

in which divided differences act on the y variables.

Proposition 2.9.1

$$x_r \mathbb{X}_w = y_{w_r} \mathbb{X}_w + \sum_{\nu = w\, t_{ir},\, i>r,\, l(\nu) = l(w)+1} \mathbb{X}_\nu - \sum_{\eta = w\, t_{ir},\, i<r,\, l(\eta) = l(w)+1} \mathbb{X}_\eta, \tag{2.44}$$

where t_{ij} is the transposition that interchanges i and j.

Thus, working with two sets of variables involves only adding an extra term in Monk's formula (2.38). Furthermore, as in the case of simple Schubert polynomials, one may choose r such that

$$\mathbb{X}_\nu = (x_r - y_{w_r}) \mathbb{X}_w + \sum \mathbb{X}_\eta. \tag{2.45}$$

In the same manner as for the simple Schubert polynomials, it gives a way to efficiently compute the expression of a double Schubert polynomial in the basis of monomials but the most interesting point to notice is that it also provides an efficient way to specialize (`Specialize`) double Schubert polynomials, since in the relation (2.45) the term $(x_r - y_{w_r}) \mathbb{X}_w$ will be very soon specialized to zero whenever the specialization of the second alphabet to a permutation of the first one sends the factor $x_r - y_w$ onto zero.

The `TableXX` function gives the table of all double Schubert polynomials for $\mathfrak{S}_n$.

2.9.3 Schubert polynomials and algebraic geometry

Schubert polynomials have an important geometrical interpretation as *Schubert cycles*. One now works in the quotient ring (`Flag`) of the ring of polynomials modulo the ideal I^+ generated by the symmetric polynomials without constant term, which is the so-called ring of coinvariants of the symmetric group. This ring can be interpreted as the cohomology ring $H^*(\mathcal{F}_n, \mathbb{Z})$, or Chow ring, of the variety of complete flags in $\mathbb{C}^n$. With this interpretation, the class of the Schubert polynomial $\mathbf{X}_w$ is the Poincaré dual of the Schubert cycle $[\overline{B}_w]$. The flag manifold is equipped with tautological line bundles $L_1, \ldots, L_n$, whose respective first Chern classes are equal to the classes $\gamma_1, \ldots, \gamma_n$ of $x_1, \ldots, x_n$ modulo the ideal I^+.

The double Schubert polynomials also have a geometrical interpretation. 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.

Chapter 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, one should be aware of the following remark: even if `ACE` is available for any UNIX or MS-DOS system, for any Release 1, 2 or 3 of *Maple V*, one gets a specific version of the environment. That is to say, one gets for instance 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 and MS-DOS 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 (solely given with UNIX versions) to compile the environment on a UNIX system, and there are other programs (solely given with MS-DOS versions of `ACE`) for the installation on a MS-DOS system.

Before installing the environment, one should get the version corresponding to its system (UNIX or MS-DOS) and its *Maple V* release (1, 2 or 3), and then refer to the corresponding section.

3.1 Installing `ACE` on a UNIX system

One should read this section to install `ACE` in a UNIX environment.

One gets the `ACE` environment, as a compressed archive of files. First of all, one has to restore these files. According to the *Maple V* release one wants `ACE` to be installed for, one gets the corresponding compressed archive file, `ACE1UNIX.tgz` (*Maple V* release 1), `ACE2UNIX.tgz` (*Maple V* release 2) or `ACE3UNIX.tgz` (*Maple V* release 3).

In the following, we will suppose that one gets the `ACE3UNIX.tgz`, or in other words, one has 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 others.

3.1.1 Restoring the data

The `ACE3UNIX.tgz` is a "gziped" file, that is to say, it has been compressed by the UNIX program `gzip`. At first, one has 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 new file is the archive of the files available with `ACE`. It has been created by using the `tar` command. To see the detail on the files contained in the archive, please run the UNIX command

```
tar tvf ACE3UNIX.tar
```

and to really restore the `ACE` environment, 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.

One can get some informations about (almost) each file in the `ACE_PACK` directory by using the UNIX command `what`: for instance, in the directory `ACE_PACK/BIN`, one finds the file `runACE`, so that the result of the UNIX command `what runACE` gives:

```
runACE:
        runACE (C-shell script) Wed Mar  1 14:22:23 Frh 1995
```

that indicates that `runACE` is a C-shell script created on Wednesday, the 1$^{\text{st}}$ of March 1995.

3.1.2 Description of the `ACE_PACK` directory

This directory contains all the sources of the `ACE` environment.

The `BIN` subdirectory contains the `.modelinit` file that is a model of the `.mapleinit` *Maple* file that one should modify before installing `ACE`. It also contains the installation program, *i.e.* the `INSTALL.ACE` file (that is a C-shell program), the files `INSTALL.SRC` and `INSTALL.HLP` that describe how to respectively install source files and help files. Finally, we provide in the `BIN` subdirectory the `runACE` program (that is a C-shell program) that can be used to run *Maple* and automatically load `ACE`.

The `CPRIGHTS` subdirectory contains the `rights` file that gives the copyright notice. To see this copyright, one can for instance run the UNIX command: `what < rights`.

The `DOCUMENT` directory contains the documentation of `ACE`. This documentation is stored in various locations. Thus, one finds in the `DOCUMENT` directory, the `MANUAL` directory in which one gets a DVI and a postscript version of the LaTeX manual, both of them being stored in the `LaTeX` directory. Another directory in the `DOCUMENT` directory, is the `DATA` one that stores the documentation of each function. Each package has its corresponding directory in which the help of each function is stored. This help is exactly the *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 provided with the ACE environment, while the SOURCES subdirectory of ACE_PACK contains the sources of all functions.

3.1.3 Description of a typical package

Let us choose the SG package as a typical one, to present the organization of any package on the disk. It is essentially stored in the SG directory of respectively the SOURCES directory for the *Maple* sources and installation files, and the PACKAGES one for the *Maple* declaration of the package.

The SOURCES/SG directory contains the src directory in which one finds the *Maple* code of all functions of SG, the help directory that regroup the *Maple* help files for each function of the package. Moreover, the files directory is used by the individual installation program that is the file COMPILE.SG.

The PACKAGES/SG directory contains three files: SG that declares the package, SGext that declares all functions used in the package and finally, SGhlp that declares to *Maple* where to find the compiled help files about the functions of the package.

3.1.4 Step by step compilation

Having restored all the environment on the disk, we are able to install the package.

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 a part of this file, for *Maple V* Release 3:

```
####################################################################
#                                                                  #
# 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`:
#
ERRLib:=``.ACELib.`/ACE`:
GLLib:=``.ACELib.`/ACE`:
HEKALib:=``.ACELib.`/ACE`:
HEQALib:=``.ACELib.`/ACE`:
```

```
IDCALib:=``.ACELib.`/ACE`:
NCALib:=``.ACELib.`/ACE`:
NCSFLib:=``.ACELib.`/ACE`:
SGALib:=``.ACELib.`/ACE`:
SGLib:=``.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, GLLib, TEXLib, SGALib, NCALib, IDCALib,
           NCSFLib, HEQALib:
#
## 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, copy the `.modelinit` file into the existing `.mapleinit` file in the `HOME` directory. If you don't have any `.mapleinit` file, then create it by directly copying the `.modelinit` file into a new `.mapleinit` file. Then, 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 easiest 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` that 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. Thus, to install the environment, enter a UNIX command like:

```
csh INSTALL.ACE /people/roger/maple/lib
```

Individual installation

One can compile each package individually. This is possible since we provide for each package its installation program in its corresponding directory. For instance, one can compile the `GL` package by running a UNIX command that looks like:

```
csh COMPILE.GL /people/roger/maple/lib/ACE
```

As usually, enter the UNIX command `csh COMPILE.GL` to get more information and for instance to see that one can solely compile one file among all, by running a command like:

```
csh COMPILE.GL /people/roger/maple/lib/ACE src/Toe
```

3.1.5 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 such a shell so that this trouble could be due to an incomplete `.cshrc` file (that 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 would give some hints about the problem.

Whenever one gets a `writting error` message from *Maple* that looks like

```
Error, could not open ... for writing
```

be sure that the corresponding directory do exist. If it does not, be sure of having correctly set the corresponding variable in the `.mapleinit` file.

3.1.6 Using the environment

Having installed the environment, it is now time to use it. To do so, we have several possibilities.

First of all, one can run *Maple* by the (usual) UNIX command `maple`, and then to load the complete `ACE` environment, that is to load all packages, enter the *Maple* command `with(ACE);`. One can solely load a specific package, say `SYMF`, by the *Maple* command `with(SYMF);`.

Another way to use the `ACE` environment is to enter the `runACE` UNIX command provided in the `BIN` directory. It runs *Maple* and automatically loads the `ACE` environment. One can add the `BIN` directory to its `PATH` variable to have the `runACE` command accessible from anywhere. It can done for instance in the corresponding shell configuration file.

3.2 Installing ACE on a MS-DOS system

One should read this section to install `ACE` in a MS-DOS environment.

One gets the `ACE` environment, as a compressed archive of files. First of all, one has to restore these files. According to the *Maple V* release one wants `ACE` to be installed for, one gets 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 one gets the `ACE3DOS.tgz`, or in other words, one has to install the `ACE` environment on a MS-DOS operating system for *Maple V* release 3. Anyway, the same work has to be done for the others.

3.2.1 Restoring the data

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 that we refer by `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 the files: this is the `tar.exe` file given on the floppy disk that 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 one gets this "special" `tar` program, one 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 one does 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 Description of the ACE_PACK directory

This directory contains all the sources of the `ACE` environment.

The `BIN` subdirectory contains the `MAPLE.MOD` file that is a model of the `MAPLE.INI` *Maple* file that one should modify to install `ACE`. It also contains the installation program, *i.e.* the `INSTALL.BAT` file (that is a batch file), the files `INSTALL.SRC` and `INSTALL.HLP` that 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, one can for instance run the MS-DOS command: `type rights`.

The `DOCUMENT` directory contains the documentation of `ACE`. This documentation is stored in various locations. Thus, one finds in the `DOCUMENT` directory, the `MANUAL` directory in which one gets a DVI and a postscript version of the LaTeX manual, both of them being stored in the `LaTeX` directory. Another directory in the `DOCUMENT` directory, is the `DATA` one that stores the documentation of each function. Each package has its corresponding directory in which the help of each function is stored. This help is exactly the *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 provided with the `ACE` environment, while the `SOURCES` subdirectory of `ACE_PACK` contains the sources of all functions.

3.2.3 Description of a typical package

Let us choose the SG package as a typical one, to present the organization of any package on the disk. It is essentially stored in the SG directory of respectively the SOURCES directory for the *Maple* sources and installation files, and the PACKAGES one for the *Maple* declaration of the package.

The SOURCES/SG directory contains the src directory in which one finds the *Maple* code of all functions of SG, the help directory that regroup the *Maple* help files for each function of the package. Moreover, the files directory is used by the individual installation program that is the file COMPSG.BAT. The SG.ini file gives the lines that must be added to the MAPLE.INI file in the *Maple* library directory[1].

The PACKAGES/SG directory contains three files: SG that declares the package, SGEXT that declares all the functions used in the package and finally, SGHLP that declares to *Maple* where to find the help files about the functions of the package.

3.2.4 Step by step compilation

Having restored all the environment on the disk, we are able to install the package.

First of all, we have to tell *Maple* where to store the compiled files (*i.e.* where to store the .m files). This is done in the MAPLE.INI file in the *Maple* library directory (by default \MAPLEV\LIB), 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 MAPLE.INI file, so-called MAPLE.MOD in the BIN directory. Here is a part of this file, for *Maple V* Release 3:

```
####################################################################
#                                                                  #
# Modify the following line so that the ACELib variable  contains  #
# the directory where one wants maple to store the compiled files. #
#                                                                  #
####################################################################
               #
               #
            #######
             #####
              ###
               #
#
ACELib:=`/MAPLEV/LIB`:
#
ERRLib:=``.ACELib.`/ACE`:
GLLib:=``.ACELib.`/ACE`:
HEKALib:=``.ACELib.`/ACE`:
HEQALib:=``.ACELib.`/ACE`:
```

[1] that is usually the \MAPLEV\LIB directory.

```
IDCALib:=``.ACELib.`/ACE`:
NCALib:=``.ACELib.`/ACE`:
NCSFLib:=``.ACELib.`/ACE`:
SGALib:=``.ACELib.`/ACE`:
SGLib:=``.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, GLLib, TEXLib, SGALib, NCALib, IDCALib,
           NCSFLib, HEQALib:
#
## 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, 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).

One can 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 `MODEL.INI` file, all packages will be compiled into subdirectories of the `ACELib` directory but even if it is not necessary, it is easiest to manage.

Global installation

Having modified the `MAPLE.INI` file, we are ready to install all packages by using the `INSTALL.BAT` program in the `BIN` directory. Being in this directory, enter the MS-DOS command, `INSTALL.BAT` that 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. Thus, to install the environment, enter a MS-DOS command like:

```
INSTALL.BAT \MAPLEV\LIB
```

Individual installation

One can compile each package individually. This is possible since we provide for each package, its installation program in its corresponding directory. For instance, one can compile the `GL` package by running a MS-DOS command that looks like:

```
COMPGL.BAT \MAPLEV\LIB\ACE
```

As usually, enter the MS-DOS command `COMPGL.BAT` to get more information and for instance to see that one can solely compile one file among all, by running a command like:

```
COMPGL.BAT \MAPLEV\LIB\ACE SRC\TOE
```

3.2.5 Using the environment

Having installed the environment, one can use it by running *Maple* and then loading the complete `ACE` environment (that loads all packages) by the *Maple* command `with(ACE);`. One can solely load a specific package, say `SYMF`, by the *Maple* command `with(SYMF);`.

Chapter 4

Examples

This chapter presents some examples provided with the environment. These examples are given in the subdirectories of the `EXAMPLES` directory.

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 such a computation.

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 of Grothendieck polynomials for S3.

              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 $x_1^{i_1}x_2^{i_2}\cdots$, $i_1 \geq i_2 \geq \cdots$, under a Demazure operator. The following program gives a way to compute such 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;
  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 the code *cod* of a permutation.

```
KeyPol:=proc(cod)
  local
    i,j,pos,
    perm,rcod,tmp;
  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;
  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...
    d; # the projective degree of the Schubert variety...
  n:=nops(perm);
  d:=n*(n-1)/2 - Perm2Length(perm);
  Flag(n);
  r:=subs(X[op(LastPerm(n))]=1, ToX(h(n)^d*X[op(perm)]));
  Flag();
  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 $e^{\sum_{1\le i<j\le n}(x_i-x_j)}$, 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...
    k;   # variable for loop...
  pol:=1;
  hk:=1;
  Flag(n);
  for k from 1 to n*(n-1)/2 do
    hk:=ToX(hk*h(n));
    pol:=pol + hk/factorial(k)
  od;
  Flag();
  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

$$\kappa_n = \frac{1}{n} \sum_{\sigma\in\mathfrak{S}_n} \varepsilon^{\mathrm{maj}(\sigma)}\sigma$$

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

$$\sum_{n\ge 0} \frac{K_n(q)}{(q)_n} = \overleftarrow{\prod_{k\ge 0}} \sigma(q^k) \ .$$

Let us check these properties on the example of κ_4. The corresponding symmetric function is constructed by the sequence

```
ACE> C:=ListCompo(4): 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, one enters

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

one has 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, one 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

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

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

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

is a Lie idempotent [5, 11], 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;
```

Chapter 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 or for instance Schubert polynomials.

Each function description has the following format : after a one-line description showing how the function can be called, one finds 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` in for instance 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 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 that 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 the followings : `GL`, `HEKA`, `HEQA`, `IDCA`, `NCA`, `SGA`, `SG`, `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 for 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:

`SysInfo`	`Time`

- For help with a particular function do `?ACE[<function>]` or `?ACE,<function>` where `<function>` is one from the above list.

5.1.1 `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:

5.1.2 `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 $f(a_1, ..., a_k)$ and saves in the parameter 't' the computation time.
- The value of `Time`$(f,'t', a_1, ..., a_k)$ is the value of $f(a_1, ..., a_k)$.
- 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, 't', 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]

> t;
                                     .316
```

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	Gauss	Interval	Inversions
InvPerm	IsDominant	IsGrassmannian	IsVexillar
LastPerm	ListDominant	ListGrassmannian	ListPerm
ListRd	Mat2Perm	MultPerm	NbCC
NbDominant	NbGrassmannian	NbRd	NbVexillar
NextPerm	Perm2Code	Perm2Cycle	Perm2CycleType
Perm2Desc	Perm2Length	Perm2Mat	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 `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`

5.2.2 `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`

5.2.3 `CCard` — compute the cardinal of a conjugacy class

Calling Sequence:

`CCard`($part$)
`SG[CCard]`($part$)

Parameters:

$part$ — any list denoting a partition

Synopsis:

- The `CCard` 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 `CCard` and another name used in the same session, use the long form `SG['CCard']`.

Examples:

```
> with(SG):
> CCard([2,1,1]);
```

6

SEE ALSO: `CycleType, Zee`

5.2.4 `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`

5.2.5 `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`

5.2.6 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

5.2.7 DimRep — compute the dimension of an irreducible representation

Calling Sequence:

DimRep(*part*)
SG[DimRep](*part*)

Parameters:

part — any list denoting a partition

Synopsis:

- The DimRep function computes the dimension of the irreducible representation of index *part*, for the symmetric group of order the weight of *part*.
- 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
```

SEE ALSO: `SnCharTable`

5.2.8 `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:

5.2.9 `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`

5.2.10 `Gauss` — compute a Gaussian polynomial

Calling Sequence:

`Gauss`$(m, n, 'q')$
`SG[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 $\binom{n}{m}$.
- `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`$(m-1, n-1, q) + q^n \times$ `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 `SG['Gauss']`.

Examples:

```
> with(SG):
> 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:

5.2.11 `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]`

5.2.12 `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`

5.2.13 `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:

5.2.14 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`

5.2.15 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`

5.2.16 `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

5.2.17 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

5.2.18 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`

5.2.19 `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`

5.2.20 `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' or 'cixel'

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.
- 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]]
```

SEE ALSO: `FirstPerm`, `LastPerm`, `NextPerm`, `PrevPerm`

5.2.21 ListRd — list all reduced decompositions of a given permutation

Calling Sequence:

ListRd(*perm*)
SG[ListRd](*perm*)

Parameters:

perm — any list denoting a permutation

Synopsis:

- The ListRd 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 ListRd and another name used in the same session, use the long form SG['ListRd'].

Examples:

```
> with(SG):
> ListRd([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: NbRd, Perm2Rd, Rd2Perm

5.2.22 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`

5.2.23 `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]`

5.2.24 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

5.2.25 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

5.2.26 `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`

5.2.27 `NbRd` — compute the number of reduced decompositions

Calling Sequence:

`NbRd`($perm$)
`SG[NbRd]`($perm$)

Parameters:

$perm$ — any list denoting a permutation

Synopsis:

- The `NbRd` function computes the number of reduced decompositions of a given permutation.
- Whenever there is a conflict between the function name `NbRd` and another name used in the same session, use the long form `SG['NbRd']`.

Examples:

```
> with(SG):
> NbRd([6,5,4,3,2,1]);
```

```
292864
```

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

```
11
```

SEE ALSO: `ListRd`

5.2.28 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

5.2.29 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

5.2.30 `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`

5.2.31 `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

5.2.32 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

5.2.33 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`

5.2.34 `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`

5.2.35 `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`

5.2.36 `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`

5.2.37 `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 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 sub-matrix of x's is the matrix which represents 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 `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`

5.2.42 `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`

5.2.43 `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

5.2.44 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

5.2.45 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`

5.2.46 `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`

5.2.47 `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

5.2.48 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 sub-matrix of x's is the matrix which represents 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`

5.2.49 `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]`

5.2.50 `Transpo` — return a transposition

Calling Sequence:

`Transpo`(i, n)
`Transpo`(i, j, n)
`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, n)=$[1, ..., i+1, i, ..., n]$.
- `Transpo`(i, j, n) =$[1, ..., j, ..., i, ..., n]$.
- 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:

5.2.51 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.
- Do either ?TYP,Permutation or ?TYP[Permutation] to get more information on permutations.
- 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	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 length$(perm\, s_i)$ > 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 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]

5.3.4 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`

5.3.5 `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:

5.3.6 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:

5.3.7 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 $y_1, y_2, y_3, \ldots$

- 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 action of the element Transpo(i, n) on the Schubert polynomial $\mathbf{X}_{perm}$ is : let ν =MultPerm($perm$, Transpo(i, n)), then

$$\begin{array}{lll} \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_{perm} & \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_{perm} + (x_{i+1} - x_i)\mathbf{X}_{\nu} & \text{otherwise.} \end{array}$$

- The result is expressed on the $\mathbf{X}$ 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 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.
- { 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]
```

SEE ALSO:

5.4 NCA and 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	NcaOnXX	NcaOnY
NcaOnYY	NcaYang		

- 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 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:

5.4.2 `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:

5.4.3 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 :

$$f \longrightarrow \frac{f(\ldots, x_i, x_{i+1}, \ldots) - f(\ldots, x_{i+1}, x_i, \ldots)}{x_i - x_{i+1}}$$

- 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 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 **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 result is expressed on the **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).
- 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 `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.
- 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`

5.4.9 `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).

- 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`

5.4.10 `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.
- 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`

5.5 IDCA and 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	IdcaYang

- 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 $s_i s_j s_h \cdots$ 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 `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:

5.5.3 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:

5.5.4 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 :

$$exp \longrightarrow \frac{(exp - exp')}{x_i - x_{i+1}} x_{i+1}$$

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 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 **X** Schubert basis.

- 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

$$\begin{array}{lll} \mathbf{X}_{perm} & \longrightarrow & 0 \quad \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & \mathbf{X}_{\nu}\, x_{i+1} \quad \text{otherwise.} \end{array}$$

- The result is expressed on the **X** Schubert basis and is not collected.

- Whenever there is a conflict between the function name 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 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.
- { 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:

5.6 HEKA and the generic Hecke algebra

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	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 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:

5.6.2 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:

5.6.3 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 :

$$exp \longrightarrow \frac{(exp - exp')(q_2 x_i + q_1 x_{i+1})}{(x_i - x_{i+1})} + q_1\, exp$$

  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 $y_1, y_2, y_3, \ldots$
- 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 **X** Schubert basis.
- The action of the element Transpo(i, n) on the Schubert polynomial $\mathbf{X}_{perm}$ is : let ν =MultPerm($perm$, Transpo(i, n)), then

$$\begin{array}{lll} \mathbf{X}_{perm} & \longrightarrow & q_1\mathbf{X}_{perm} & \text{if length}(\nu) > \text{length}(perm) \\ \mathbf{X}_{perm} & \longrightarrow & q_1\mathbf{X}_{perm} + (q_2x_i + q_1x_{i+1})\mathbf{X}_\nu & \text{otherwise.} \end{array}$$

- The result is expressed on the **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 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.
- { 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 HEQA and the Hecke algebra

Calling Sequence:

HEQA[<function>](*args*)

<function>(*args*)

Synopsis:

- This package provides functions to work with the Hecke algebra of type A. This algebra is generated by simple operators τ_i's satisfying braid relations and the Hecke relation:

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

- An element of the Hecke algebra is expressed as a linear combination of permutations with arbitrary coefficients, the identity element being represented by the identity permutation.
- Whenever there is a conflict between a function name in HEQA and another name used in the same session, use the long form HEQA[<function>].
- The available functions are:

HeqaCIdemp	HeqaCharCT	HeqaCharT	HeqaEmbed
HeqaGall	HeqaInv	HeqaJucis	HeqaMinCT
HeqaMinT	HeqaMult	HeqaRepCT	HeqaRepT
HeqaScal			

- Functions as HeqaChaCT and HeqaCharT (or HeqaMinCT and HeqaMinT) 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 ?HEQA[<function>] or ?HEQA,<function> where <function> is one from the above list.

Examples:

```
> with(HEQA):
> HeqaRepT(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 )    ]
```

5.7.1 `HeqaCIdemp` — central idempotent of the Hecke algebra

Calling Sequence:

`HeqaCIdemp`(*part*)
`HEQA[HeqaCIdemp]`(*part*)

Parameters:

f — any list of positive integers in weakly decreasing order

Synopsis:

- The `HeqaCIdemp` function calculates the central idempotent of the Hecke algebra of type A. It is the sum of the minimal idempotents taken over all tableaux of the given shape *part*.

- Whenever there is a conflict between the function name `HeqaCIdemp` and another name used in the same session, use the long form `HEQA['HeqaCIdemp']`.

Examples:

```
> with(HEQA):
> HeqaCIdemp([1,1]);

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

SEE ALSO: `HeqaGall`

5.7.2 `HeqaCharCT` — characteristic polynomial corresponding to an increasing shape

Calling Sequence:

`HeqaCharCT`($e, part, x$)
`HEQA[HeqaCharCT]`($e, part, x$)

Parameters:

e — any element of the Hecke algebra
part — any list of positive integers in weakly increasing order
x — any name or algebraic expression

Synopsis:

- The `HeqaCharCT` function returns the characteristic polynomial in x of the image of the element e of the Hecke algebra in the irreducible representation indexed by the shape *part*.

- Whenever there is a conflict between the function name `HeqaCharCT` and another name used in the same session, use the long form `HEQA['HeqaCharCT']`.

Examples:

```
> with(HEQA):
> HeqaCharCT(a*[2,1]+b*[3,1,2],[1,2],x);
```

$$x^2 - a q x + x a + x b q - a q^2 b - a^2 q + a q b + b^2 q^2$$

SEE ALSO: HeqaCharT, HeqaMinCT, HeqaMinT

5.7.3 HeqaCharT — characteristic polynomial corresponding to a decreasing shape

Calling Sequence:

HeqaCharT($e, part, x$)
HEQA[HeqaCharT]($e, part, x$)

Parameters:

e	—	any element of the Hecke algebra
$part$	—	any list of integers in weakly decreasing order
x	—	any name or algebraic expression

Synopsis:

- The HeqaCharT function returns the characteristic polynomial in x of the image of the element e of the Hecke algebra in the irreducible representation indexed by the shape $part$.
- Whenever there is a conflict between the function name HeqaCharT and another name used in the same session, use the long form HEQA['HeqaCharT'].

Examples:

```
> with(HEQA):
> HeqaCharT(a*[2,1]+b*[3,1,2],[2,1],x);
```

$$x^2 - a q x + x a + x b q - a q^2 b - a^2 q + a q b + b^2 q^2$$

SEE ALSO: HeqaCharCT, HeqaMinCT, HeqaMinT

5.7.4 HeqaEmbed — normalize an element of the Hecke algebra

Calling Sequence:

HeqaEmbed(e)
HEQA[HeqaEmbed](e)

Parameters:

e	—	any element of the Hecke algebra

Synopsis:

- The `HeqaEmbed` function normalizes the element e of the Hecke algebra of type A.
- Whenever there is a conflict between the function name `HeqaEmbed` and another name used in the same session, use the long form `HEQA['HeqaEmbed']`.

Examples:

```
> with(HEQA):
> HeqaEmbed(a*[3,1,2,4,5] + b^3*[5,1,4,2,3] + (c-1)*[3,1,2]);

                 3
                b  [5, 1, 4, 2, 3] + (a + c - 1) [3, 1, 2]
```

SEE ALSO:

5.7.5 `HeqaGall` — Gallois polynomial

Calling Sequence:

`HeqaGall`(t)
`HEQA[HeqaGall]`(t)

Parameters:

t — any Yamanouchi word coding a tableau

Synopsis:

- The `HeqaGall` function calculates the minimal idempotent of the Hecke algebra of type A for the tableau t.
- Yamanouchi words are given on the alphabet $0, 1, 2, \ldots$
- Whenever there is a conflict between the function name `HeqaGall` and another name used in the same session, use the long form `HEQA['HeqaGall']`.

Examples:

```
> with(HEQA):
> HeqaGall([0,1]);

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

SEE ALSO: `HeqaCIdemp`

5.7.6 HeqaInv — invert an element of the Hecke algebra

Calling Sequence:

HeqaInv(e)
HEQA[HeqaInv](e)

Parameters:

e — any element of the Hecke algebra

Synopsis:

- The HeqaInv function returns the inverse of e in the Hecke algebra.
- Whenever the element e is not invertible, the function returns 0.
- Whenever there is a conflict between the function name HeqaMult and another name used in the same session, use the long form HEQA['HeqaMult'].

Examples:

```
> with(HEQA):
> e:=a*[2,1]:
> HeqaInv(e);

                    [2, 1]   /         1 \
                    ------ + |- 1/a + ---| [1]
                      a q    \        a q/

> simplify( HeqaMult(e, ") );

                                 [1]

> HeqaInv([2,1] - [1,3,2]);

                                  0
```

SEE ALSO:

5.7.7 HeqaJucis — compute a Jucis-Murphy element

Calling Sequence:

Jucis(i)
HEQA[Jucis](i)

Parameters:

i — any positive integer

Synopsis:

- The HeqaJucis function computes the i-th Jucis-Murphy element inside the 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`$(3) = T(1,3)/q^2 + T(2,3)/q, \ldots,$ `Jucis`(k) $= T(1,k)/q^(k-1) + T(2,k)/q^(k-2) + \cdots + T(k-1,k)/q$.

- Whenever there is a conflict between the function name `HeqaJucis` and another name used in the same session, use the long form `HEQA['HeqaJucis']`.

Examples:

```
> with(HEQA):
> HeqaJucis(4);
```

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

SEE ALSO: `SG[Transpo]`

5.7.8 `HeqaMinCT` — minimal polynomial corresponding to an increasing shape

Calling Sequence:

`HeqaMinCT`(e, x)
`HeqaMinCT`$(e, part, x)$
`HEQA[HeqaMinCT]`(e, x)
`HEQA[HeqaMinCT]`$(e, part, x)$

Parameters:

e	—	any element of the Hecke algebra
$part$	—	any (extra) list of positive integers in weakly increasing order
x	—	any name or algebraic expression

Synopsis:

- The `HeqaMinCT` function returns the minimal polynomial in x of the representation of the element e of the Hecke algebra for the shape $part$.

- 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 `HeqaMinCT` and another name used in the same session, use the long form `HEQA['HeqaMinCT']`.

Examples:

```
> with(HEQA):
> HeqaMinCT(a*[2,1] + b*[3,2,1], [1,2], x);
```

```
             2         2       2  3                  2
          - a  q + a q  b - b  q  - a (q - 1) x + x
```

```
> HeqaMinCT(a*[2,1], x);
```

```
                    2                    2
                   a  q + a q x - x a - x
```

SEE ALSO: HeqaCharCT, HeqaCharT, HeqaMinT

5.7.9 HeqaMinT — minimal polynomial corresponding to a decreasing shape

Calling Sequence:

HeqaMinT(e, x)
HeqaMinT($e, part, x$)
HEQA[HeqaMinT](e, x)
HEQA[HeqaMinT]($e, part, x$)

Parameters:

e	—	any element of the Hecke algebra
$part$	—	any (extra) list of positive integers in weakly decreasing order
x	—	any name or algebraic expression

Synopsis:

- The HeqaMinT function returns the minimal polynomial in x of the representation of the element e of the Hecke algebra for the shape $part$.
- If $part$ is not specified then it returns the least common multiple of all minimal polynomials for the decreasing shapes of minimal weight.
- Whenever there is a conflict between the function name HeqaMinT and another name used in the same session, use the long form HEQA['HeqaMinT'].

Examples:

```
> with(HEQA):
> HeqaMinT(a*[2,1]+b*[3,2,1],[2,1],x);
```

```
             2         2       2  3                  2
          - a  q + a q  b - b  q  - a (q - 1) x + x
```

```
> HeqaMinT(a*[2,1],x);
```

```
                    2                    2
                   a  q + a q x - x a - x
```

SEE ALSO: HeqaCharCT, HeqaCharT, HeqaMinCT

5.7.10 HeqaMult — multiplication of two elements of the Hecke algebra

Calling Sequence:

HeqaMult(e_1, e_2)
HEQA[HeqaMult](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the Hecke algebra

Synopsis:

- The HeqaMult function multiplies two elements of the Hecke algebra.
- The identity element is denoted by the identity permutation.
- Whenever there is a conflict between the function name HeqaMult and another name used in the same session, use the long form HEQA['HeqaMult'].

Examples:

```
> with(HEQA):
> a:=v*[1,2,3] + u/t*[3,1,2]:
> b:=t*[2,1,3] - [1,3,2]:
> HeqaMult(a, b);

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

SEE ALSO:

5.7.11 HeqaRepCT — representation of an element of the Hecke algebra for an increasing shape

Calling Sequence:

HeqaRepCT($e, part$)
HEQA[HeqaRepCT]($e, part$)

Parameters:

e — any element of the Hecke algebra
$part$ — any list of integers in weakly increasing order

Synopsis:

- The HeqaRepCT function returns the representation matrix of the element e of the Hecke algebra in the irreducible representation indexed by an increasing shape $part$.
- Whenever there is a conflict between the function name HeqaRepCT and another name used in the same session, use the long form HEQA['HeqaRepCT'].

Examples:

```
> with(HEQA):
> HeqaRepCT(a*[3,2,1] + b*[1], [1,1,2]);

      [                                         2              ]
      [  b       0                         - a q               ]
      [                                                        ]
      [  0    - a q + b              a q (- q + 1)             ]
      [                                                        ]
      [ - a  a (- q + 1)  a ((q - 1) (- q + 1) + q (q - 1)) + b ]
```

SEE ALSO: HeqaRepT

5.7.12 HeqaRepT — representation of an element of the Hecke algebra for a decreasing shape

Calling Sequence:

HeqaRepT($e, part$)
HEQA[HeqaRepT]($e, part$)

Parameters:

e — any element of the Hecke algebra
$part$ — any list of integers in weakly decreasing order

Synopsis:

- The HeqaRepT function returns the representation matrix of the element e of the Hecke algebra in the irreducible representation indexed by a decreasing shape *part*.
- Whenever there is a conflict between the function name HeqaRepT and another name used in the same session, use the long form HEQA['HeqaRepT'].

Examples:

```
> with(HEQA):
> HeqaRepT(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 ]
```

SEE ALSO: HeqaRepCT

5.7.13 HeqaScal — the invariant scalar product between two elements of the Hecke algebra

Calling Sequence:

HeqaScal(e_1, e_2)
HEQA[HeqaScal](e_1, e_2)

Parameters:

e_1, e_2 — any elements of the Hecke algebra

Synopsis:

- The HeqaScal function returns the invariant scalar product between two elements of the Hecke algebra of type A.
- Whenever there is a conflict between the function name HeqaScal and another name used in the same session, use the long form HEQA['HeqaScal'].

Examples:

```
> with(HEQA):
> HeqaScal([2,3,1], [2,3,1]);

                                    2
                                   q

> HeqaScal(a*[3,1,2] + b*[2,1], c*[3,1,2,4] + d*[2,3,1]);

                                     2
                                a c q
```

See Also: SG[Perm2Length]

5.8 SYMF and 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 by default {'e', 'h', 'p', 's', 'm'}.
- 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:

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			

- 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]
```

5.8.1 `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: `CharTable`, `ITensor`, `Plethysm`, `Sf2Char`, `Zee`

5.8.2 `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:

5.8.3 `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 `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 $sf = p_i p_j \cdots$, `EvalSf`$(sf, expr)$ is set to be equal to the product `EvalSf`$(p_i, expr) \times$ `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);
```

$$(2x^3 + 3y^3)(2x^4 + 3y^4) + q(2x^2 + 3y^2)$$

SEE ALSO: `Plethysm`

5.8.4 `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

$$a \times \mathrm{arm\_length}[i, j] + \mathrm{leg\_length}[i, j] + 1$$

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

$$1 - q^{\mathrm{arm\_length}[i,j]}\, t^{(1+\mathrm{leg\_length}[i,j])}$$

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(", *);
```

```
> Hooks([2,1], z);

                                    z + 2

> Hooks([2,1], z, t);

                                  2         2
                          (1 - z t ) (1 - t)
```

SEE ALSO: Conjugate, ListPart

5.8.5 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, ITensor($\cdots p_2^j p_1^i, \cdots p_2^{j'} p_1^{i'}$) is equal to 0 or Zee($[\ldots, 2^j, 1^i]$) $\cdots p_2^j p_1^i$ whether $(\ldots, j, i) \neq (\ldots, j', i')$ 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

5.8.6 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 bi's instead of the hi'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:

5.8.7 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

5.8.8 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:

5.8.9 `ListPartIn` — list of partitions contained into a given partition

Calling Sequence:

`ListPartIn`($part$)
`SYMF[ListPart]`($part$)

Parameters:

$part$ — any list denoting a partition

Synopsis:

- The `ListPartIn` function gives all partitions contained into the diagram of a given partition. The list is ordered by increasing length.

- 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]]
```

SEE ALSO:

5.8.10 `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`

5.8.11 `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`

5.8.12 `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

5.8.13 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: `Char2Sf`, `EvalSf`, `ITensor`, `Sf2Char`

5.8.14 `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:

5.8.15 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 $sf = \cdots p_3^i p_2^j p_1^k$, then ScalarSf($sf, sf$) = Zee($[\ldots, 3^i, 2^j, 1^k]$).

- 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

5.8.16 `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: `CharTable`, `Char2Sf`, `ITensor`, `Plethysm`, `Zee`

5.8.17 `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`

5.8.18 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: Toh, Top, Tos

5.8.19 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: Toe, Top, Tos

5.8.20 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: Toe, Toh, Tos

5.8.21 `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: `Toe, Toh, Top`

5.8.22 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 $p = \cdots p_2^i\, p_1^j$, 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:

5.8.23 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

5.8.24 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

5.8.25 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`

5.8.26 `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`

5.8.27 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

5.8.28 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`

5.8.29 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`

5.8.30 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`

5.8.31 `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`

5.8.32 `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:

5.8.33 `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:

5.8.34 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

5.9 GL and the linear group

Calling Sequence:

GL[<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 GL_n (default degree is 5).
- Whenever there is a conflict between a function name in GL and another name used in the same session, use the long form GL[<function>].
- The available functions are:

GL_n	LRrSf_n	MuirSf_n	PieriSf_n
Toe_n	Toh_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 ?GL[<function>] or ?GL,<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(GL):
> GL_n(5);
```

5

5.9.1 GL_n — fix the order of the linear group

Calling Sequence:

GL_n(n)
GL[GL_n](n)

Parameters:

n — any positive integer

Synopsis:

- The GL_n function may be used to fix the linear group degree to n. Default order is 5.
- Whenever there is a conflict between the function name GL_n and another name used in the same session, use the long form GL['GL_n'].

Examples:

```
> with(GL):
> GL_n(5);
```

```
                                  5
```

SEE ALSO:

5.9.2 LRrSf_n — Littlewood-Richardson rule

Calling Sequence:

LRrSf_n($schur_1, schur_2$)
GL[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 $\leq$ _GLn.
- Whenever there is a conflict between the function name LRrSf_n and another name used in the same session, use the long form GL['LRrSf_n'].

Examples:

```
> with(GL):
> _GLn;
```

```
                              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

5.9.3 MuirSf_n — Muir's formula

Calling Sequence:

MuirSf_n(*mono*, *schur*)
GL[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 $\leq$ _GLn.
- Whenever there is a conflict between the function name MuirSf_n and another name used in the same session, use the long form GL['MuirSf_n'].

Examples:

```
> with(GL):
> _GLn;

                                   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

5.9.4 PieriSf_n — Pieri's formula

Calling Sequence:

PieriSf_n(*f*, *schur*)
GL[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 $\leq$ _GLn.

- Whenever there is a conflict between the function name `PieriSf_n` and another name used in the same session, use the long form `GL['PieriSf_n']`.

Examples:

```
> with(GL):
> _GLn;

                                  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`

5.9.5 `Toe_n` — convert any symmetric function to a e-polynomial

Calling Sequence:

`Toe_n`(sf)
`Toe_n`(sf, b)
`GL[Toe_n]`(sf)
`GL[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 `GL['Toe_n']`.

Examples:

```
> with(GL):
> _GLn;

                                     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`

5.9.6 Toh_n — convert any symmetric function to a h-polynomial

Calling Sequence:

`Toh_n`(sf)
`Toh_n`(sf, b)
`GL[Toh_n]`(sf)
`GL[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 `GL['Toh_n']`.

Examples:

```
> with(GL):
> _GLn;

                                  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`

5.9.7 Top_n — convert any symmetric function to a p-polynomial

Calling Sequence:

`Top_n(`sf`)`
`Top_n(`sf, b`)`
`GL[Top_n](`sf`)`
`GL[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 `GL['Top_n']`.

Examples:

```
> with(GL):
> _GLn;

                                    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`

5.9.8 Tos_n — express any symmetric function in the basis of Schur functions

Calling Sequence:

Tos_n(sf)
Tos_n(sf, b)
GL[Tos_n](sf)
GL[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 GL['Tos_n'].

Examples:

```
> with(GL):
> _GLn;

                              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`

5.9.9 `Tox_n` — express any symmetric function in the basis of monomials

Calling Sequence:

`Tox_n`(sf)
`Tox_n`(sf, b)
`GL[Tox_n]`(sf)
`GL[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 `GL['Tox_n']`.

Examples:

```
> with(GL):
> _GLn;

                              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]

5.9.10 e2x_n — from e-basis to the basis of monomials

Calling Sequence:

e2x_n(sf)
GL[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 GL['e2x_n'].

Examples:

```
> with(GL):
> GL_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:

5.9.11 h2x_n — from h-basis to the basis of monomials

Calling Sequence:

h2x_n(sf)
GL[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 GL['h2x_n'].

Examples:

```
> with(GL):
> GL_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:

5.9.12 m2x_n — from monomial symmetric function basis to the basis of monomials

Calling Sequence:

m2x_n(sf)
GL[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 `GL['m2x_n']`.

Examples:

```
> with(GL):
> GL_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:

5.9.13 p2x_n — from p-basis to the basis of monomials

Calling Sequence:

`p2x_n(`sf`)`
`GL[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 `GL['p2x_n']`.

Examples:

```
> with(GL):
> GL_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:

5.9.14 s2x_n — from Schur function basis to the basis of monomials

Calling Sequence:

s2x_n(sf)
GL[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 GL['s2x_n'].

Examples:

```
> with(GL):
> GL_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:

5.9.15 x2m_n — from the basis of monomials to monomial symmetric function basis

Calling Sequence:

`x2m_n`(*pol*)
`GL[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 `GL['x2m_n']`.

Examples:

```
> with(GL):
> 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]`

5.10 Non-commutative symmetric functions via NCSF

Calling Sequence:

NCSF[<function>](*args*)
<function>(*args*)

Synopsis:

- This package provides functions to work with noncommutative 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	Internal	IsFiner	L2PhMat
L2PsMat	L2RMat	L2SMat	ListCompo
ListCompoFiner	MajorIndex	Ph2LMat	Ph2PsMat
Ph2RMat	Ph2SMat	Ps2LMat	Ps2PhMat
Ps2RMat	Ps2SMat	R2LMat	R2PhMat
R2PsMat	R2SMat	S2LMat	S2PhMat
S2PsMat	S2RMat	SpDirect	SpInverse
ToL	ToPh	ToPs	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 ]
```

5.10.1 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:

5.10.2 `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 non-negative 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:

5.10.3 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

5.10.4 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.5 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 (elementary symmetric functions) to the Ψ-basis (power sums symmetric functions of the first kind).
- The result is a $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.6 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 (elementary symmetric functions) to the R-basis (ribbon Schur functions).
- The result is a $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.7 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 (elementary symmetric functions) to the S-basis (complete symmetric functions).
- The result is a $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.8 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:

5.10.9 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:

5.10.10 `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:

5.10.11 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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`

5.10.12 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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`

5.10.13 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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`

5.10.14 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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`

5.10.15 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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`

5.10.16 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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`

5.10.17 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.18 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.19 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.20 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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`

5.10.21 `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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.22 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.23 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.24 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.25 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.26 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 $2^{(2n-2)} \times 2^{(2n-2)}$ 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

5.10.27 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

5.10.28 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 $S^n((1-q)\,A) = (1-q)(-q)^k \sum_{k=0}^{n-1} R_{1^k,n-k}$.
- 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`

5.10.29 `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 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

5.10.30 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 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

5.10.31 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 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

5.10.32 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 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`

5.10.33 `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 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

5.11 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	ListStdTab	ListYama	Mat2CTab
Mat2Tab	Part2ListStdTab	PermOnWord	StdTab2Idemp
Tab2Mat	Tab2Part	Tab2Word	Word2Charge
Word2CoCharge	Word2CTab	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 ]
```

5.11.1 `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`

5.11.2 `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

5.11.3 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]

5.11.4 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

5.11.5 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

5.11.6 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.
- 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(4));
```

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

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

SEE ALSO: `Part2ListStdTab`, `TYP[Tableau]`

5.11.7 `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 $\ldots 3^{part[3]} 2^{part[2]} 1^{part[1]}$ or, equivalently, give the corresponding tableaux $[\ldots,[3\ldots 3],[2\ldots 2],[1\ldots 1]]$ by Schensted algorithm.
- 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]`

5.11.8 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]`

5.11.9 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]

5.11.10 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([2,1,1]);

       [[[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]

5.11.11 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`

5.11.12 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: $\texttt{StdTab2Idemp}(t) = \texttt{StdTab2Idemp}(t') \times \prod(\texttt{Jucis}(n,n) - c(n,t_i))$ where $\texttt{Jucis}(n,n)$ 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]`

5.11.13 `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]`

5.11.14 `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]`

5.11.15 `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`

5.11.16 `Word2Charge` — returns the charge vector 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 vector of the word w that is obtained by repeated circular reading of w.
- The word should contain at least as many 1's as 2's, as many 2's as 3's, and so on.
- When called with the second argument `'nb'`, it returns the numerical charge of w, *i.e.* the sum of the components of the charge vector.

- 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]);

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

> Word2Charge([3,2,2,1,4,1,1,3], 'nb');

                                   2
```

SEE ALSO: `Word2CoCharge`

5.11.17 `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`

5.11.18 `Word2CoCharge` — returns the cocharge vector 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 vector of the word w by repeated circular reading of the word.
- The word should contain at least as many 1's as 2's, as many 2's as 3's and so on.
- When called with the second argument `'nb'`, it returns the numerical cocharge of w, *i.e.* the sum of the components of the cocharge vector of the word w.
- 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]);

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

> Word2CoCharge([3,2,2,1,4,1,1,3], 'nb');

                               7
```

SEE ALSO: `Word2Charge`

5.11.19 `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`

5.11.20 `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]`

5.12 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	Specialize	TableX
TableXX	TableY	TableYY	Tox
ToX	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)
               ])
```

5.12.1 `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`

5.12.2 `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`

5.12.3 `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`

5.12.4 `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:

5.12.5 `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 callled 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

5.12.6 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

5.12.7 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:

5.12.8 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`

5.12.9 `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
                              ])
```

```
> t[ [2,3,1] ];
```

```
                                   x2 x1
```

SEE ALSO: `TableXX`

5.12.10 `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`

5.12.11 `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):
> t:=TableY(3);
                  t := 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
                      ])

> t[ [2,1,0] ];
```

```
  2
x1  x2
```

SEE ALSO: TableYY

5.12.12 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`

5.12.13 `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`

5.12.14 `ToY` — convert any expression to the 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`

5.12.15 `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`

5.12.16 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

5.12.17 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`

5.12.18 `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

5.12.19 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:

5.12.20 Y2X — from Y Schubert basis to 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

5.12.21 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

5.12.22 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

5.12.23 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:

5.12.24 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`

5.12.25 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`

5.13 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.

- 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]

5.13.1 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

5.13.2 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

5.13.3 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`

5.13.4 `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`

5.13.5 `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`

5.13.6 `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:
```

5.13.7 `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]`

5.13.8 `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]`

5.13.9 `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 `h1` or `h2` for instance, the function `TeX` will simply call `` `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]`

5.13.10 `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]`

5.14 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:

IsCTab	IsCompo	IsPart	IsPe
IsPerm	IsStdCTab	IsStdTab	IsTab
IsWord	IsX	IsXX	IsY
IsYY	IsYama	Iscl	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
```

5.14.1 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`

5.14.2 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`

5.14.3 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.

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`

5.14.4 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.

- $\begin{matrix} 2 & 5 & 7 \\ 1 & 4 & 4 \\ & & 3 \end{matrix}$ 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`

5.14.5 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]`

5.14.6 `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`

5.14.7 IsPe — test whether the argument is an element of the $\mathcal{P}e$-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 $\mathcal{P}e_I$ 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:

5.14.8 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`

5.14.9 `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`

5.14.10 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`

5.14.11 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

> IsYY(YY[0,-1]);

                    false
```

SEE ALSO: `IsX, IsXX, IsY`

5.14.12 `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`

5.14.13 `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

5.14.14 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:

5.14.15 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]);
```

$$\mathtt{true}$$

SEE ALSO: `IsPart`

5.14.16 `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);
```

$$\mathtt{true}$$

```
> Ise(e12);
```

$$\mathtt{true}$$

SEE ALSO: `Ish, Isp, Iss, Sf`

5.14.17 `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: `Ise, Isp, Iss, Sf`

5.14.18 `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: `Ise`, `Ish`, `Iss`, `Sf`

5.14.19 `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: `Ise`, `Ish`, `Isp`, `Sf`

5.14.20 `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

5.14.21 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`

5.14.22 `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`

5.14.23 `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`

5.14.24 `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:

5.14.25 `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`

5.15 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:

GLPerror	HEKAPerror	HEQAPerror	IDCAPerror
NCAPerror	NCSFPerror	SGAPerror	SGPerror
SPPerror	SYMFPerror	TABPerror	TEXPerror

About…

NAME	`ACE`	Algebraic Combinatorics Environment.
VERSION	1.0 (1995)	The development started in 1994.
MAIN CONTACT	Sébastien Veigneau	`Email: veigneau@univ-mlv.fr`
AVAILABLE FOR	Maple V	Release 1, 2 and 3.
SYSTEM	UNIX and MS-DOS	From MS-DOS 3.3.
FORMAT	"*gzip*ed" *tar* file	UNIX release of *gunzip* and *tar* programs needed (provided for MS-DOS).
MS-DOS	`ACE` $\boxed{\begin{matrix}1\\2\\3\end{matrix}}$ `DOS.tgz`	*gzip***ed file size** : ~650Kb **Tar file size** : ~4.1Mb **Before compilation** : ~10Mb **After compilation** : ~12Mb
UNIX	`ACE` $\boxed{\begin{matrix}1\\2\\3\end{matrix}}$ `UNIX.tgz`	*gzip***ed file size** : ~730Kb **Tar file size** : ~4.5Mb **Before compilation** : ~4.2Mb **After compilation** : ~5.8Mb
`DVI` DOC.	`manual.dvi`	**File size** : ~700Kb **Number of pages** : ~300
POSTSCRIPT DOC.	`manual.ps`	**File size** : ~1.5Mb **Number of pages** : ~300
TEXT DOC.	`*.txt`	**Number of files** : ~280

Contacts...

Package	Author	Email
ACE	Sébastien Veigneau	veigneau@univ-mlv.fr
ERR	Sébastien Veigneau	veigneau@univ-mlv.fr
GL	Sébastien Veigneau	veigneau@univ-mlv.fr
HEKA	Sébastien Veigneau	veigneau@univ-mlv.fr
HEQA	Éric Laugerotte	laugerot@dir.univ-rouen.fr
IDCA	Sébastien Veigneau	veigneau@univ-mlv.fr
NCA	Sébastien Veigneau	veigneau@univ-mlv.fr
NCSF	Bun-Chan-Vorac Ung	ung@univ-mlv.fr
SG	Sébastien Veigneau	veigneau@univ-mlv.fr
SGA	Sébastien Veigneau	veigneau@univ-mlv.fr
SP	Sébastien Veigneau	veigneau@univ-mlv.fr
SYMF	Sébastien Veigneau	veigneau@univ-mlv.fr
TAB	Sébastien Veigneau	veigneau@univ-mlv.fr
TYP	Sébastien Veigneau	veigneau@univ-mlv.fr
TEX	Sébastien Veigneau	veigneau@univ-mlv.fr

Bibliography

[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) (1995), to appear.

[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.

Noncommutative symmetric functions

[4] F.BERGERON and D.KROB, *Acyclic complexes related to noncommutative symmetric functions*, preprint, 1995.

[5] G.DUCHAMP, D.KROB, B.LECLERC and J.-Y.THIBON, *Déformations de projecteurs de Lie*, C.R. Acad. Sci. Paris **319** (1994), 909-914.

[6] 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.

[7] I.M.GELFAND and V.S.RETAKH, *Determinants of matrices over noncommutative rings*, Funct. Anal. Appl. **25**, (1991), 91-102.

[8] 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.

[9] I.GESSEL, *Multipartite P-partitions and inner product of skew Schur functions*, Contemporary Math. **34** (1984), 289-301.

[10] A.A.KLYACHKO, *Lie elements in the tensor algebra*, Siberian Math. J. **15** (1974), 1296-1304.

[11] D.KROB, B.LECLERC and J.-Y.THIBON, *Noncommutative symmetric functions II: Transformations of alphabets*, preprint, 1995.

[12] P.A.MacMahon, *Combinatory analysis*, Cambridge University Press, 1915, 1916; Chelsea reprint, 1960.

[13] C.Malvenuto and C.Reutenauer, *Duality between quasi-symmetric functions and the Solomon descent algebra*, preprint UQAM, Montréal, 1993.

[14] B.Mielnik and J.Plebański, *Combinatorial approach to Baker-Campbell-Hausdorff exponents*, Ann. Inst. Henri Poincaré, Section A vol. **XII** (1970), 215-254.

[15] C.Reutenauer, *Free Lie algebras*, Oxford University Press, 1993.

[16] B.Leclerc, T.Scharf and J.-Y.Thibon, *Noncommutative cyclic characters of symmetric groups*, preprint, 1994.

[17] L.Solomon, *A Mackey formula in the group ring of a Coxeter group*, J. Algebra **41** (1976), 255-268.

Schubert polynomials

[18] 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.

[19] M.Demazure, *Désingularisation des variétés de Schubert*, Annales E.N.S. **6** (1974), 163-172.

[20] W.Fulton, *Flags, Schubert polynomials, degeneracy loci and determinantal formulas*, Duke Math.J. **65** (1991), 381-420.

[21] F.Hirzebruch, *Topological Methods in Algebraic Geometry*, 3_{rd} ed., Springer-Verlag, 1978.

[22] A.Kohnert, *Weintrauben, Polynome, Tableaux*, Bayreuther Math. Schr. **38** (1991), 1-97.

[23] A.Kohnert and S.Veigneau, *Using Schubert toolkit to compute with polynomials in several variables*, preprint, 1995.

[24] A.Lascoux, *Polynômes de Schubert. Une approche historique*, Actes du 4^e colloque "Séries formelles et combinatoire algébrique", Montréal 1992, Publ. LACIM, UQAM, 1992.

[25] A.Lascoux and M.P.Schützenberger, *Schubert polynomials and the Littlewood-Richardson Rule*, Lett. Math. Phys. **10** (1985), 111-124.

[26] A.Lascoux and M.P.Schützenberger, *Symmetry and Flag manifolds*, Springer L.N. **996** (1983), 118-144.

[27] A.Lascoux, *Classes de Chern des variétés de drapeaux*, C.R. Acad. Sci. Paris **295** (1982), 393-398.

[28] A.Lascoux and M.P.Schützenberger, *Polynômes de Schubert*, C.R. Acad. Sci. Paris **294** (1982), 447-450.

[29] I.G.Macdonald, *Notes on Schubert polynomials*, Publ. LACIM **6**, UQAM, Montréal, 1991.

[30] D.Monk, *the Geometry of flags manifolds*, Proc. London M.S. (3)**9** (1959), 253-286.

Symmetric functions

[31] I.G.Macdonald, *Symmetric functions and Hall polynomials*, $2_{\rm nd}$ ed., Oxford: Clarendon Press, 1995.

[32] D.Knutson, *λ-rings and the representation theory of the symmetric group*, Springer L.N.M. **308**, 1973.

[33] D.E.Littlewood, *The theory of group characters*, $2_{\rm nd}$ ed., Oxford, 1950.

[34] J.Stembridge, *SF, a maple package for symmetric functions*, University of Michigan, 1993.

Tableaux

[35] D.E.Knuth, *Permutations, matrices and generalized young tableaux*, Pacific J. Math. **34** (1970), 709-727.

[36] A.Lascoux and M.P.Schützenberger, *Keys and standard bases, in invariant theory and tableaux*, D. Stanton ed., Springer-Verlag (1990), 125-144.

[37] A.Lascoux and M.P.Schützenberger, *Le monoïde plaxique*, Quaderni de la Ricerca Scientifica, **109** (1981), 129-156.

[38] C.Schensted, *Longest increasing and decreasing subsequences*, Canad. J. Math. **13** (1961), 179-191.

Index