ACE

an Algebraic Combinatorics Environment

for the computer algebra system MAPLE

User's Reference Manual
Version 3.0

Sébastien Veigneau

Université de Marne-la-Vallée

Institut Gaspard Monge

June 2, 1998

Université de
Marne-la-Vallée
Institut Gaspard Monge

ACE

an Algebraic Combinatorics Environment
for the computer algebra system MAPLE

User's Reference Manual

Version 3.0

ACE

Version 3.0

ACE is a joint effort of a number of people from the University of Marne-la-Vallée. The project has been led since 1994 by Sébastien Veigneau (University of Marne-la-Vallée) who is grateful to all people having made up the version 3.0 of `ACE`. All the development has been done under supervision of Alain Lascoux (University of Marne-la-Vallée, CNRS) and Jean-Yves Thibon (University of Marne-la-Vallée). Many thanks also to Vincent Prosper for providing the necessary TEX and Perl knowledge to produce this documentation. The following persons have mainly contributed to the `ACE` project:

Sébastien Veigneau

Institut Gaspard Monge,
Université de Marne-la-Vallée,
Cité Descartes,
5, boulevard Descartes,
Champs-sur-Marne,
77454 Marne-la-Vallée, Cedex 2, France.
`Email: Sebastien.Veigneau@univ-mlv.fr`
`Url: http://phalanstere.univ-mlv.fr/~sv`

Alain Lascoux

Institut Gaspard Monge,
Université de Marne-la-Vallée,
Cité Descartes,
5, boulevard Descartes,
Champs-sur-Marne,
77454 Marne-la-Vallée, Cedex 2, France.
`Email: Alain.Lascoux@univ-mlv.fr`
`Url: http://phalanstere.univ-mlv.fr/~al`

Jean-Yves Thibon

Institut Gaspard Monge,
Université de Marne-la-Vallée,
Cité Descartes,
5, boulevard Descartes,
Champs-sur-Marne,
77454 Marne-la-Vallée, Cedex 2, France.
Email: Jean-Yves.Thibon@univ-mlv.fr
Url: http://www-igm.univ-mlv.fr/~jyt

Vincent Prosper

Institut Gaspard Monge,
Université de Marne-la-Vallée,
Cité Descartes,
5, boulevard Descartes,
Champs-sur-Marne,
77454 Marne-la-Vallée, Cedex 2, France.
Email: Vincent.Prosper@univ-mlv.fr
Url: http://phalanstere.univ-mlv.fr/~vince

Bun-Chan-Vorac Ung

Institut Gaspard Monge,
Université de Marne-la-Vallée,
Cité Descartes,
5, boulevard Descartes,
Champs-sur-Marne,
77454 Marne-la-Vallée, Cedex 2, France.
Email: Ung@univ-mlv.fr
Url: http://phalanstere.univ-mlv.fr/~ung

Copyright notice

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

New York Herald Tribune, October 30, 1956.

ACE is copyright © 1994-1998 by Sébastien Veigneau, Institut Gaspard Monge, Université de Marne-la-Vallée, Cité Descartes, 5, boulevard Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée, Cedex 2, France. Permission is granted to anyone to use, modify, or redistribute this software freely, subject to the following restrictions:

- The author accepts no responsibility for any consequences of this software and makes no guarantee that the software is free of defects.
- The origin of this software must not be misrepresented, either by explicit claim or by omission.
- This notice and the copyright must be included in all copies or altered versions of this software.
- This software may not be included or redistributed as part of any package to be sold for profit without the explicit written permission of the author.

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

Université de Marne-la-Vallée
Institut Gaspard Monge

New features of release 3.0

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

H. Poincaré.

Users already familiar with `ACE` may be interested in the differences between versions 2.0 and 3.0. New users of `ACE` may skip this part and proceed with the next chapter. In the previous documentation, new features of release 2.0 with respect to version 1.0 of `ACE` were identified by a ✘ in the margin. This is not the case for this documentation because many changes have been ✘ made. New features of release 3.0 include:

- **New package** `COMP`: this package provides functions to work with compositions. Available functions deal with comparisons, major indexes, conjugates, lists of compositions.
- **New package** `FREE`: this package provides functions to work with the free algebra, that is the algebra of words. It is also possible to work in the plactic monoid, *i.e.* the free algebra of words modulo the plactic relations.
- **New package** `PART`: this package provides functions to work with partitions. Available functions deal with border, conjugate, diagonals, Frobenius notation, hooks, etc. of a partition, and also with lists of (skew) partitions.
- **New package** `SFA`: this is an extension of the `SYMF` package which gives a way to handle symmetric functions viewed as operators acting on the ring of polynomials with real coefficients. It corresponds to an implementation of the Λ-ring theory.
- **New possibilities** in the `SYMF` package: it is from now possible to add new bases of the ring of symmetric functions, by the mean of two new functions `SfAddBasis` and `SfDualBasis`.
- **Many function names** have been changed, many functions have disappeared, either becoming internal functions or being included as options of other more general ones. Function names have been strictly normalized.
- **New functions and algorithms** have been included in already existing packages in order to improve their efficiency. This is in particular the case for the `SG` (symmetric group), the `TAB` (tableaux) and `SYMF` (symmetric functions) packages.

- **Heavy modifications** in the coding of the elements of the different algebras implemented in `ACE`. Elements are now coded as sums of indexed names instead of sums of lists for some compatibility reasons with *Maple V.*4.
- **Heavy internal modifications** in the `CG` (classical groups) package have been performed in order to get it more efficient.
- **ACE installation** has been completely rebuilt in order to use new possibilities with respect to the development of packages, in particular in *Maple V*.4.
- **A test protocol** has been included in `ACE` in order to have the possibility to test whether the installation has been successful or not. It corresponds to the automatic production of test files (including more than 1000 tests) that you should read to test a specific package in `ACE`.
- **Removed package** `SHRI`: this package has been removed from the `ACE` system.
- **Removed package** `FOCK`: this package has been removed from the core of the `ACE` system. However, it is still available as a satellite package using `ACE` functionalities.
- **Some bugs** have been fixed.
- **A new documentation** is included in `ACE` 3.0, available both in postscript format and in the HTML hypertext language.
- **The full system** is available on CD-ROM, including related papers (articles, theses), a mathematical manual explaining many algorithms used in `ACE` and how to experiment with `ACE`, the HTML and postscript documentations. It also contains all sources, all installed (compiled) versions of `ACE` 3.0, together with the old versions of `ACE`.
- **Informations** related to the `ACE` CD-ROM can be obtained from:

  `http://phalanstere.univ-mlv.fr/~ace`

  or by email at:

  `ACE@univ-mlv.fr`.

  In any case, you can contact Alain Lascoux directly by email at:

  `Alain.Lascoux@univ-mlv.fr`.

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 in July 1994. We made up our minds to develop a more general project about algebraic combinatorics in October 1994. The aim was to build an "easy to use" Algebraic Combinatorics Environment. We have chosen *Maple* because this system is widely used by combinatorists and has a convenient user's interface. We had in mind to develop an efficient toolbox of functions for research in algebraic combinatorics. The interface had to be very flexible for first experimentations, whereas it may be more interesting to use some specific programs for large scale computations: for instance, we refer the reader to the `SYMMETRICA` system that is well suited for such computations [3].

We also refer the reader to the new system μ-EC (`http://phalanstere.univ-mlv.fr/~muec`) developped by Vincent Prosper (University of Marne-la-Vallée) which includes the `SYMMETRICA` system (to compute with symmetric functions) and can be seen as an environment of computation in combinatorics for the computer algebra system `MuPAD`. The μ-EC system includes many functions available in `ACE`.

The story started in 1994. `ACE` 1.0 was divided into various packages dealing with specific objects such as noncommutative symmetric functions, Schubert polynomials, tableaux or Hecke algebra, each package having the ability to use some functions defined in the other packages. The first version of `ACE` was available for any MS-DOS or UNIX system, for *Maple V* Release 1, 2 and 3.

The next version 2.0 was available since June 1996 for any MS-DOS, UNIX and MACINTOSH system. It provided new packages and new functions in the already existing packages. Several people had been involved in the project in order to develop new programs as for instance to work with the hyperoctahedral group and with several algebras associated to the hyperoctahedral groups B_n, together with their realizations as algebras of operators on the ring of polynomials. Moreover, a package has been developed to work with the representation theory of classical Lie groups (symplectic and orthogonal groups), whereas a new package was devoted to computations in the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ regarded as a free module of rank $n!$ over the ring $\mathbf{Sym}[x_1, \ldots, x_n]$ of symmetric polynomials, and another one was devoted to calculations in the Fock space representations of the quantum affine algebras $U_q(\widehat{sl}_n)$.

The third version of `ACE`, that is `ACE` 3.0, has been released at the beginning of 1998. Many changes have been made in order to get a coherent system devoted to computations in algebraic combinatorics. Many function names have been changed, many functions have disappeared, either becoming internal functions or being included as options of other more general ones. Function names have been strictly normalized.

`ACE` 3.0 has thus less functions but more functionalities. In particular, several packages have been included such as two packages to handle partitions and compositions of integers, one package to work in the free algebra, one package dealing with symmetric functions on different alphabets.

The ACE system, version 3.0, is now available for *Maple V* Release 1, 2, 3 and 4, for any MS-DOS, UNIX or MACINTOSH system.

This documentation describes how this new version of ACE can be installed and how the new user can make his first steps towards an efficient use of the environment. It gives a complete documentation about all mathematical available functions.

I hope this system will be useful for further developments in computer science in relation with algebraic combinatorics. I would like to thank you for your interest in ACE, "*quant à moi, je vais désormais voler vers d'autres horizons...*".

Sébastien Veigneau,
Université de Marne-la-Vallée, June 2, 1998.

```
############################################################
# Algebraic Combinatorics Environment,  3.0,  June 2, 1998 #
############################################################
# Alain Lascoux, Sebastien Veigneau & Co                   #
# Institut Gaspard Monge, Universite de Marne-la-Vallee,   #
# Cite Descartes, 5 Boulevard Descartes, Champs-sur-Marne, #
# 77454, Marne-la-Vallee, Cedex 2, FRANCE.                 #
# Email: ACE@univ-mlv.fr                                   #
# URL: http://phalanstere.univ-mlv.fr/~ace                 #
#####################################################
# ACE  Algebraic Combinatorics Environment         #
#                                                  #
# SFA  Symmetric Functions on different Alphabets  #
# FM   Free Module over symmetric polynomials      #
# NCSF NonCommutative Symmetric  Functions         #
# CLG  Characters of Linear Groups                 #
# SP   Schubert Polynomials                        #
# SYMF SYMmetric Functions                         #  __/__/ __/_/_/ __/_/__/
# TAB  TABleaux                                    #  __/ __/ __/     __/
# FREE FREE algebra                                #  __/_/__/ __/     __/_/
# BNA  hyperoctahedral Algebras                    #  __/   __/ __/     __/
# SGA  Symmetric Group Algebra                     #  __/    __/ __/_/_/ __/_/__/
# NCA  NilCoxeter Algebra                          #
# IDCA IDCoxeter Algebra                           #               _/_/_/    _/_/_/
# HEKA HEcKe Algebra                               #                _/_/    _/  _/
#                                                  #           _/_/_/ _/ _/_/_/
# BN   hyperoctahedral groups                      #
# SG   Symmetric Groups                            #
# CG   Classical Groups                            #
#                                                  #
# PART PARTitions                                  #
# COMP COMPositions                                #
#                                                  #
# ERR  ERRors                                      #
# TEX  TEX output                                  #
# TYP  TYPe checking                               #
#######################
```

Table of contents

Acknowledgments i

Copyright notice iii

New features of Release 3.0 v

Preface vii

1 Overview 1
- 1.1 The BN library . . . 2
- 1.2 The BNA library . . . 2
- 1.3 The CG library . . . 3
- 1.4 The CLG library . . . 4
- 1.5 The COMP library . . . 5
- 1.6 The FM library . . . 5
- 1.7 The FREE library . . . 6
- 1.8 The HEKA library . . . 8
- 1.9 The IDCA library . . . 9
- 1.10 The NCA library . . . 9
- 1.11 The NCSF library . . . 10
- 1.12 The PART library . . . 11
- 1.13 The SFA library . . . 12
- 1.14 The SG library . . . 13
- 1.15 The SGA library . . . 14
- 1.16 The SP library . . . 15
- 1.17 The SYMF library . . . 17
- 1.18 The TAB library . . . 19
- 1.19 The TEX library . . . 21

2 Background 27
- 2.1 Symmetric group . . . 27
- 2.2 Symmetric group algebra . . . 31
- 2.3 NilCoxeter algebra . . . 32
- 2.4 IdCoxeter algebra . . . 33
- 2.5 Generic Hecke algebra . . . 34
- 2.6 Characters of classical Lie groups . . . 35
  - 2.6.1 Dimension of representations . . . 38

2.6.2 Product of characters . . . 39
2.6.3 Freudenthal's multiplicity formula . . . 40
2.7 Symmetric functions . . . 40
2.7.1 The structure of Λ-ring . . . 44
2.7.2 Alphabets . . . 46
2.7.3 Expanding the action of a symmetric function over $\mathbb{R}[\mathbf{x}]$ . . . 46
2.8 Noncommutative symmetric functions . . . 48
2.8.1 Change of basis . . . 50
2.8.2 Descent algebras . . . 51
2.8.3 Lie idempotents . . . 52
2.8.4 Quasi-symmetric functions . . . 55
2.8.5 Pairing between **Sym** and **Qsym** . . . 55
2.9 Hyperoctahedral groups . . . 58
2.10 Algebras related to hyperoctahedral groups . . . 60
2.11 Words and tableaux . . . 62
2.12 Schubert polynomials . . . 66
2.12.1 Symmetric groups and divided difference operators . . . 67
2.12.2 Schubert polynomials . . . 69
2.12.3 Double Schubert polynomials . . . 73
2.12.4 Examples from algebraic geometry . . . 75
2.13 Computations in the ring of polynomials in several variables . . . 77
2.13.1 Schubert bases . . . 78
2.13.2 Multiplicative structure . . . 79
2.13.3 Applications . . . 80
2.14 Fock space representation of $U_q(\widehat{sl}_n)$ . . . 83
2.14.1 The quantum affine algebra $U_q(\widehat{sl}_n)$ . . . 83
2.14.2 Fock space representation of $U_q(\widehat{sl}_n)$ . . . 84
2.14.3 Crystal graph of the Fock space . . . 87
2.14.4 Canonical basis of the basic representation . . . 89
2.14.5 Kashiwara-Miwa-Stern algebra and ribbon tableaux . . . 90
2.14.6 Canonical basis of the Fock space representation . . . 91
2.14.7 Application: decomposition matrices of Hecke and q-Schur algebras . . . 93

3 Installation **95**
3.1 Installing `ACE` on a UNIX system . . . 95
3.1.1 Restoring sources . . . 96
3.1.2 Description of the `ACE_PACK` directory . . . 96
3.1.3 Description of a typical package . . . 96
3.1.4 Configuration file . . . 97
3.1.5 Having some troubles . . . 99
3.1.6 Using the environment . . . 99
3.2 Installing `ACE` on a MS-DOS system . . . 100
3.2.1 Restoring sources . . . 100
3.2.2 Description of the `ACE_PACK` directory . . . 100
3.2.3 Description of a typical package . . . 101
3.2.4 Configuration file . . . 101
3.2.5 Using the environment . . . 103
3.3 Installing `ACE` on a MACINTOSH system . . . 103
3.3.1 Restoring sources . . . 104

3.3.2 Description of the ACE_PACK folder . . . 104
3.3.3 Description of a typical package . . . 104
3.3.4 Configuration file . . . 104
3.3.5 Using the environment . . . 106

4 Examples **107**
4.1 Grothendieck polynomials . . . 107
4.2 Key polynomials . . . 110
4.3 Projective degree and Todd class . . . 112
4.3.1 Projective degree of a Schubert variety . . . 112
4.3.2 Todd class of the flag variety . . . 113
4.4 Some computations with NCSF . . . 114
4.5 Generating functions for Schubert polynomials ($\mathfrak{S}_n$ and B_n) . . . 117
4.6 Divided differences on modified powers . . . 120
4.6.1 Complete Functions of Alphabets . . . 120
4.6.2 Action of Divided Differences on Modified Powers . . . 120
4.6.3 q-specialization . . . 122
4.6.4 $\mathbb{N}$-specialization: generalized Stirling numbers . . . 123

5 Main routines **127**
5.1 About the ACE package . . . 128
AceBanner - print a banner about ACE . . . 129
AceBuildData - build and store data . . . 130
AceLoadData - load data from disk . . . 131
AceSysInfo - give information on the system . . . 132
AceTime - give time information . . . 133
AceUnloadData - unload data . . . 134
AceWhichIs - look for a function in a list of packages . . . 134
5.2 PART, a package to handle partitions . . . 136
Border2Part - partition associated to a given border . . . 136
CompPart - compare two partitions . . . 137
Diagonal2Part - partition from a list of diagonals . . . 138
Exp2Part - partition from its exponential notation . . . 139
Frob2Part - from Frobenius notation to standard notation for partitions . . . 139
ListPart - list of partitions of a given weight . . . 140
ListPartIn - list of partitions contained into a given partition or between two given partitions . . . 141
ListSkewDiag - list of skew diagrams . . . 142
PCore2Part - partition from p-core and p-quotient . . . 143
Part2Border - border of a given partition . . . 144
Part2Conjugate - conjugate a partition . . . 145
Part2Diagonal - diagonals of the given partition . . . 145
Part2Exp - multiplicities of a given partition . . . 146
Part2Frob - from standard partition notation to Frobenius notation . . . 147
Part2ListHook - calculate the list of hook lengths . . . 147
Part2Mat - a planar representation of a given partition . . . 148
Part2PCore - p-core and p-quotient of a partition . . . 149
PartOrderMat - order matrix on partitions . . . 150
SkewPart2Mat - a planar representation of a skew partition . . . 152

SplitPart - split a partition into several ones . . . 153
5.3 COMP, a package to handle compositions . . . 154
CompCompo - compare two compositions . . . 154
Compo2Conjugate - conjugate a composition . . . 155
Compo2MajorIndex - major index of a given composition . . . 156
Compo2Mat - a planar representation of a composition . . . 156
ListCompo - list of compositions of a given weight . . . 157
ListCompoFatter - all compositions fatter than a given one . . . 158
ListCompoFiner - all compositions finer that a given one . . . 159
5.4 SG, a package for the symmetric group . . . 160
Code2Perm - calculate a permutation from its code . . . 161
CompPerm - compare two permutations . . . 161
Cycle2Perm - calculate the permutation from its decomposition into disjoint cycles . . . 162
Desc2ListPerm - generate all permutations with a given descent set . . . 163
FactorPerm - test whether a permutation factorizes into a direct product . . . 164
GenPerm - generate an element of the symmetric group . . . 164
ListPerm - generate all permutations . . . 165
Mat2Perm - calculate a permutation from its matrix of permutation . . . 167
MultPerm - multiply permutations . . . 167
Part2SgCC - compute a conjugacy class of the symmetric group . . . 168
Perm2Betti - compute the Betti polynomial . . . 169
Perm2Code - calculate the code of a permutation . . . 169
Perm2Cycle - decompose a permutation into disjoint cycles . . . 170
Perm2CycleType - compute the cycle type of a permutation . . . 171
Perm2Desc - compute the list of descents of a permutation . . . 171
Perm2Interval - compute all codes of permutations below a given one . . . 172
Perm2Inv - invert a permutation . . . 173
Perm2Length - calculate the length of a permutation . . . 173
Perm2ListInv - calculate the set of inversions of a permutation . . . 174
Perm2ListRd - list all reduced decompositions of a given permutation . . . 175
Perm2Mat - calculate the permutation matrix . . . 175
Perm2RRep - calculate the matrix representing a permutation in the regular representation 176
Perm2Rd - calculate a canonical reduced decomposition of a permutation . . . 177
Perm2Rise - compute the list of rises of a permutation . . . 177
Perm2Rothe - calculate the Rothe diagram of a permutation . . . 178
Perm2Sign - return the signature of a permutation . . . 179
RRep2Perm - calculate a permutation from its matrix in the regular representation . . . 180
RandPerm - return a random permutation . . . 181
Rd2Perm - calculate a permutation from a decomposition . . . 181
SgCharTable - compute the table of characters . . . 182
SgDimRep - compute the dimension of a representation . . . 183
SgTranspo - return a transposition . . . 184
Vect2Perm - compute a canonical permutation from a vector . . . 185
5.5 Using SGA to work with the symmetric group algebra . . . 186
Part2SgaCIdemp - compute a central idempotent of the symmetric group algebra . . . 187
Rd2Sga - calculate a product of generators . . . 187
Sga2Carre - Carré element of the symmetric group algebra . . . 188
Sga2Nabla - Nabla element of the symmetric group algebra . . . 189
Sga2Table - convert an element of the symmetric group algebra into a table . . . 190

SgaAdd - sum of two elements of the symmetric group algebra . . . 191
SgaJucis - Jucis-Murphy element of the symmetric group algebra . . . 191
SgaMinus - difference of two elements of the symmetric group algebra . . . 192
SgaMult - multiplication of two elements of the symmetric group algebra . . . 193
SgaNormal - reduce each permutation inside an expression to the minimum symmetric group containing it . . . 194
SgaOnPol - action of an element of the algebra on an expression . . . 195
SgaOnX - action of an element of the symmetric group algebra on a linear combination of X[perm] . . . 195
SgaOnXfix - action of an element of the symmetric group algebra on the ring of polynomials as a free module over **Sym** . . . 196
SgaYang - compute a special element of the symmetric group algebra . . . 198
Table2Sga - convert a table into an element of the symmetric group algebra . . . 198
5.6 NCA to compute in the nilCoxeter algebra . . . 200
Nca2Table - convert an element of the nilCoxeter algebra into a table . . . 201
NcaAdd - sum of two elements of the nilCoxeter algebra . . . 201
NcaMinus - difference of two elements of the nilCoxeter algebra . . . 202
NcaMult - multiplication of two elements of the nilCoxeter algebra . . . 203
NcaNormal - normalize an element of the nilCoxeter algebra . . . 204
NcaOnPol - action of an element of the algebra on a polynomial . . . 204
NcaOnX - action of an element of the nilCoxeter algebra on a linear combination of X[perm] . . . 205
NcaOnXX - action of an element of the nilCoxeter algebra on a linear combination of XX[perm] . . . 206
NcaOnXfix - action of an element of the nilCoxeter algebra on the ring of polynomials as a free module over **Sym** . . . 207
NcaOnY - action of an element of the nilCoxeter algebra on a linear combination of Y[code] . . . 208
NcaOnYY - action of an element of the nilCoxeter algebra on a linear combination of YY[code] . . . 209
NcaYang - compute a special element of the nilCoxeter algebra . . . 210
Rd2Nca - calculate a product of generators . . . 211
Table2Nca - convert a table into an element of the nilCoxeter algebra . . . 212
5.7 IDCA to compute in the idCoxeter algebra . . . 213
Idca2Demazure - Demazure element of the idCoxeter algebra . . . 214
Idca2Table - convert an element of the idCoxeter algebra into a table . . . 214
IdcaAdd - sum of two elements of the idCoxeter algebra . . . 215
IdcaMinus - difference of two elements of the idCoxeter algebra . . . 216
IdcaMult - multiplication of two elements of the idCoxeter algebra . . . 217
IdcaNormal - normalize an element of the idCoxeter algebra . . . 217
IdcaOnPol - action of an element of the algebra on a polynomial . . . 218
IdcaOnX - action of an element of the idCoxeter algebra on a linear combination of X[perm] . . . 219
IdcaOnXfix - action of an element of the idCoxeter algebra on the ring of polynomials as a free module over **Sym** . . . 220
IdcaYang - compute a special element of the idCoxeter algebra . . . 221
Rd2Idca - calculate a product of generators . . . 222
Table2Idca - convert a table into an element of the idCoxeter algebra . . . 223
5.8 Working in the generic Hecke algebra with HEKA . . . 224

- `Heka2Table` - convert an element of the Hecke algebra into a table . . . 225
- `HekaAdd` - sum of two elements of the Hecke algebra . . . 225
- `HekaCharTable` - compute the table of characters . . . 226
- `HekaJucis` - Jucis-Murphy element of the Hecke algebra . . . 227
- `HekaMinus` - difference of two elements of the Hecke algebra . . . 228
- `HekaMult` - multiplication of two elements of the Hecke algebra . . . 229
- `HekaNormal` - normalize an element of the Hecke algebra . . . 230
- `HekaOnPol` - action of an element of the algebra on a polynomial . . . 231
- `HekaOnX` - action of an element of the Hecke algebra of the symmetric group on a linear combination of `X[perm]` . . . 232
- `HekaOnXfix` - action of an element of the Hecke algebra of the symmetric group on the ring of polynomials as a free module over **Sym** . . . 233
- `HekaYang` - compute a special element of the Hecke algebra . . . 234
- `Rd2Heka` - calculate a product of generators . . . 235
- `Table2Heka` - convert a table into an element of the Hecke algebra . . . 235

5.9 `CG`: a package about classical Lie groups . . . 237

- `CgMultiplicity` - calculate weight multiplicities . . . 237
- `CgPositiveRoots` - positive roots of a classical group . . . 238
- `CgRho` - half the sum of positive roots . . . 239
- `CgSchurSeries` - truncated Schur function series . . . 239
- `O2s` - from the orthogonal Schur basis to the Schur basis . . . 240
- `ODimRep` - calculate the dimension of an irreducible representation of the orthogonal group . . . 241
- `OJtMat` - Jacobi-Trudi-type matrix . . . 241
- `Sp2s` - from the symplectic Schur basis to the Schur basis . . . 242
- `SpDimRep` - calculate the dimension of an irreducible representation of the symplectic group . . . 243
- `SpJtMat` - Jacobi-Trudi-type matrix . . . 243
- `ToO` - express any symmetric function in the basis of orthogonal Schur functions . . . 244
- `ToSp` - express any symmetric function in the basis of symplectic Schur functions . . . 245

5.10 `BN` and the hyperoctahedral groups . . . 246

- `BnBar2Code` - code of a signed permutation . . . 247
- `BnBar2Mat` - signed permutation matrix from a signed permutation . . . 247
- `BnCode2Bar` - signed permutation corresponding to a code . . . 248
- `BnCode2Interval` - list of codes of all group elements below a given one . . . 248
- `BnCode2Inv` - code of the inverse of a group element . . . 249
- `BnCode2Length` - length of a code . . . 250
- `BnCode2ListRd` - list of all reduced decompositions of a code . . . 250
- `BnCode2Palin` - palindromic permutation corresponding to a code . . . 251
- `BnCode2Rd` - canonical reduced decomposition of a code . . . 252
- `BnGenCode` - generate one code . . . 252
- `BnListCode` - generate all codes . . . 253
- `BnMat2Bar` - calculate the signed permutation from a given permutation matrix . . . 254
- `BnMultBar` - multiply signed permutations . . . 254
- `BnMultCode` - multiply codes . . . 255
- `BnPalin2Code` - code corresponding to a palindromic permutation . . . 256
- `BnRd2Code` - calculate a code from a decomposition into simple reflections . . . 256
- `BnTranspoPalin` - image of a simple reflection in $\mathfrak{S}_{2n}$ . . . 257

5.11 Algebras related to hyperoctahedral groups: package `BNA` . . . 258

- BnGaMult - multiplication of two elements of the B_n-group algebra . . . 259
- BnGaOnPol - action of an element of the B_n-algebra . . . 259
- BnGaYang - Yang-Baxter element of the B_n-group algebra . . . 260
- BnIdcaMult - multiplication of two elements of the B_n-idCoxeter algebra . . . 261
- BnIdcaOnPol - action of an element of the B_n-idCoxeter algebra . . . 262
- BnIdcaYang - Yang-Baxter element of the B_n-idCoxeter algebra . . . 263
- BnNcaMult - multiplication of two elements of the B_n-nilCoxeter algebra . . . 264
- BnNcaOnPol - action of an element of the B_n-nilCoxeter algebra . . . 265
- BnNcaYang - Yang-Baxter element of the B_n-nilCoxeter algebra . . . 266
- Bna2Table - convert an element of any B_n-algebra into a table . . . 267
- BnaAdd - sum of two elements of any B_n-algebra . . . 268
- BnaCode2Bar - convert codes into signed permutations . . . 268
- BnaCode2Palin - convert codes into palindromic permutations . . . 269
- BnaMinus - difference of two elements of any B_n-algebra . . . 270
- BnaNormal - normalize an element of a B_n-algebra . . . 270
- BnaPalin2Code - convert palindromic permutations into codes . . . 271
- Rd2BnGa - calculate a product of generators . . . 272
- Rd2BnIdca - calculate a product of generators . . . 273
- Rd2BnNca - calculate a product of generators . . . 273
- Table2Bna - convert a table into an element of any B_n-algebra . . . 274

5.12 SYMF to handle symmetric functions . . . 276

- Char2Sf - convert a virtual character into a symmetric function . . . 277
- Sf2Char - convert a symmetric function into a virtual character . . . 277
- Sf2Table - convert a symmetric function into a table . . . 278
- Sf2TableVar - variable set of a symmetric function . . . 279
- SfAddBasis - addition of a new basis . . . 281
- SfCCProd - compute the conjugacy class product of symmetric functions . . . 282
- SfDiff - differential operator . . . 283
- SfDualBasis - addition of a new basis, dual to an existing one . . . 284
- SfEval - realize the action of symmetric functions on polynomials . . . 286
- SfInternal - compute the inner tensor product of symmetric functions . . . 286
- SfJtMat - compute the Jacobi-Trudi matrix . . . 287
- SfMat - transition matrix between symmetric functions bases . . . 288
- SfOmega - apply the Ω-automorphism . . . 289
- SfPlethysm - plethysm of symmetric functions . . . 290
- SfScalar - scalar product of symmetric functions . . . 291
- SfTheta - apply the Θ-automorphism . . . 292
- SfZee - compute the scalar product of a p-function with itself . . . 293
- Table2Sf - convert a table into a symmetric function . . . 294
- Toc - express any symmetric function in the basis of cycle indices . . . 295
- Toe - convert any symmetric function to a e-polynomial . . . 296
- Toh - convert any symmetric function to a h-polynomial . . . 297
- Tom - express any symmetric function in the basis of basis of monomial symmetric functions . . . 298
- Top - convert any symmetric function to a p-polynomial . . . 299
- Tos - express any symmetric function in the basis of Schur functions . . . 300

5.13 SFA to handle symmetric functions on different alphabets . . . 302

- Pol2SfA - from the basis of monomials to SFA . . . 303
- Sf2SfA - from SYMF objects to SFA objects . . . 304

- SfA2Sf - from SFA objects to SYMF objects . . . 304
- SfA2TableVar - symmetric functions appearing in an expression . . . 305
- SfACollect - collect products that are on algebraic bases . . . 307
- SfAExpand - expand a SFA expression . . . 307
- SfAOmega - apply the Ω-automorphism . . . 309
- SfAVars - specify valid variables in alphabets . . . 310
- ToeA - develop any symmetric function on the e-basis . . . 311
- TohA - develop any symmetric function on the h-basis . . . 312
- TomA - develop any symmetric function on the m-basis . . . 312
- TopA - develop any symmetric function on the p-basis . . . 313
- TosA - develop any symmetric function on the s-basis . . . 314

5.14 FREE, a package to work in the free algebra . . . 316

- Free2NilFree - inverse plaxification map . . . 317
- Free2Plax - quotient of the free algebra by plactic relations . . . 317
- Free2PlaxClass - replace each word by its plactic class . . . 318
- Free2Pol - from words to polynomials . . . 318
- Free2StdFree - standardization of words . . . 319
- Free2Table - convert an element of the free algebra into a table . . . 320
- FreeAdd - sum of two elements of the free algebra . . . 320
- FreeConcat - concatenation product in the free algebra . . . 321
- FreeMinus - difference of two elements of the free algebra . . . 322
- FreeNormal - normalize an element of the free algebra . . . 323
- FreeShuffle - shuffle product in the free algebra . . . 323
- IdcaOnFree - action of the idCoxeter algebra on words . . . 324
- NilFree2Free - plaxification map . . . 325
- Pol2Free - from polynomials to words . . . 326
- Pol2Row - from polynomials to increasing words . . . 326
- Sf2Free - from symmetric polynomials to words . . . 327
- SgaOnFree - action of the symmetric group algebra on words . . . 328
- Table2Free - convert a table into an element of the free algebra . . . 328

5.15 CLG, a package to handle characters of linear groups . . . 330

- CLG_n - fix the order of the linear group . . . 331
- Gauss - compute a Gaussian polynomial . . . 331
- GlDimRep - compute the dimension of a representation . . . 332
- Toe_n - convert any symmetric function to a e-polynomial . . . 333
- Toh_n - convert any symmetric function to a h-polynomial . . . 334
- Tom_n - express any symmetric function in the basis of monomial symmetric functions 335
- Top_n - convert any symmetric function to a p-polynomial . . . 336
- Tos_n - express any symmetric function in the basis of Schur functions . . . 337
- Tox_n - express any symmetric function in the basis of monomials . . . 338
- x2m_n - from the basis of monomials to monomial symmetric function basis . . . 339

5.16 Noncommutative and quasi-symmetric functions with NCSF . . . 341

- NcsfInternal - internal product . . . 341
- NcsfMat - transition matrix from X-basis to Y-basis . . . 342
- NcsfPairing - computes the scalar product of quasi-symmetric function and noncommutative symmetric function . . . 343
- NcsfPlethysm - computes the plethysm of symmetric function and a quasi-symmetric function . . . 344
- NcsfTransAlpha - specialization $A/(1-q) \longrightarrow A$ or $A \longrightarrow A/(1-q)$ . . . 345

- Qsf2x - from quasi-symmetric function to the basis of monomials . . . 346
- QsfMat - transition matrix from X-basis to Y-basis . . . 347
- Sf2Qsf - expand a symmetric function on F-basis (dual basis of noncommutative ribbon Schur function) . . . 348
- ToE - express any quasi-symmetric function in the E-basis (dual basis of noncommutative elementary symmetric function) . . . 349
- ToF - express any quasi-symmetric function in the F-basis (dual basis of Ribbon Schur basis) . . . 350
- ToL - express any noncommutative symmetric function in the L-basis (noncommutative elementary symmetric function) . . . 351
- ToM - express any quasi-symmetric function in the M-basis (quasi-monomial symmetric functions) . . . 352
- ToPh - express any noncommutative symmetric function in the Ph-basis (noncommutative power sums symmetric function of the second kind) . . . 353
- ToPs - express any noncommutative symmetric function in the Ps-basis (noncommutative power sums symmetric function of the first kind) . . . 354
- ToQPh - express any quasi-symmetric function in the QPh-basis (dual basis of noncommutative power sums symmetric function of the second kind) . . . 355
- ToQPs - express any quasi-symmetric function in the QPs-basis (dual basis of noncommutative power sums symmetric functions of the first kind) . . . 356
- ToR - express any noncommutative symmetric function in the R-basis (ribbon Schur function) . . . 357
- ToS - express any noncommutative symmetric function in the S-basis (noncommutative complete symmetric function) . . . 358

5.17 Using TAB to manipulate tableaux . . . 360

- Bump - insert a word into a tableau . . . 361
- BumpC - insert a word into a contretableau . . . 362
- CTab2Mat - convert a contretableau into a matrix . . . 363
- CTab2Part - compute the shape of a contretableau . . . 363
- CTab2Word - convert a contretableau into a word . . . 364
- InvSchensted - inverse Schensted correspondence . . . 365
- ListStdTab - generate all standard tableaux . . . 366
- ListTab - generate all tableaux of a given shape and evaluation . . . 367
- ListYama - generate all Yamanouchi words of a given evaluation . . . 368
- Mat2CTab - convert a matrix into a contretableau . . . 368
- Mat2Tab - convert a matrix into a tableau . . . 369
- Part2ListStdTab - generate all standard tableaux of a given shape . . . 370
- PermOnCTab - action of a permutation on a product of columns . . . 371
- PermOnTab - action of a permutation on a product of rows . . . 371
- Schensted - Schensted correspondence . . . 372
- StdTab2SgaIdemp - build an idempotent from a standard tableau . . . 373
- StdTab2Yama - convert a standard tableau into a Yamanouchi word . . . 374
- Tab2Mat - convert a tableau into a matrix . . . 375
- Tab2Part - compute the shape of a tableau . . . 375
- Tab2Word - convert a tableau into a word . . . 376
- Word2CTab - convert a word into a contretableau . . . 377
- Word2Charge - returns the charge of a word . . . 377
- Word2CoCharge - returns the cocharge of a word . . . 378
- Word2Key - left or right key . . . 379

- `Word2Tab` - convert a word into a tableau . . . 380
- `Yama2StdTab` - convert a Yamanouchi word into a standard tableau . . . 381

5.18 Manipulating Schubert polynomials with `SP` . . . 382

- `Flag` - fix the degree of the symmetric group . . . 383
- `NewtonInterp` - Newton Interpolation formula . . . 383
- `Sp2TableVar` - variable set for the `SP` package . . . 384
- `SpScalarPol` - scalar product on polynomials . . . 386
- `SpSpecialize` - specialize a set of variables . . . 386
- `TableX` - table of all Schubert polynomials . . . 388
- `TableXX` - table of all double Schubert polynomials . . . 389
- `TableY` - table of all Schubert polynomials . . . 390
- `TableYY` - table of all double Schubert polynomials . . . 391
- `ToX` - convert any expression to the **X** Schubert basis . . . 392
- `ToXX` - convert any expression to the $\mathbb{X}$ Schubert basis . . . 393
- `ToY` - convert any expression to the **Y** Schubert basis . . . 394
- `ToYY` - convert any expression to the $\mathbb{Y}$ Schubert basis . . . 395
- `Tox` - express any expression in the basis of monomials . . . 397
- `X2Dim` - specialization in **X** Schubert basis . . . 398
- `x2X` - from the basis of monomials to **X** Schubert basis . . . 399
- `x2XX` - from the basis of monomials to $\mathbb{X}$ Schubert basis . . . 399

5.19 Computations in the ring $\mathbb{Z}[x_1, \ldots, x_n]$ with `FM` . . . 401

- `FM_n` - fix the number of variables . . . 401
- `ToXfix` - convert any expression to the **X** Schubert basis of the ring of polynomials as a free module over **Sym** . . . 402
- `ToXfixScal` - convert any expression to the **X** Schubert basis of the ring of polynomials as a free module over **Sym** . . . 403
- `X2Xfix` - from **X** Schubert basis to **X** Schubert basis of the ring of polynomials as a free module over **Sym** . . . 405
- `Xfix2X` - from **X** Schubert basis with symmetric functions as coefficients to **X** Schubert basis . . . 406
- `Xfix2x` - from **X** Schubert basis with symmetric functions as coefficients to the basis of monomials . . . 407
- `x2Xfix` - from the basis of monomials to **X** Schubert basis of the ring of polynomials as a free module over **Sym** . . . 407
- `x2XfixScal` - from the basis of monomials to **X** Schubert basis of the ring of polynomials as a free module over **Sym** . . . 408

6 About ACE 3.0 **411**

Bibliography **413**

Characters of classical groups . . . 414
Fock space representation of $U_q(\widehat{sl}_n)$ . . . 415
Noncommutative symmetric functions . . . 415
Schubert polynomials . . . 417
Schubert polynomials computations in $\mathbb{Z}[x_1, \ldots, x_n]$ . . . 418
Symmetric functions . . . 418
Words and tableaux . . . 418

Index **419**
A . . . 419
B . . . 419
C . . . 420
D . . . 421
E . . . 421
F . . . 421
G . . . 421
H . . . 421
I . . . 421
K . . . 422
L . . . 422
M . . . 422
N . . . 422
O . . . 422
P . . . 422
Q . . . 423
R . . . 423
S . . . 423
T . . . 424
V . . . 424
W . . . 425
X . . . 425
Y . . . 425

1

Overview

The Algebraic Combinatorics Environment includes various packages providing functions to work with classical structures in algebraic combinatorics (partitions, compositions, permutations, words, tableaux, divided differences, symmetric functions), to compute in several algebras related to the symmetric group and to the hyperoctahedral group. It also gives functions to handle characters and representations of several classical groups, and to manipulate more recent objects like Schubert polynomials, non commutative symmetric functions or quasi-symmetric functions:

- `BN`: hyperoctahedral group.
- `BNA`: algebras related to hyperoctahedral groups.
- `CG`: classical Lie groups.
- `CLG`: characters of the linear groups.
- `COMP`: compositions.
- `FM`: polynomial ring in several variables.
- `FREE`: free algebra of words and free algebra modulo the plactic relations.
- `HEKA`: Hecke algebra.
- `IDCA`: idCoxeter algebra.
- `NCA`: nilCoxeter algebra.
- `NCSF`: noncommutative symmetric functions and quasi-symmetric functions.
- `PART`: partitions.
- `SFA`: symmetric functions on different alphabets.
- `SG`: symmetric group.
- `SGA`: symmetric group algebra.
- `SP`: Schubert polynomials.
- `SYMF`: symmetric functions.
- `TAB`: tableaux.

We give here a brief presentation of these libraries. The background needed to understand all the examples is presented in the next chapter, and is mainly detailed in the mathematical manual available with the `ACE` system.

1.1 The BN library

This package gives functions to work with the hyperoctahedral group, in particular with its Bruhat order. It provides functions to handle the elements of this group, represented by codes. It also gives functions to make conversions between a code and a signed permutation (`Bar`) or a palindromic permutation (`Palin`). For instance, the `BnCode2Interval` function computes the list of all codes of permutations that are smaller than a given one according to the Bruhat order:

```
ACE> r := BnCode2Interval([1,1,3]);

 r := [[1, 1, 3], [1, 1, 2], [1, 2, 1], [1, 1, 1], [1, 0, 3], [1, 0, 2], [1, 2, 0],

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

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

ACE> map(BnCode2Bar, r);

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

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

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

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

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

1.2 The BNA library

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

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

                                      2
                                    x1  x2 x3
```

One has also the `BnIdcaYang` function which calculates a Yang-Baxter element of the B_n-idCoxeter algebra.

The set `BnIdcaYang(code)`, for all codes in B_n is a linear basis of the B_n-idCoxeter algebra, as a free module with coefficients in the x_i's. This basis is defined by recursion:

```
ACE> BnIdcaYang([0,1]);

                                    (- x1 x3 + x2 x4) B[0, 1]
                        B[0, 0] - -------------------------
                                              x1 x3
```

1.3 The CG library

This package gives functions to work with the representation theory of the classical Lie groups, *i.e.* the symplectic and orthogonal groups. It provides functions to compute products of characters, dimension of irreducible representations together with functions to handle new bases of the ring of symmetric polynomials, that are the orthogonal, named 'O' and the symplectic, named 'Sp', Schur function bases.

Using a special rule, one can for instance easily compute the product of symplectic Schur functions. This is automatically recognized in the ToSp function which expresses any symmetric function in the symplectic Schur function basis:

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

      12 Sp[2, 2, 1, 1, 1] + 15 Sp[2, 2, 2, 1] + 20 Sp[1, 1, 1]  +  43 Sp[2, 1]

           + Sp[2, 2, 2, 1, 1, 1]  +  12 Sp[5, 2]  + 6 Sp[5]  +  42 Sp[2, 2, 1]

           + 18 Sp[4, 1, 1, 1] +  12 Sp[5, 1, 1]  +  3 Sp[6, 1]  +  30 Sp[4, 1]

           + 9 Sp[4, 3, 1, 1] + 8 Sp[4, 3, 2] + 48 Sp[3, 1, 1] + 24 Sp[3, 3, 1]

           + 36 Sp[4, 2, 1]  +  15 Sp[4, 3]   +   24 Sp[3, 2, 2]   +   20 Sp[3]

           + 9 Sp[4, 2, 2, 1] + 4 Sp[4, 4, 1]  +  4 Sp[5, 2, 2]  +  42 Sp[3, 2]

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

           + Sp[6, 3] + 6 Sp[5, 2, 1, 1] +  3 Sp[2, 1, 1, 1, 1, 1]  +  12 Sp[1]

           + Sp[4, 1, 1, 1, 1, 1]  +  2 Sp[3, 2, 1, 1, 1, 1]  +  Sp[6, 1, 1, 1]

           + 6 Sp[4, 2, 1, 1, 1] + 2 Sp[5, 1, 1, 1, 1]  +  12 Sp[3, 1, 1, 1, 1]

           + 8 Sp[3, 3, 2, 1]   +   30 Sp[2, 1, 1, 1]   +   6 Sp[3, 2, 2, 1, 1]

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

           + 36 Sp[3, 2, 1, 1] + 2 Sp[3, 3, 3]
```

Moreover, the ToO function expresses any symmetric function in the orthogonal Schur function basis:

```
ACE> ToO(s[2,1]*O[3]);

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

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

1.4 The CLG library

This library provides functions to manipulate characters of the linear group $GL(n)$. It essentially amounts to consider symmetric functions in a fixed number of variables. For instance, given the order of the linear group, say 3, the `Tox_n` function converts any symmetric function to the basis of monomials:

```
ACE> CLG_n(3);
                                    3

ACE> Tox_n(s[2,1]*h2);

      4          3   2       2   3        4       2   3       3   2        4
     x3  x2 + 2 x3  x2  + 2 x3  x2  + x1 x3  + 2 x1  x3  + 2 x1  x3  + x3 x2

               4     4          3   2       2   3        4       3
         + x3 x1  + x2  x1 + 2 x2  x1  + 2 x2  x1  + x2 x1  + 4 x3  x2 x1

              2   2          2      2          3             2   2             3
         + 6 x3  x2  x1 + 6 x3  x2 x1  + 4 x3 x2  x1 + 6 x3 x2  x1  + 4 x3 x2 x1
```

At the level of the Schur basis, it amounts to consider only partitions having at most 3 parts:

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

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

In the basis of product of complete symmetric functions, it amounts to reduce all products of complete functions having more than 3 terms:

```
ACE> Toh_n(h2^2*h1^2);

                          3                   2
                        h2  + 2 h3 h2 h1 - h3  - 2 h4 h2 + h6
```

1.5 The COMP library

This package provides functions to work with compositions. Available functions give tools to compare, conjugate, generate, etc., compositions.

For instance, the `ListCompo` function generates compositions of a given weight:

```
ACE> ListCompo(4);

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

It accepts many options which make it very attractive to generate compositions having certain properties:

```
ACE> ListCompo(2, 'allowzeros', 'lg'=3);

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

One has also the `ListCompoFiner` function which generates all compositions finer than a given composition, that is all compositions obtained by decomposing the parts of the initial composition. The result is a list of compositions, as shown below:

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

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

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

1.6 The FM library

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

```
ACE> FM_n(4);

                                      4
ACE> x2XfixScal(q^2*x2^3*x4);

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

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

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

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

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

Furthermore, we also have generalized the Monk formula [70] to describe the multiplicative structure of this module, as illustrated by the following example:

```
ACE> ToXfix(x3*X[3,1,4,2]);

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

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

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

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

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

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

1.7 The FREE library

This package provides functions to work with the free algebra, that is the algebra of words. A general element is expressed as a linear combination of `w[`*word*`]`, where *word* is a sequence of integers, with arbitrary coefficients.

The `Sf2Free` function converts a symmetric polynomial in n variables into an element of the free algebra. The correspondance amounts to develop each Schur function as a sum of tableaux (seen as special words) of a given shape. Thus, the symmetric function is first expressed in the basis of Schur functions and then each Schur function is converted:

```
ACE> Sf2Free(s[2,1], 3);

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

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

The `Pol2Free` function converts a polynomial into an element of the free algebra. The polynomial is first expressed in the basis of products of elementary symmetric functions on an alphabet flag (dual basis of the usual basis of monomials). Then, this product is seen as the noncommutative concatenation of elementary symmetric functions which are seen as sums of increasing words:

```
ACE> Pol2Free((x1+x2)*x1 + x2^2);

                            w[2, 2] + w[1, 2] + w[1, 1]
```

The `FreeShuffle` function computes the shuffle product of two elements of the free algebra:

```
ACE> FreeShuffle(w[1,2,1,1] + w[2,1,1,1], w[1,2,1]);

              4 w[2, 1, 1, 1, 1, 2, 1]  + 6 w[2, 1, 1, 1, 2, 1, 1]

            + 4 w[2, 1, 2, 1, 1, 1, 1]  + 4 w[1, 2, 1, 1, 1, 2, 1]

            + 8 w[1, 2, 1, 1, 2, 1, 1] + 10 w[1, 2, 1, 2, 1, 1, 1]

            + 8 w[1, 2, 2, 1, 1, 1, 1] +  2 w[1, 1, 2, 1, 1, 2, 1]

            + 6 w[1, 1, 2, 1, 2, 1, 1] + 12 w[1, 1, 2, 2, 1, 1, 1]

            + 6 w[2, 1, 1, 2, 1, 1, 1]
```

It is also possible to work in the plactic monoid, *i.e.* the free algebra of words modulo the plactic relations:

```
ACE> Free2Plax(w[1,2,1] + w[2,1,1] + z*w[1,3,2]);

                              2 w[2, 1, 1] + z w[3, 1, 2]

ACE> Free2PlaxClass(w[1,2,1]);

                                w[2, 1, 1] + w[1, 2, 1]

ACE> Free2PlaxClass(w[2,1,1]);

                                w[2, 1, 1] + w[1, 2, 1]

ACE> Free2PlaxClass(w[1,3,2]);

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

The `Free2NilFree` is the inverse plaxification map which transforms a word into a reduced decomposition. The inverse plaxification map has the property to send a plactic class onto a nil plactic class:

```
ACE> r := Free2PlaxClass(w[1,2,1,2]);

               r := w[2, 1, 1, 2] + w[1, 2, 1, 2] + w[1, 2, 2, 1]

ACE> Free2NilFree(r);

                    w[2, 1, 2, 2] + w[1, 1, 2, 1] + w[1, 2, 1, 2]
```

The `NilFree2Free` function transforms each word seen as a reduced decomposition into a word, according to the plaxification map:

```
ACE> NilFree2Free(w[2, 1, 2, 2] + w[1, 1, 2, 1] + w[1, 2, 1, 2]);

                    w[2, 1, 1, 2] + w[1, 2, 2, 1] + w[1, 2, 1, 2]
```

1.8 The HEKA library

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

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

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

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

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

1.9 The IDCA library

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

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

together with the extra relations: $\pi_i^2 = -\pi_i$. This algebra corresponds to the Hecke algebra for which we set $q = 0$. It is related to *Grothendieck polynomials* and the K-theory of the flag manifold. We can distinguish special elements of this algebra such as the *Demazure* elements: `Idca2Demazure(A[`*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> r := Idca2Demazure(A[2,3,1]);

             r := A[2, 3, 1] + A[2, 1, 3] + A[1, 3, 2] + A[1, 2, 3]

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

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

1.10 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*A[1,3,2] - q^3*A[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]
```

1.11 The NCSF library

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

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

corresponds to a Lie idempotent of the descent algebra. This can be checked as follows:

```
ACE> r := NcsfTransAlpha(Ps[3], 'direct', 'q'):
ACE> r := map(simplify, ToR(r*(1-q^3)/3, 'collect'));

                    R[2, 1] q       R[1, 2]
          r := - 1/3 --------- - 1/3 ------- + 1/3 R[1, 1, 1] + 1/3 R[3]
                      q + 1          q + 1

ACE> map(simplify, ToR(NcsfInternal(r, r), 'collect'));

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

The NCSF package gives a function to compute the transition matrices between the different bases of noncommutative symmetric functions. For instance:

```
ACE> NcsfMat(3, 'R', 'S');

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

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

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

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

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

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

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

1.12 The PART library

This package provides functions to work with partitions. Available functions deal with border, conjugate, diagonals, Frobenius notation, hooks, etc. of a partition, and also with lists of (skew) partitions.

For instance, we can count the number of partitions of 5 with at most 3 parts and with the first part lesser than 3 and the second one lesser than 2:

```
ACE> ListPart(5, 'maxpart'=[3,2], 'maxlg'=3);

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

In fact, we are also able to count the number of partitions whose diagram is contained into the diagram of a given partition:

```
ACE> ListPartIn([40,30,15,10,8,7,5,3,1], 'nb');

                                      9631414
```

We also give a function to construct the order matrix on all partitions of n. The considered order is the natural order: $I \geq J$ if $\sum_I \leq \sum_J$ for the lexicographic order, in which $\sum_I = [I_1, I_1+I_2, I_1+I_2+I_3, \ldots]$. The inverse matrix is known as the Moebius matrix:

```
ACE> ListPart(5);

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

```
ACE> PartOrderMat(5);

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

The package also includes several functions to change the coding of partitions.

1.13 The SFA library

Because symmetric functions can also be considered as operators acting on the ring of polynomials with coefficients in $\mathbb{R}$, the **SFA** package has been included in **ACE**.

In the **SFA** package—*Symmetric Functions on different Alphabets*, we give the possibility of doing more formal computations (with respect to the **SYMF** package for instance) by implementing the structure of Λ-ring of symmetric functions viewed as such operators. In other words, the objects, which are symmetric functions, contain their own alphabets, which are polynomials with coefficients in $\mathbb{R}$.

For instance, the **SFAExpand** function expands, whenever possible, any expression in **SFA**. It is a direct implementation of the Λ-ring structure. Expansion is made considering the shape of the alphabet of each symmetric function appearing in the expression, and also products of symmetric functions:

```
ACE> SfAExpand( p[4,2](7+A1) );

                    49 + 7 p[2](A1) + 7 p[4](A1) + p[4, 2](A1)
```

Expansion is inductively propagated, thus enabling chains of plethysms:

```
ACE> SfAExpand( (p[3]/3 + p[2,1]/2 + p[1,1,1]/6)(p[2](6*A1^2) + k) );

              3        2              2
          1/6 k  + 1/2 k  + 1/3 k + 3 k  p[2, 2](A1) + 3 k p[2, 2](A1)

            + 3 k p[4, 4](A1) + 18 k p[2, 2, 2, 2](A1) + 2 p[6, 6](A1)
```

```
              + 18 p[4, 4, 2, 2](A1) + 36 p[2, 2, 2, 2, 2, 2](A1)
```

The `SfAVars` function specifies which variables are considered for the `SfAExpand` function. Thus, after having specified that the x_i's and the t variables must be considered:

```
ACE> SfAVars({ {x}, t });

                                        {t, {x}}
```

we can develop a Schur function as follows:

```
ACE> SfAExpand( s[3](x1+x2-t) );

                    2                 2     3        2     2        3
               - t x2  - t x1 x2 - t x1  + x2  + x1 x2  + x1  x2 + x1
```

1.14 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 instance a function that gives a way to realize the comparison between two permutations according to the *Bruhat order*, or the `Perm2Interval` function that computes the list of the Lehmer codes of all permutations that are smaller than a given one, according to this order:

```
ACE> r := Perm2Interval([3,4,1,2]);

      r := [[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, r);

          [[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> Perm2Betti([3,4,1,2], 'q');

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

corresponds, in geometric terms, to the Poincaré polynomial:

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

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

One can also list or count permutations satisfying certain properties. For instance, there are 103 vexillary permutations in $\mathfrak{S}_5$:

```
ACE> ListPerm(5, 'vexillary', 'nb');

                                        103
```

among which, 15 have 7 inversions and for instance 9 have 8 inversions:

```
ACE> ListPerm(5, 'vexillary', 'nb', 'level');

                          [1, 4, 6, 11, 16, 18, 18, 15, 9, 4, 1]
```

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

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

One has also the possibility to generate a central idempotent of the symmetric group algebra corresponding to a given partition, as an element of this algebra. The `Part2SgaCIdemp` function is devoted to this purpose:

```
ACE> r := Part2SgaCIdemp([2,1,1]);

  r := 3/8 A[1, 2, 3, 4] - 1/8 A[1, 2, 4, 3] + 1/8 A[4, 1, 2, 3] - 1/8 A[1, 3, 2, 4]

     - 1/8 A[1, 4, 3, 2] - 1/8 A[3, 4, 1, 2] + 1/8 A[3, 1, 4, 2] + 1/8 A[4, 3, 1, 2]

     - 1/8 A[2, 1, 3, 4] + 1/8 A[2, 4, 1, 3] - 1/8 A[2, 1, 4, 3] + 1/8 A[2, 3, 4, 1]

     - 1/8 A[4, 2, 3, 1] - 1/8 A[3, 2, 1, 4] + 1/8 A[3, 4, 2, 1] - 1/8 A[4, 3, 2, 1]

ACE> SgaMinus(r, SgaMult(r, r));

                                          0
```

1.16 The SP library

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

```
ACE> TableX(3);

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

```
                                                    2
                                        [3, 1, 2] = x1
                                                        2
                                        [3, 2, 1] = x2 x1
                                    ])
```

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

```
ACE> ToXX(x4*XX[4,1,2,5,3,7,8,6]);

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

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

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

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

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

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

1.17 The SYMF library

It provides functions to handle symmetric functions. Several bases are available but one can add new ones declared for instance as dual to another already existing basis. This package uses specific algorithms for the changes of basis to accelerate certain computations.

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

```
ACE> Tos(h3^2*h2);

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

                                        + 3 s[6, 2] + s[6, 1, 1] + 2 s[7, 1] + s[8]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The general changes of basis divide calculations into elementary operations easily computed using these formulae. The package provides also some internal functions to compute specific plethysms. These special functions are called by the `SfPlethysm` function:

```
ACE> SfPlethysm(e4, e4, 'e', 'e', 's');

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

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

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

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

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

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

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

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

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

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

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

The SYMF package gives a function to compute the transition matrices between the different bases of the ring of symmetric functions. For instance:

```
ACE> SfMat(5, 's', 'm');

                              [ 1  1  1  1  1  1  1 ]
                              [                     ]
                              [ 0  1  1  2  2  3  4 ]
                              [                     ]
                              [ 0  0  1  1  2  3  5 ]
                              [                     ]
                              [ 0  0  0  1  1  3  6 ]
                              [                     ]
                              [ 0  0  0  0  1  2  5 ]
                              [                     ]
                              [ 0  0  0  0  0  1  4 ]
                              [                     ]
                              [ 0  0  0  0  0  0  1 ]

ACE> ListPart(5);

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

```
ACE> Tom(s[3,1,1]);

             m[3, 1, 1] + m[2, 2, 1] + 3 m[2, 1, 1, 1] + 6 m[1, 1, 1, 1, 1]
```

1.18 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> r := Word2Tab(w[3,1,2]);

                                r := [[3], [1, 2]]

ACE> StdTab2SgaIdemp(r);

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

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

We have for instance, the `ListStdTab` function that computes the list of all standard tableaux filled with n letters. The `Part2ListStdTab` function generates all standard tableaux of a given shape. This is done by generating all Yamanouchi words of a given weight:

```
ACE> Part2ListStdTab([4,2,1]);

                [[[3], [2, 5], [1, 4, 6, 7]], [[3], [2, 6], [1, 4, 5, 7]],

                 [[3], [2, 7], [1, 4, 5, 6]], [[4], [2, 7], [1, 3, 5, 6]],

                 [[5], [2, 7], [1, 3, 4, 6]], [[6], [2, 7], [1, 3, 4, 5]],

                 [[7], [2, 6], [1, 3, 4, 5]], [[7], [3, 6], [1, 2, 4, 5]],

                 [[7], [4, 6], [1, 2, 3, 5]], [[7], [5, 6], [1, 2, 3, 4]],

                 [[6], [3, 7], [1, 2, 4, 5]], [[6], [4, 7], [1, 2, 3, 5]],
```

```
[[6], [5, 7], [1, 2, 3, 4]], [[5], [3, 7], [1, 2, 4, 6]],

[[5], [4, 7], [1, 2, 3, 6]], [[4], [3, 7], [1, 2, 5, 6]],

[[4], [2, 6], [1, 3, 5, 7]], [[5], [2, 6], [1, 3, 4, 7]],

[[6], [2, 5], [1, 3, 4, 7]], [[7], [2, 5], [1, 3, 4, 6]],

[[7], [3, 5], [1, 2, 4, 6]], [[7], [4, 5], [1, 2, 3, 6]],

[[6], [3, 5], [1, 2, 4, 7]], [[6], [4, 5], [1, 2, 3, 7]],

[[5], [3, 6], [1, 2, 4, 7]], [[5], [4, 6], [1, 2, 3, 7]],

[[4], [3, 6], [1, 2, 5, 7]], [[4], [2, 5], [1, 3, 6, 7]],

[[5], [2, 4], [1, 3, 6, 7]], [[6], [2, 4], [1, 3, 5, 7]],

[[7], [2, 4], [1, 3, 5, 6]], [[7], [3, 4], [1, 2, 5, 6]],

[[6], [3, 4], [1, 2, 5, 7]], [[5], [3, 4], [1, 2, 6, 7]],

[[4], [3, 5], [1, 2, 6, 7]]]
```

We have also the `ListTab` function which returns the list of all tableaux of a given shape and a given weight:

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

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

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

The `PermOnTab` function plactically permutes rows in a product of rows (not necessarily a tableau, but the image of a tableau under a row-permutation).

A simple plactic row-transposition transforms a two-row contre-tableau into the two-row tableau which is obtained from it by Schensted algorithm, or conversely. This transformation is also given by the "jeu de taquin".

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

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

1.19 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 convert all standard internal *Maple* functions or objects such as integrals, limits, sums, products, and even arrays and tables. A special format devoted to the objects manipulated in `ACE` is included by default. For instance:

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

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

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

One can also compute for example the plactic class corresponding to a given word as explained previously:

```
ACE> Tex(Free2PlaxClass(w[2,3,1,3,2,1,1]), 'TEX_MODE'='MATHS');
$3\,2\,1\,2\,1\,3\,1+3\,2\,1\,2\,3\,1\,1+3\,2\,2\,1\,3\,1\,1
+3\,2\,2\,3\,1\,1\,1+2\,1\,3\,3\,2\,1\,1+2\,1\,3\,3\,1\,2\,1
+2\,3\,1\,3\,1\,2\,1+2\,3\,3\,1\,1\,2\,1+2\,3\,1\,3\,2\,1\,1
+2\,3\,3\,1\,2\,1\,1+2\,3\,2\,3\,1\,1\,1+2\,3\,3\,2\,1\,1\,1
+2\,3\,2\,1\,3\,1\,1+3\,2\,2\,1\,1\,3\,1+2\,1\,3\,2\,1\,3\,1
+2\,1\,3\,2\,3\,1\,1+2\,1\,3\,1\,3\,2\,1+2\,3\,1\,1\,3\,2\,1
+2\,3\,1\,2\,3\,1\,1+2\,3\,1\,2\,1\,3\,1+2\,3\,2\,1\,1\,3\,1$
```

and obtain the following TEX output: $3\,2\,1\,2\,1\,3\,1+3\,2\,1\,2\,3\,1\,1+3\,2\,2\,1\,3\,1\,1+3\,2\,2\,3\,1\,1\,1+2\,1\,3\,3\,2\,1\,1+$

2 1 3 3 1 2 1 + 2 3 1 3 1 2 1 + 2 3 3 1 1 2 1 + 2 3 1 3 2 1 1 + 2 3 3 1 2 1 1 + 2 3 2 3 1 1 1 + 2 3 3 2 1 1 1 + 2 3 2 1 3 1 1 + 3 2 2 1 1 3 1 + 2 1 3 2 1 3 1 + 2 1 3 2 3 1 1 + 2 1 3 1 3 2 1 + 2 3 1 1 3 2 1 + 2 3 1 2 3 1 1 + 2 3 1 2 1 3 1 + 2 3 2 1 1 3 1.

Let us see another example about tableaux. By running the next command, we obtain the TEX output corresponding to the list of tableaux (seen as lists of lists), of a given shape and a given evaluation, that is:

```
ACE> Tex(ListTab([4,2,1], [1$7]), 'TEX_MODE'='MATHS');
$[[[7], [5, 6], [1, 2, 3, 4]], [[7], [4, 6], [1, 2, 3, 5]], [
[7], [3, 6], [1, 2, 4, 5]], [[7], [2, 6], [1, 3, 4, 5]], [[7
], [4, 5], [1, 2, 3, 6]], [[7], [3, 5], [1, 2, 4, 6]], [[7],
[2, 5], [1, 3, 4, 6]], [[7], [3, 4], [1, 2, 5, 6]], [[7], [2
, 4], [1, 3, 5, 6]], [[6], [5, 7], [1, 2, 3, 4]], [[6], [4,
7], [1, 2, 3, 5]], [[6], [3, 7], [1, 2, 4, 5]], [[6], [2, 7]
, [1, 3, 4, 5]], [[5], [4, 7], [1, 2, 3, 6]], [[5], [3, 7],
[1, 2, 4, 6]], [[5], [2, 7], [1, 3, 4, 6]], [[4], [3, 7], [1
, 2, 5, 6]], [[4], [2, 7], [1, 3, 5, 6]], [[3], [2, 7], [1,
4, 5, 6]], [[6], [4, 5], [1, 2, 3, 7]], [[6], [3, 5], [1, 2,
4, 7]], [[6], [2, 5], [1, 3, 4, 7]], [[6], [3, 4], [1, 2, 5,
7]], [[6], [2, 4], [1, 3, 5, 7]], [[5], [4, 6], [1, 2, 3, 7]
], [[5], [3, 6], [1, 2, 4, 7]], [[5], [2, 6], [1, 3, 4, 7]],
[[4], [3, 6], [1, 2, 5, 7]], [[4], [2, 6], [1, 3, 5, 7]], [[
3], [2, 6], [1, 4, 5, 7]], [[5], [3, 4], [1, 2, 6, 7]], [[5]
, [2, 4], [1, 3, 6, 7]], [[4], [3, 5], [1, 2, 6, 7]], [[4],
[2, 5], [1, 3, 6, 7]], [[3], [2, 5], [1, 4, 6, 7]]]$
```

But, this output is not very attractive. However, we can transform each tableau into the corresponding word and convert the list of words into a sum to obtain by the next command:

```
ACE> Tex(convert(map(Tab2Word, ListTab([4,2,1], [1$7])), `+`), 'TEX_MODE'='MATHS');
```

the sum: 7 4 6 1 2 3 5 + 7 2 4 1 3 5 6 + 6 5 7 1 2 3 4 + 6 4 7 1 2 3 5 + 7 3 5 1 2 4 6 + 7 2 5 1 3 4 6 + 7 3 4 1 2 5 6 + 7 2 6 1 3 4 5 + 7 4 5 1 2 3 6 + 7 3 6 1 2 4 5 + 6 3 5 1 2 4 7 + 6 2 5 1 3 4 7 + 3 2 7 1 4 5 6 + 6 4 5 1 2 3 7 + 4 3 7 1 2 5 6 + 4 2 7 1 3 5 6 + 5 3 7 1 2 4 6 + 5 4 7 1 2 3 6 + 5 2 7 1 3 4 6 + 6 3 7 1 2 4 5 + 6 2 7 1 3 4 5 + 7 5 6 1 2 3 4 + 4 3 5 1 2 6 7 + 4 2 5 1 3 6 7 + 3 2 5 1 4 6 7 + 5 3 4 1 2 6 7 + 5 2 4 1 3 6 7 + 3 2 6 1 4 5 7 + 4 2 6 1 3 5 7 + 5 2 6 1 3 4 7 + 4 3 6 1 2 5 7 + 6 2 4 1 3 5 7 + 5 4 6 1 2 3 7 + 5 3 6 1 2 4 7 + 6 3 4 1 2 5 7.

If we want a two-dimensional representation of tableaux, we can tranform each tableau into the corresponding matrix and print 5 tableaux per row by the next command:

```
ACE> lt := map(Tab2Mat, ListTab([4,2,1], [1$7])):
ACE> for i from 1 by 5 to nops(lt) do
       Tex([seq(op(j, lt), j=i..i+4)], 'TEX_MODE'='MATHS');
       TexStr(``);
     od:
```

Thus, we can obtain the next output:

$$\begin{pmatrix} 7 & & & \\ 5 & 6 & & \\ 1 & 2 & 3 & 4 \end{pmatrix}, \begin{pmatrix} 7 & & & \\ 4 & 6 & & \\ 1 & 2 & 3 & 5 \end{pmatrix}, \begin{pmatrix} 7 & & & \\ 3 & 6 & & \\ 1 & 2 & 4 & 5 \end{pmatrix}, \begin{pmatrix} 7 & & & \\ 2 & 6 & & \\ 1 & 3 & 4 & 5 \end{pmatrix}, \begin{pmatrix} 7 & & & \\ 4 & 5 & & \\ 1 & 2 & 3 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 7 & & & \\ 3 & 5 & & \\ 1 & 2 & 4 & 6 \end{pmatrix}, \begin{pmatrix} 7 & & & \\ 2 & 5 & & \\ 1 & 3 & 4 & 6 \end{pmatrix}, \begin{pmatrix} 7 & & & \\ 3 & 4 & & \\ 1 & 2 & 5 & 6 \end{pmatrix}, \begin{pmatrix} 7 & & & \\ 2 & 4 & & \\ 1 & 3 & 5 & 6 \end{pmatrix}, \begin{pmatrix} 6 & & & \\ 5 & 7 & & \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

$$\begin{pmatrix} 6 & & & \\ 4 & 7 & & \\ 1 & 2 & 3 & 5 \end{pmatrix}, \begin{pmatrix} 6 & & & \\ 3 & 7 & & \\ 1 & 2 & 4 & 5 \end{pmatrix}, \begin{pmatrix} 6 & & & \\ 2 & 7 & & \\ 1 & 3 & 4 & 5 \end{pmatrix}, \begin{pmatrix} 5 & & & \\ 4 & 7 & & \\ 1 & 2 & 3 & 6 \end{pmatrix}, \begin{pmatrix} 5 & & & \\ 3 & 7 & & \\ 1 & 2 & 4 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 5 & & & \\ 2 & 7 & & \\ 1 & 3 & 4 & 6 \end{pmatrix}, \begin{pmatrix} 4 & & & \\ 3 & 7 & & \\ 1 & 2 & 5 & 6 \end{pmatrix}, \begin{pmatrix} 4 & & & \\ 2 & 7 & & \\ 1 & 3 & 5 & 6 \end{pmatrix}, \begin{pmatrix} 3 & & & \\ 2 & 7 & & \\ 1 & 4 & 5 & 6 \end{pmatrix}, \begin{pmatrix} 6 & & & \\ 4 & 5 & & \\ 1 & 2 & 3 & 7 \end{pmatrix}$$

$$\begin{pmatrix} 6 & & & \\ 3 & 5 & & \\ 1 & 2 & 4 & 7 \end{pmatrix}, \begin{pmatrix} 6 & & & \\ 2 & 5 & & \\ 1 & 3 & 4 & 7 \end{pmatrix}, \begin{pmatrix} 6 & & & \\ 3 & 4 & & \\ 1 & 2 & 5 & 7 \end{pmatrix}, \begin{pmatrix} 6 & & & \\ 2 & 4 & & \\ 1 & 3 & 5 & 7 \end{pmatrix}, \begin{pmatrix} 5 & & & \\ 4 & 6 & & \\ 1 & 2 & 3 & 7 \end{pmatrix}$$

$$\begin{pmatrix} 5 & & & \\ 3 & 6 & & \\ 1 & 2 & 4 & 7 \end{pmatrix}, \begin{pmatrix} 5 & & & \\ 2 & 6 & & \\ 1 & 3 & 4 & 7 \end{pmatrix}, \begin{pmatrix} 4 & & & \\ 3 & 6 & & \\ 1 & 2 & 5 & 7 \end{pmatrix}, \begin{pmatrix} 4 & & & \\ 2 & 6 & & \\ 1 & 3 & 5 & 7 \end{pmatrix}, \begin{pmatrix} 3 & & & \\ 2 & 6 & & \\ 1 & 4 & 5 & 7 \end{pmatrix}$$

$$\begin{pmatrix} 5 & & & \\ 3 & 4 & & \\ 1 & 2 & 6 & 7 \end{pmatrix}, \begin{pmatrix} 5 & & & \\ 2 & 4 & & \\ 1 & 3 & 6 & 7 \end{pmatrix}, \begin{pmatrix} 4 & & & \\ 3 & 5 & & \\ 1 & 2 & 6 & 7 \end{pmatrix}, \begin{pmatrix} 4 & & & \\ 2 & 5 & & \\ 1 & 3 & 6 & 7 \end{pmatrix}, \begin{pmatrix} 3 & & & \\ 2 & 5 & & \\ 1 & 4 & 6 & 7 \end{pmatrix}$$

But this is not so attractive too because tableaux are not represented as matrices in your articles. The best thing would be to write TEX macros to dump tableaux:

```
TabMacros:=proc()
  TexStr(`\\def\\boxit#1#2{\\setbox1=\\hbox{\\kern#1{#2}\\kern#1}`);
  TexStr(`\\dimen1=\\ht1 \\advance\\dimen1 by #1 \\dimen2=\\dp1 \\advance\\dimen2 by #1`);
  TexStr(`\\setbox1=\\hbox{\\vrule height\\dimen1 depth\\dimen2\\box1\\vrule}`);
  TexStr(`\\setbox1=\\vbox{\\hrule\\box1\\hrule}`);
  TexStr(`\\advance\\dimen1 by .4pt \\ht1=\\dimen1 `);
  TexStr(`\\advance\\dimen2 by .4pt \\dp1=\\dimen2  \\box1\\relax}`);
  TexStr(`\\catcode``\\@=12`);
  TexStr(`\\newdimen\\squaresize \\squaresize=14pt`);
  TexStr(`\\newdimen\\thickness \\thickness=0.5pt`);
  TexStr(`\\def\\square#1{\\hbox{\\vrule width \\thickness`);
  TexStr(`\\vbox to \\squaresize{\\hrule height \\thickness\\vss`);
  TexStr(`\\hbox to \\squaresize{\\hss#1\\hss}`);
  TexStr(`\\vss\\hrule height\\thickness}`);
  TexStr(`\\unskip\\vrule width \\thickness}\\kern-\\thickness}`);
  TexStr(`\\def\\vsquare#1{\\vbox{\\square{$#1$}}\\kern-\\thickness}`);
  TexStr(`\\def\\blank{\\omit\\hskip\\squaresize}`);
  TexStr(`\\def\\tableau#1{`);
  TexStr(`\\vbox{\\smallskip\\offinterlineskip`);
  TexStr(`\\halign{&\\vsquare{##}\\cr #1}}}`);
end:
```

and then to have a function to dump a tableau:

```
`TeX/UserdefinedFun/Tab`:=proc()
  local
    tab,  # the argument...
    i, j; # variables for loops...
  tab:=args[1];
  TexCat(`\\tableau{`);
  for i from 1 to nops(tab) do
    for j from 1 to nops(op(i, tab))-1 do
      Texer(op(j, op(i, tab)));
      TexCat(` & `);
    od;
    Texer(op(nops(op(i, tab)), op(i, tab)));
    TexCat(`\\cr`);
  od;
  TexCat(`}\\quad`);
end:
```

Then, we have just to dump the macros by:

```
ACE> TabMacros();
```

and each tableau by:

```
ACE> lt := ListTab([4,2,1], [1$7]):
ACE> for i from 1 by 5 to nops(lt) do
       TexStr(‘$$‘);
       for j from 0 to 4 do
         Tex(Tab(op(i+j, lt)));
       od;
       TexStr(‘$$‘);
     od:
```

to obtain the following output:

```
\def\boxit#1#2{\setbox1=\hbox{\kern#1{#2}\kern#1}
\dimen1=\ht1 \advance\dimen1 by #1 \dimen2=\dp1 \advance\dimen2 by #1
\setbox1=\hbox{\vrule height\dimen1 depth\dimen2\box1\vrule}
\setbox1=\vbox{\hrule\box1\hrule}
\advance\dimen1 by .4pt \ht1=\dimen1
\advance\dimen2 by .4pt \dp1=\dimen2  \box1\relax}
\catcode‘\@=12
\newdimen\squaresize \squaresize=14pt
\newdimen\thickness \thickness=0.5pt
\def\square#1{\hbox{\vrule width \thickness
\vbox to \squaresize{\hrule height \thickness\vss
\hbox to \squaresize{\hss#1\hss}
\vss\hrule height\thickness}
\unskip\vrule width \thickness}\kern-\thickness}
\def\vsquare#1{\vbox{\square{$#1$}}\kern-\thickness}
\def\blank{\omit\hskip\squaresize}
\def\tableau#1{
\vbox{\smallskip\offinterlineskip
\halign{&\vsquare{##}\cr #1}}}
$$
\tableau{7\cr5 & 6\cr1 & 2 & 3 & 4\cr}\quad \tableau{7\cr4 & 6\cr1 & 2 & 3 & 5\cr}\quad
\tableau{7\cr3 & 6\cr1 & 2 & 4 & 5\cr}\quad \tableau{7\cr2 & 6\cr1 & 3 & 4 & 5\cr}\quad
\tableau{7\cr4 & 5\cr1 & 2 & 3 & 6\cr}\quad
$$
$$
\tableau{7\cr3 & 5\cr1 & 2 & 4 & 6\cr}\quad \tableau{7\cr2 & 5\cr1 & 3 & 4 & 6\cr}\quad
\tableau{7\cr3 & 4\cr1 & 2 & 5 & 6\cr}\quad \tableau{7\cr2 & 4\cr1 & 3 & 5 & 6\cr}\quad
\tableau{6\cr5 & 7\cr1 & 2 & 3 & 4\cr}\quad
$$
$$
\tableau{6\cr4 & 7\cr1 & 2 & 3 & 5\cr}\quad \tableau{6\cr3 & 7\cr1 & 2 & 4 & 5\cr}\quad
\tableau{6\cr2 & 7\cr1 & 3 & 4 & 5\cr}\quad \tableau{5\cr4 & 7\cr1 & 2 & 3 & 6\cr}\quad
\tableau{5\cr3 & 7\cr1 & 2 & 4 & 6\cr}\quad
$$
$$
\tableau{5\cr2 & 7\cr1 & 3 & 4 & 6\cr}\quad \tableau{4\cr3 & 7\cr1 & 2 & 5 & 6\cr}\quad
\tableau{4\cr2 & 7\cr1 & 3 & 5 & 6\cr}\quad \tableau{3\cr2 & 7\cr1 & 4 & 5 & 6\cr}\quad
```

```
\tableau{6\cr4 & 5\cr1 & 2 & 3 & 7\cr}\quad
$$
$$
\tableau{6\cr3 & 5\cr1 & 2 & 4 & 7\cr}\quad \tableau{6\cr2 & 5\cr1 & 3 & 4 & 7\cr}\quad
\tableau{6\cr3 & 4\cr1 & 2 & 5 & 7\cr}\quad \tableau{6\cr2 & 4\cr1 & 3 & 5 & 7\cr}\quad
\tableau{5\cr4 & 6\cr1 & 2 & 3 & 7\cr}\quad
$$
$$
\tableau{5\cr3 & 6\cr1 & 2 & 4 & 7\cr}\quad \tableau{5\cr2 & 6\cr1 & 3 & 4 & 7\cr}\quad
\tableau{4\cr3 & 6\cr1 & 2 & 5 & 7\cr}\quad \tableau{4\cr2 & 6\cr1 & 3 & 5 & 7\cr}\quad
\tableau{3\cr2 & 6\cr1 & 4 & 5 & 7\cr}\quad
$$
$$
\tableau{5\cr3 & 4\cr1 & 2 & 6 & 7\cr}\quad \tableau{5\cr2 & 4\cr1 & 3 & 6 & 7\cr}\quad
\tableau{4\cr3 & 5\cr1 & 2 & 6 & 7\cr}\quad \tableau{4\cr2 & 5\cr1 & 3 & 6 & 7\cr}\quad
\tableau{3\cr2 & 5\cr1 & 4 & 6 & 7\cr}\quad
$$
```

which gives:

7				7				7				7				7			
5	6			4	6			3	6			2	6			4	5		
1	2	3	4	1	2	3	5	1	2	4	5	1	3	4	5	1	2	3	6

7				7				7				7				6			
3	5			2	5			3	4			2	4			5	7		
1	2	4	6	1	3	4	6	1	2	5	6	1	3	5	6	1	2	3	4

6				6				6				5				5			
4	7			3	7			2	7			4	7			3	7		
1	2	3	5	1	2	4	5	1	3	4	5	1	2	3	6	1	2	4	6

5				4				4				3				6			
2	7			3	7			2	7			2	7			4	5		
1	3	4	6	1	2	5	6	1	3	5	6	1	4	5	6	1	2	3	7

6				6				6				6				5			
3	5			2	5			3	4			2	4			4	6		
1	2	4	7	1	3	4	7	1	2	5	7	1	3	5	7	1	2	3	7

5				5				4				4				3			
3	6			2	6			3	6			2	6			2	6		
1	2	4	7	1	3	4	7	1	2	5	7	1	3	5	7	1	4	5	7

5				5				4				4				3			
3	4			2	4			3	5			2	5			2	5		
1	2	6	7	1	3	6	7	1	2	6	7	1	3	6	7	1	4	6	7

that seems to be more convenient!

2

Background

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

Mein Weltbild, Albert Einstein.

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

The last section deals with the `FOCK` package which is now a satellite package of `ACE`, *i.e.* it is not included in `ACE` but uses the `ACE` environment. The `FOCK` package, developped by Jean-Yves Thibon (Université de Marne-la-Vallée) and Bernard Leclerc (Université de Caen) is included in the Cd-Rom of `ACE` and can be obtained from Jean-Yves Thibon (Université de Marne-la-Vallée, `Jean-Yves.Thibon@univ-mlv.fr`).

2.1 Symmetric group

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

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

We usually denote a permutation $w \in \mathfrak{S}_n$ by the list of the images of $\{1, 2, \ldots, n\}$ under w, seen as a mapping of $[1, n]$ onto itself: $w := [w(1), w(2), \ldots, w(n)]$.
A basic statistic on permutations is the notion of inversion: given a permutation $w \in \mathfrak{S}_n$, an inversion is defined to be a couple (i, j) of indices such that $1 \leq i < j \leq n$ and $w(i) > w(j)$. We will refer to an inversion on consecutive positions i and $i + 1$ as a descent (`Perm2Desc`) of the permutation and conversely, a rise of w is a position i such that $w(i) < w(i + 1)$.

The length $l(w)$ (`Perm2Length`) of w is the number of its inversions, *i.e.* the cardinal of the set $I(w)$ (`Perm2ListInv`):

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

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

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

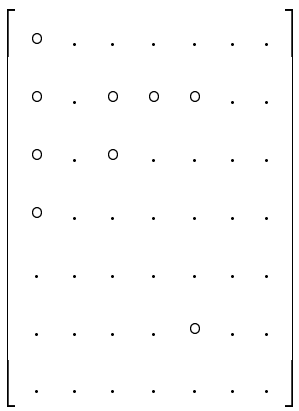
```
ACE> map(Perm2Code, ListPerm(4));

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

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

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

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

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

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

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

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

in which w^{-1} denotes the inverse permutation of w. The diagram $D(w)$ of the previous permutation is the following:



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

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

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

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

known as braid relations, together with the extra relation:

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

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

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

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

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

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

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

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

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

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

The number of reduced decompositions is obtained as follows:

```
ACE> Perm2ListRd([6,5,4,3,2,1], 'nb');

                                    292864
```

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

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

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

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

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

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

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

differing by 1. The order is now called the *Bruhat order* (or strong order), and is the natural order for the idCoxeter algebra (`IDCA`).

The interval of permutations below a given one for the Bruhat order is obtained with the function `Perm2Interval`:

```
ACE> Perm2Interval([3,4,2,1]);

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

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

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

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

whereas the function `Perm2Betti` gives the number of permutations in the interval corresponding to a given permutation according to their lengths:

```
ACE> Perm2Betti([3,4,2,1], 'q');

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

2.2 Symmetric group algebra

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

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

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

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

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

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

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

Using these elements, one can build central idempotents corresponding to standard tableaux, which are mutually orthogonal (`StdTab2SgaIdemp`):

```
ACE> StdTab2SgaIdemp([[3], [1,2,4]]);

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

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

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

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

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

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

Both functions `Sga2Carre` and `Sga2Nabla`) give other distinguished linear bases of $\mathbb{K}[\mathfrak{S}_n]$. One can obtain idempotents from them: `Sga2Carre`(`GenPerm`(n, '`last`')) is the sum of all permutations in $\mathfrak{S}_n$ and `Sga2Nabla`(`GenPerm`(n, '`last`')) is the alternating sum of all permutations in $\mathfrak{S}_n$. More generally, corresponding to a given shape, e.g. $[2,3]$, one will have the idempotent `Sga2Carre(A[2,4,1,3,5])` $\times^{(1)}$ `Sga2Nabla(A[2,5,1,4,3])` that one gets from reading the following special tableaux and contretableaux:

2	4	
1	3	5

2	5
1	4
	3

2.3 NilCoxeter algebra

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

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

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

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

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

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

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

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

equation (2.3) being replaced by:

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

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

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

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

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

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

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

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

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

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

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

involving extra parameters.

2.4 IdCoxeter algebra

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

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

This operator sends polynomials to polynomials and preserves the degree.

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

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

This operator still satisfies the braid relations but now

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

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

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

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

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

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

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

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

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

2.5 Generic Hecke algebra

The simple operators now act as follows:

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

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

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

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

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

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

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

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

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

                                x2 q1 + x1 q2 + q1 x1
```

2.6 Characters of classical Lie groups

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

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

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

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

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

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

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

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

Symplectic and orthogonal Schur functions are defined by

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

where

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

and

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

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

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

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

and

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

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

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

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

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

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

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

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

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

r is the *rank* of the partition.

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

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

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

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

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

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

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

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

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

(2) In French notation.

by the following formulas:

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

and

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

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

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

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

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

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

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

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

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

The inverse series are also known:

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

and

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

and their expansion on Schur basis is:

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

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

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

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

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

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

Similarly:

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

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

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

These branching rules are implemented and used by the functions `ToSp` and `ToO`:

```
ACE> ToSp(s[2,2,1]);
                Sp[1, 1, 1] + Sp[2, 1] + Sp[1] + Sp[2, 2, 1]
ACE> ToO(s[5,3]);
       O[5, 3] + O[3, 3] + O[4, 2] + O[5, 1] + 2 O[3, 1] + O[4] + O[1, 1] + O[2]
```

2.6.1 Dimension of representations

For the symplectic group, the dimension of V_λ is given by [11]:

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

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

```
ACE> SpDimRep([7,4,2]);
                                39312
```

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

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

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

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

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

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

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

```
ACE> ODimRep(4, [3,2]);
                                    24
ACE> ODimRep(5, [3,2]);
                                   105
```

2.6.2 Product of characters

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

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

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

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

Various proofs of this identity can be found (see e.g. [23] [15] [13]).

Using the `ToSp` function that uses this identity when needed, one can obtain for example:

```
ACE> ToSp(Sp[3,2]*Sp[2,1]);
 Sp[5, 3] + Sp[5, 2, 1] + Sp[4, 4] + 2 Sp[4, 3, 1] + Sp[4, 2, 2]
+ Sp[4, 2, 1, 1] + Sp[3, 3, 2] + Sp[3, 3, 1, 1] + Sp[3, 2, 2, 1]
+ 2 Sp[3, 3]   +   4 Sp[3, 2, 1]  +  Sp[2, 2, 1, 1] + 3 Sp[4, 2]
+ Sp[2, 2, 2]   +  2 Sp[4, 1, 1]  +  Sp[3, 1, 1, 1]  +  Sp[5, 1]
+ 3 Sp[3, 1] + 2 Sp[2, 2]  +  2 Sp[2, 1, 1]  +  Sp[4] + Sp[1, 1]
+ Sp[2]
```

2.6.3 Freudenthal's multiplicity formula

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

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

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

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

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

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

                                    22
```

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

```
ACE> CgPositiveRoots(D, 4);

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

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

ACE> CgRho(D, 4);

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

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

2.7 Symmetric functions

The basic reference for this section is Macdonald's book [88]. Let $X = \{x_1,\ x_2,\ \ldots\}$ an infinite set of commutative indeterminates, which will be called an *alphabet*. The *elementary symmetric functions* $e_k(X)$ are defined by the generating series:

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

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

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

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

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

These generating series satisfy the following relations:

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

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

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

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

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

For a partition λ, one sets:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

                                                18

ACE> SfZee([3,3,1]);

                                                18

ACE> SfScalar(s[6,5,3,2,1,1], s[6,5,3,2,1,1], 's', 's');

                                                1
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

                      2      2
                72 p3  p2 p1
```

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

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

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

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

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

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

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

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

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

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

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

One can check that its adjoint is the internal product:

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

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

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

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

```
ACE> Tom(s[5,3,1]);

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

          + 2 m[4, 3, 2]  + 4 m[4, 3, 1, 1] + 5 m[4, 2, 2, 1]  +  9 m[4, 2, 1, 1, 1]

          + 27 m[3, 2, 1, 1, 1, 1] + 45 m[3, 1, 1, 1, 1, 1, 1] + 21 m[2, 2, 2, 2, 1]

          + 99 m[2, 1, 1, 1, 1, 1, 1, 1] + 162 m[1, 1, 1, 1, 1, 1, 1, 1, 1]

          + 12 m[3, 3, 1, 1, 1] + 9 m[3, 2, 2, 2] + 16 m[3, 2, 2, 1, 1]

          + 15 m[4, 1, 1, 1, 1, 1] + 3 m[3, 3, 3] + 7 m[3, 3, 2, 1]

          + 36 m[2, 2, 2, 1, 1, 1] + 60 m[2, 2, 1, 1, 1, 1, 1]

ACE> Tom(s[3,1]*m[2,1,1]);

               45 m[2, 1, 1, 1, 1, 1, 1] + 30 m[3, 1, 1, 1, 1, 1] + 15 m[3, 2, 2, 1]

           + 24 m[3, 2, 1, 1, 1] + 2 m[4, 3, 1] + 7 m[4, 2, 1, 1] + 24 m[2, 2, 2, 2]

           + 48 m[2, 2, 1, 1, 1, 1] + 10 m[3, 3, 1, 1] + 4 m[4, 2, 2]

           + 12 m[4, 1, 1, 1, 1] + 36 m[2, 2, 2, 1, 1] + m[5, 2, 1]

           + 3 m[5, 1, 1, 1] + 5 m[3, 3, 2]
```

We have just seen differences of alphabets. More generally, symmetric functions can be considered as operators acting on the ring of polynomials with coefficients in $\mathbb{R}$.

2.7.1 The structure of Λ-ring

Let P and Q be two elements of the ring $\mathbb{R}[\mathbf{x}]$ of polynomials with coefficients in $\mathbb{R}$. Let $x \in \mathbf{x}$ and $r \in \mathbb{R}$. Let μ and ρ denote some partitions. One defines the Λ operator over $\mathbb{R}[\mathbf{x}]$ by the following relations:

$$\Lambda^i(P) = 0, \quad i < 0, \qquad \Lambda^0(P) = 1, \qquad \Lambda^1(P) = P, \tag{2.52}$$

$$\Lambda^i(x) = 0, \quad i > 1, \qquad \Lambda^i(r) = \binom{r}{i} = \left(\prod_{j=1}^{i}(r-j+1)\right)/i!, \tag{2.53}$$

$$\Lambda^i(P+Q) = \sum_j \Lambda^{i-j}(P)\Lambda^j(Q), \tag{2.54}$$

$$\exists c_\mu^\rho \in \mathbb{Z} \;\; \textit{such that } \Lambda^i(PQ) = \sum_{|\mu|=|\rho|=i} c_\mu^\rho \Lambda^\mu(P)\Lambda^\rho(Q) \quad . \tag{2.55}$$

These axioms totally define the $\Lambda^k(P)$ for all $k \in \mathbb{Z}$ and $P \in \mathbb{R}[\mathbf{x}]$. For the definition of general Λ-rings, one has to add an axiom for the composition $P \to \Lambda^j(P) \to \Lambda^i(\Lambda^j(P))$. Here, because we only deal with polynomials, $\Lambda^j(P)$ is a polynomial and thus $\Lambda^i(\Lambda^j(P))$ is determined by the axioms (2.52—2.55).

Axiom $\Lambda^i(PQ)$ is also known as Cauchy's formula, which is commonly written:

$$\Lambda^i(PQ) = \sum_{|\mu|=i} \Lambda_\mu(P)\Lambda_{\mu^\sim}(Q) \quad , \tag{2.56}$$

where $\Lambda_\mu(P)$ is the minor of index $0, \mu$ of the matrix $\left(\Lambda^{j-i}(P)\right)_{i,j\geq 1}$.

When P is the polynomial $P = x_1 + x_2 + \cdots + x_n$, from (2.52—2.55), one gets that:

$$\Lambda^n(x_1 + x_2 + \cdots + n_x - 1) = \sum(-1)^{n-k}\Lambda^k(P) = \prod(x_i - 1) \quad , \tag{2.57}$$

i.e. $\Lambda^k(P)$ is the elementary symmetric function of the alphabet $\mathbf{x} = \{x_1, x_2, \cdots, x_n\}$ of degree k $(= \sum_{1\leq i_1<\cdots<i_k\leq n} x_{i_1} \ldots x_{i_k})$.

From the axioms, we deduce that if $P = \sum ru$, $r \in \mathbb{R}$ and $u \in \mathbf{x}^m$, $m \in \mathbb{N}$:

$$\sum z^i \Lambda^i(P) = \prod(1 + zu)^r \quad , \tag{2.58}$$

and this could have been taken as a definition of the Λ-ring structure instead of (2.52—2.55).

Instead of the elementary symmetric functions, one could take any other algebraic basis of the ring of symmetric functions and define accordingly the action of elements of this basis on the ring of polynomials $\mathbb{R}[\mathbf{x}]$.

For instance, complete functions h_k, viewed as operators on $\mathbb{R}[\mathbf{x}]$ enjoy:

$$h_i(P) = 0, \quad i < 0, \qquad h_0(P) = 1, \qquad h_1(P) = P, \tag{2.59}$$

$$h_i(x) = x^i, \quad i > 1, \qquad h_i(r) = \left(\prod_{j=1}^{i}(r + j - 1)\right)/i!, \tag{2.60}$$

$$h_i(P + Q) = \sum_j h_{i-j}(P)h_j(Q), \tag{2.61}$$

$$\exists c_\mu^\rho \in \mathbb{Z} \ \ such\ that \qquad h_i(PQ) = \sum_{|\mu|=|\rho|=i} c_\mu^\rho h_\mu(P)h_\rho(Q) \quad . \tag{2.62}$$

This action is equivalent to the one described by axioms (2.52—2.55), because in fact:

$$h_k(P) = (-1)^k\Lambda^k(-P) \quad . \tag{2.63}$$

Accordingly, (2.59—2.62) becomes, for $P = \sum ru$, $r \in \mathbb{R}$ and $u \in \mathbf{x}^m$, $m \in \mathbb{N}$:

$$\sum z^i h_i(P) = \prod(1 - zu)^{-r} \quad . \tag{2.64}$$

Remark that Cauchy's formula (2.56) can be rewritten:

$$h_i(PQ) = \sum_{|\mu|=i} s_\mu(P)s_\mu(Q) \quad , \tag{2.65}$$

where the Schur functions $s_\mu(P)$ are defined by (2.69).

On the other hand, powers sums p_i provide simpler axioms than elementary symmetric functions or complete functions. Let $P = \sum ru$, $r \in \mathbb{R}$ and $u \in \mathbf{x}^m$, $m \in \mathbb{N}$ and let $i \geq 1$, then:

$$p_i(P) = \sum ru^i \quad . \tag{2.66}$$

In particular:

$$p_i(P + Q) = p_i(P) + p_i(Q) \quad , \quad p_i(PQ) = p_i(P) \times p_i(Q) \quad , \tag{2.67}$$

and moreover:

$$p_i(p_j(P)) = p_{ij}(P) \quad , \tag{2.68}$$

which is also known as the *plethysm* operation. Let us notice that $p_i(x_1 + x_2 + \cdots + x_n) = \sum x_j^i$ is indeed the power sum function p_i over the alphabet $\{x_1, x_2, \ldots, x_n\}$.

Two other (linear) bases of symmetric functions may also be considered: the Schur functions and the monomial functions, whose elements are commonly written down s_μ and m_μ, μ any partition. Usually, the Schur functions are defined by:

$$s_\mu := \det |h_{\mu_j} + j| \quad . \tag{2.69}$$

The monomial symmetric function m_μ is defined to be the sum of all monoms of degree μ over a given alphabet.

The `SFA` package is an implementation of this action of symmetric functions on the rings of polynomials.

Symmetric functions in the `SYMF` package have either the type `b`n, $n \geq 0$, when `b` is an algebraic basis of $\mathbb{R}[\mathbf{x}]$ —`b` $\in \{e, h, p\}$, or `b`$[n_1, n_2, \ldots, n_k]$, $n_1 \geq n_2 \geq \cdots n_k \geq 1$, when `b` is a linear basis of $\mathbb{R}[\mathbf{x}]$ —`b` $\in \{s, m\}$.

In `SFA`, all the symmetric functions have the shape `b`$[n_1, n_2, \ldots, n_k](A)$, $n_1 \geq n_2 \geq \cdots n_k \geq 1$, `b` $\in \{e, h, p, s, m\}$ and A is any alphabet.

When `b` is an algebraic basis, the object `b`$[n_1, n_2, \ldots, n_k](A)$ is considered as a product of `b`i's in the sense of `SYMF`.

For instance, the `SYMF` object:

```
2*e1^2*e2 + q*h1*s[3,2]
```

shall be written down:

```
2*e[2,1,1](A1) + q*h[1](A1)*s[3,2](A1)  ,
```

`A1` being a formal alphabet.

And we have `e`$[\mu](A) = e_\mu(A)$, `h`$[\mu](A) = h_\mu(A)$, `p`$[\mu](A) = p_\mu(A)$, and `s`$[\mu](A) = s_\mu(A)$, `m`$[\mu](A) = m_\mu(A)$.

2.7.2 Alphabets

Any formal alphabet in `SFA` must be written down `A`n, where n is any positive integer. Any other variable is considered as a constant, *i.e.* an element of $\mathbb{R}$, except symmetric functions, when they are encoded as operators on $\mathbb{R}[\mathbf{x}]$. Hence, the expression `p[3,2](B1)` becomes, according to (2.66—2.67), the real `B1`^{2}, although `p[3,2](A1)` stays as it is.

More generally, an alphabet is recursively defined to be either a sum of alphabets $\sum \alpha_i A_i, \alpha_i \in \mathbb{R}$, or a product of alphabets, or a symmetric function of `SFA`, or a formal alphabet `A`$_i$ or a scalar. Any variable $x_i \in \mathbf{x}$ can be encoded by a formal alphabet `A`$_i$ or through a special declaration.

2.7.3 Expanding the action of a symmetric function over $\mathbb{R}[\mathbf{x}]$

A symmetric function of `SFA`, when its alphabet has not the shape `A`n, can be expanded as described in 2.52—2.55, 2.59—2.62 and 2.66—2.67. The `SfAExpand` function realizes this operation:

```
ACE> SfAExpand( p[3,1,1](3) );

                                    27
```

or:

```
ACE> SfAExpand( p[1](A1-A2) * h[1,1](2*A1+k)^2 );

        16 p[1](A1) h[1, 1, 1, 1](A1) + 32 p[1](A1) k h[1, 1, 1](A1)

                                 2                                      3             4
             + 24 p[1](A1) k  h[1, 1](A1) + 8 p[1](A1) h[1](A1) k  + p[1](A1) k

             - 16 p[1](A2) h[1, 1, 1, 1](A1) - 32 p[1](A2) k h[1, 1, 1](A1)

                               2                                        3             4
             - 24 p[1](A2) k  h[1, 1](A1) - 8 p[1](A2) h[1](A1) k  - p[1](A2) k
```

One can compute more intricated examples, for instance the plethysm operation:

```
ACE> SfAExpand( m[2,1](k*A2 - p[3](2*A1)) );

              2                2
             k  m[2, 1](A2) + k  m[3](A2) - k m[3](A2) - 2 k m[6](A1) m[1](A2)

                 - 2 k m[3](A1) m[2](A2) + 4 m[6, 3](A1) + 6 m[9](A1)
```

If desired, one can insert some variables in an alphabet (e.g. in order to give it a size). First, the user must specify the names of the variables:

```
ACE> SfAVars({ y1, y2, {x} }):
```

means that `y1`, `y2` and all the `x`i's are now considered as variables. Thus:

```
ACE> SfAExpand( s[3,1](x1+x2+x3) + q*e[2,1](y1+y2) );

          3        3        2   2             2     2   2        3             2
        x3  x1 + x3  x2 + x3  x2  + 2 x1 x2 x3  + x3  x1  + x3 x2  + 2 x3 x1 x2

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

As in `SYMF`, any element of `SFA` can be developped on any recognized basis (e, h, p, s, m). The result may thereafter be expanded via `SfAExpand`. The available functions are: `ToeA`, `TohA`, `TopA`, `TosA` and `TomA`.

Let us take an example:

```
ACE> r := TomA( s[2,1](k*A1)*p[1,1](A1) );

              r := m[2, 1](k A1) m[2](A1) + 2 m[2, 1](k A1) m[1, 1](A1)

              + 2 m[1, 1, 1](k A1) m[2](A1) + 4 m[1, 1, 1](k A1) m[1, 1](A1)

ACE> SfAExpand(r);

                  - 1/3 m[2](A1) k p[3](A1) - 2/3 m[1, 1](A1) k p[3](A1)

                            3                                      3
          + 1/3 m[2](A1) k  p[1, 1, 1](A1) + 2/3 m[1, 1](A1) k  p[1, 1, 1](A1)
```

The indexing vectors of the symmetric functions do not need to be partitions; they are thus reordered according to the corresponding rule for the basis in cause:

$$\begin{aligned} \mathtt{s}[..,i,j,..](P) &= -s[..,j-1,i+1,..](P), && \text{if } \mathtt{i} < j, \\ b[..,i,j,..](P) &= b[..,j,i,..](P), && \text{if } \mathtt{i} < j, b \neq \mathtt{s} \quad . \end{aligned} \tag{2.70}$$

```
ACE> TosA( s[1,4](A1) + e[2,3](A2) );

       - s[3, 2](A1) + s[1, 1, 1, 1, 1](A2) + s[2, 1, 1, 1](A2) + s[2, 2, 1](A2)
```

One can also choose to apply a basis change on specific alphabets:

```
ACE> TopA( s[2](A1) + e[2,3](A2)*m[2,1](q*A3), [A1, q*A3]  );

                 1/2 p[2](A1) + 1/2 p[1, 1](A1) + e[2, 3](A2) p[2, 1](q A3)

                                - e[2, 3](A2) p[3](q A3)
```

2.8 Noncommutative symmetric functions

Starting from the quasi-determinants of Gelfand and Retakh [49, 50], a noncommutative theory of symmetric functions [48] has been developed in such a way that most of the classical applications can be lifted to the noncommutative case. In particular, the character theory of the symmetric group has a natural noncommutative analogue, in which the role of the character ring is played by Solomon's descent algebra [62]. The main applications of this correspondence are connected with Lie series and free Lie algebras (*cf.* [47, 60]). Also, the algebra of noncommutative symmetric functions, together with its dual Hopf algebra, which is Gessel's algebra of quasi-symmetric functions [51, 58] encode the representation theory of the 0-Hecke algebra of the symmetric group [46]. The *algebra of formal noncommutative symmetric functions* can be defined as the free associative algebra $\mathbf{Sym}=\mathbb{C}\langle\Lambda_1,\Lambda_2,\ldots\rangle$ generated by

an infinite sequence of noncommuting indeterminates Λ_k, called *elementary* functions. It is convenient to set $\Lambda_0 := 1$. Let t be another indeterminate, commuting with the Λ_k. One introduces the generating series:

$$\lambda(t) = \sum_{k\geq 0} \Lambda_k \ t^k \quad , \tag{2.71}$$

$$\sigma(t) = \sum_{k\geq 0} S_k \ t^k = \lambda(-t)^{-1} \quad , \tag{2.72}$$

$$\Phi(t) = \sum_{k\geq 1} \frac{1}{k}\Phi_k \ t^k = \log \sigma(t) \quad . \tag{2.73}$$

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

$$\psi(t) = \sum_{k\geq 1} \Psi_k \ t^{k-1} \quad , \tag{2.74}$$

defined by the equation:

$$\psi(t) = \lambda(-t)\frac{d}{dt}\sigma(t) \quad . \tag{2.75}$$

The identities (2.73)-(2.75), between formal power series imply relations between their coefficients. We have:

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

$$\sum_{k=0}^{n-1} S_k \Psi_{n-k} = nS_n \quad ,$$

$$\sum_{k=1}^{n}(-1)^{k-1}\Psi_k\Lambda_{n-k} = n\Lambda_n \quad .$$

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

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

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

$$F^I := F_{i_1}F_{i_2}\ldots F_{i_k} \quad . \tag{2.76}$$

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

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

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

whose diagram is the conjugate (in the sense of skew diagrams) of the diagram of I. For instance, the conjugate composition of $(3,1,2,1)$ is $(2,3,1,1)$.

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

One also introduces the basis, (R_I) of *ribbon Schur* functions, originally defined in terms of quasi-determinants in [48]. Ribbon Schur functions can also be defined by one of the following two equivalent systems of equations:

$$S^I = \sum_{J \preceq I} R_J, R_I = \sum_{J \preceq I} (-1)^{\ell(I)-\ell(J)} S^J \quad . \tag{2.77}$$

Multiplication of ribbon Schur functions is given by the following formula whose commutative version is due to MacMahon [56]:

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

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

2.8.1 Change of basis

Transition matrices between the various bases of **Sym**, as well as functions `To`X (X={`S`, `L`, `R`, `Ph`, `Ps`}) giving the expansion of any element of **Sym** in one of these bases have been implemented. We use algorithms described in [48].

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

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

Computations for all compositions of 3 give:

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

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

```
ACE> NcsfMat(3, 'L', 'S');

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

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

```
ACE> ToR((1-q)^4*S[2,1]+t*L[3], 'noexpand');

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

The standard product of **Sym** (viewed as a free associative algebra) convolution product on graded endomorphisms corresponds to the convolution product on graded endomorphisms of a tensor algebra [60]. In *Maple*, the multiplication operator `*` is commutative, hence one has to represent this product by means of the infix operator `&*`.

2.8.2 Descent algebras

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

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

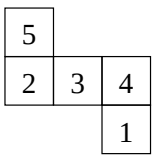
The elements $D_{=A}$ are linearly independent. Solomon shown that the subspace of $\mathbb{Q}[S_n]$ generated by the elements $D_{=A}$ is a subalgebra of $\mathbb{Q}[S_n]$ ([47],[62]). In fact, there exist nonnegative integers d_{AB}^C such that:

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

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

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

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



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

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

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

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

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

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

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

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

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

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

2.8.3 Lie idempotents

A *Lie idempotent* is an idempotent of the algebra of the symmetric group which projects a homogeneous component of the free associative algebra onto the free Lie algebra (see *e.g.* [60]). It turns out that most interesting Lie idempotents belong to Solomon's algebra, which allows one to interpret them as noncommutative symmetric functions. For example, it has been shown by Klyachko [52] that for a primitive n-th root of unity ε, the element:

$$\kappa_n = \frac{1}{n} \sum_{\sigma \in S_n} \varepsilon^{\mathrm{maj}(\sigma)} \sigma \quad , \tag{2.78}$$

is a Lie idempotent, where maj (σ) denotes the *major index* of a permutation. By definition, for $\sigma \in S_n$ with Des (σ) $= \{d_1, \ldots, d_r\}$, maj(σ) $= d_1 + \cdots + d_r$. To prove the previous assertion, one can rewrite (2.78) in terms of descent sets. Hence:

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

It follows that:

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

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

```
ACE> C:=ListCompo(4):
ACE> K:=0:
ACE> for x in C do K:=K+q^Compo2MajorIndex(x)*R[op(x)] od:
ACE> K:=subs(q=I, K)/4;

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

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

We check that it is actually an idempotent:

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

                                          0
```

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

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

Then applying α to θ_n, we find

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

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

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

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

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

                                          0
```

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

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

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

                                    0
```

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

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

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

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

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

```
ACE> r := NcsfTransAlpha(S[3], 'direct',  'q'):
ACE> r := ToR((1-q)*(1-q^2)*(1-q^3)*r, 'collect'):
ACE> map(simplify, r);

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

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

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

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

One can show that the element:

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

is a Lie idempotent [45, 53], interpolating between several known ones:

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

                                        1/4 Ph[4]

ACE> subs(q=I, P);

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

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

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

Following a recent work by Malvenuto and Reutenauer [58] the dual bialgebra (in the graded sense) of **Sym**, denoted by $\mathbf{Sym}^*$, can be identified with another interesting generalization of symmetric functions, that is, the algebra **Qsym** of *quasi-symmetric* functions, whose definition is due to Gessel [51].

2.8.4 Quasi-symmetric functions

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

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

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

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

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

2.8.5 Pairing between Sym and Qsym

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

$$\langle M_I, S^I \rangle = \delta_{I\,J} \quad . \tag{2.79}$$

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

$$\left.\begin{array}{r} \langle {}^{Q}\Phi_I, \dfrac{\Phi^J}{\Pi(J)} \rangle \\ \langle {}^{Q}\Psi_I, \dfrac{\Psi^J}{\Pi_u(J)} \rangle \\ \langle F_I, R_J \rangle \\ \langle E_I, \Lambda^J \rangle \end{array}\right\} = \delta_{IJ} \quad , \tag{2.80}$$

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

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

```
ACE> QsfMat(3, 'F', 'M');

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

ACE> QsfMat(3, 'M', 'F');

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

On **Qsym**, the usual product can be computed as follows (cf. [57]): given $|I| = m$ and $|J| = n$, one considers $u \in S_{1..m}$ and $v \in S_{m+1..m+n}$ with $c(u) = I$ and $c(v) = J$, let:

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

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

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

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

```
ACE> ToM(F[2,1] &* E[2]);
             - M[2, 3] - M[1, 1, 3] - 2 M[2, 2, 1] - M[1, 1, 2, 1] - M[2, 1, 2]
             - M[1, 1, 1, 2] - M[2, 1, 1, 1] - M[4, 1] - M[3, 1, 1] - M[1, 3, 1]
             - M[1, 2, 1, 1]
ACE> ToF(F[3] &* F[1,2,1]);
             F[1, 1, 4, 1] + F[4, 2, 1] + F[3, 3, 1] + F[3, 1, 2, 1] + F[3, 2, 2]
           + F[3, 2, 1, 1] + F[2, 4, 1] + F[2, 3, 2] + F[2, 3, 1, 1] + F[2, 1, 3, 1]
           + F[2, 1, 2, 2] + F[2, 1, 2, 1, 1] + F[2, 2, 3] + 2 F[2, 2, 2, 1]
           + F[2, 2, 1, 2] + F[1, 5, 1] + F[1, 4, 2] + F[1, 4, 1, 1] + F[1, 2, 2, 1, 1]
           + F[1, 3, 3] + 2 F[1, 3, 2, 1] + F[1, 3, 1, 2] + F[1, 1, 3, 2]
           + F[1, 1, 3, 1, 1] + F[1, 1, 2, 3] + F[1, 1, 2, 2, 1] + F[1, 1, 2, 1, 2]
           + F[1, 2, 4] + 2 F[1, 2, 2, 2] + F[1, 2, 1, 3] + 2 F[1, 2, 3, 1]
```

From the definition of quasi-symmetric functions, it follows that a symmetric function is in particular a quasi-symmetric function. According to Gessel [51], if f is a symmetric function, then:

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

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

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

For example, one can compute the development of s_{32} on the (F_I) basis by the following instruction:

```
ACE> Sf2Qsf(s[3,2]);
                    F[3, 2] + F[2, 3] + F[2, 2, 1] + F[1, 3, 1] + F[1, 2, 2]
```

The $p(n) \times 2^{n-1}$ matrix whose rows give the coefficients of the Schur functions s_λ on the basis (F_I) can be interpreted as the decomposition matrix of the 0-Hecke algebra $H_n(0)$ (cf. [43] and [46]).

2.9 Hyperoctahedral groups

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

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

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

$$(0)(101)(21012)\ldots(n-1\ldots0\ldots n-1) \quad ,$$

factorized into canonical factors of the type

$$(k\ldots101\ldots k) \quad ,$$

in which i stands for the generator s_i.

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

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

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

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

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

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

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

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

ACE> BnCode2ListRd([1,3,5,7], 'nb');

                                  24024
```

The function `BnListCode` returns the $2^n n!$ codes of the elements of B_n. Moreover, the function `BnGenCode` is available to generate step by step all codes of B_n.

Given a code c, the command `BnCode2Interval`(c) returns the list of all codes which are below c with respect to the Bruhat order:

```
ACE> BnCode2Interval([1,3,3]);

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

         [1, 1, 1], [0, 2, 4], [0, 1, 4], [0, 2, 3], [0, 2, 2], [0, 1, 3], [0, 1, 2],

         [0, 2, 1], [0, 1, 1], [1, 0, 4], [0, 0, 4], [1, 0, 3], [1, 0, 2], [0, 0, 3],

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

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

Group operations are defined on the elements given by their codes. The function `BnMultCode` is the multiplication of 2 elements of B_n whereas `BnCode2Inv` inverts a given element.

There are other descriptions of B_n. We give two of them. One can embed B_n into the symmetric group $\mathfrak{S}_{2n}$. It amounts to interpret the generators `BnTranspoPalin` as follows:

$$\begin{cases} s_0 &= t_n \quad , \\ s_1 &= t_{n-1}t_{n+1} \quad , \\ \vdots & \vdots \quad \vdots \quad , \\ s_{n-1} &= t_1 t_{2n-1} \quad , \end{cases}$$

where the t_i are the simple transpositions generating $\mathfrak{S}_{2n}$. The image of B_n consists into palindromic permutations $v := [v_1, \ldots, v_{2n}]$ that we call `Palin`, *i.e.* such that

$$v_1 + v_{2n} = v_2 + v_{2n-1} = \cdots = v_n + v_{n+1} = 2n + 1 \quad .$$

We provide the different decodings `BnCode2Palin` and `BnPalin2Code`. Of course, we can use the functions in the `SG` package to handle palindromic permutations.

Another description of B_n is as the set of signed permutations of $\{1, \ldots, n\}$, that we call `Bar`. Let $u := [u_1, \ldots, u_n]$, then u is a signed permutation when $[|u_1|, \ldots, |u_n|]$ is a permutation of $\{1, \ldots, n\}$.

The right action of $s_i, i > 0$ on a signed permutation $u := [u_1, \ldots, u_n]$ consists into transposing u_i and u_{i+1}:

$$[u_1, \ldots, u_i, u_{i+1}, \ldots, u_n]\, s_i \;=\; [u_1, \ldots, u_{i+1}, u_i, \ldots, u_n] \;, \quad i > 0 \quad .$$

The generator s_0 transforms u_1 into $-u_1$:

$$[u_1, \ldots, u_n]\, s_0 \;=\; [-u_1, \ldots, u_n] \quad .$$

The left action of $s_i, i > 0$ is to exchange values i and $i + 1$ leaving signs as they were in the initial permutation. The generator s_0 acts on u by changing the value 1 into -1, or the value -1 to 1.

We provide the different conversions `BnCode2Bar` and `BnBar2Code` as well as the direct multiplication between signed permutations (`BnMultBar`).

`Bar` permutations can also be represented by permutation matrices where some entries 1 can be replaced by -1. For that purpose, we give the two functions `BnBar2Mat` and `BnMat2Bar`.

2.10 Algebras related to hyperoctahedral groups

As in the case of the symmetric group, we provide different algebras associated to B_n, as well as their actions on the ring of polynomials:

Function name prefix	**Algebra**
`BnGa`	is the group algebra of B_n;
`BnNca`	is the nilCoxeter algebra of B_n;
`BnIdca`	is the idCoxeter algebra of B_n.

Elements of these algebras are sums of elements of B_n (given as codes) with any type of coefficients. Addition (`BnaAdd`, `BnaMinus`) is provided once for all these algebras.

For all of these algebras, we have a different multiplication:

Function	**Object**
`BnGaMult`	uses the multiplication in B_n;
`BnNcaMult`	uses the multiplication where the relations $s_i^2 = 1, i \geq 0$ have been changed into $\partial_i^B \partial_i^B = 0$;
`BnIdcaMult`	uses the multiplication where the relations $s_i^2 = 1, i \geq 0$ have been changed into $\pi_i^B \pi_i^B = -\pi_i^B, i \geq 0$.

The group algebra (`BnGa`) acts (`BnGaOnPol`) on the ring of polynomials in $x_1, \ldots, x_n$ (by default). The $\{s_i, i = 1, \ldots, n-1\}$ are the usual generators of $\mathfrak{S}_n$, *i.e.* s_i transposes (x_i, x_{i+1}), and s_0 changes x_1 into $-x_1$.

The nilCoxeter algebra (`BnNca`) acts (`BnNcaOnPol`) on the ring of polynomials in $x_1, \ldots, x_n$ (by default). The $\{\partial_i^B, i = 1, \ldots, n-1\}$ are the usual divided differences, *i.e.* ∂_i^B acts by:

$$p \quad \rightarrow \quad \frac{p - s_i(p)}{x_i - x_{i+1}} \quad .$$

The extra divided difference ∂_0^B is defined by:

$$p \quad \rightarrow \quad \frac{p - s_0(p)}{2x_1} \quad .$$

The idCoxeter algebra (`BnIdca`) acts (`BnIdcaOnPol`) on the ring of (Laurent) polynomials in $x_1, \ldots, x_n$ (by default). The $\{\pi_i^B, i = 1, \ldots, n-1\}$ are the usual generators:

$$p \quad \rightarrow \quad \frac{p - s_i(p)}{x_i - x_{i+1}} x_{i+1} \quad .$$

The extra generator π_0^B is defined by:

$$p \quad \rightarrow \quad \frac{x_1 p - \frac{1}{x_1} p'}{1/x_1 - x_1} \quad ,$$

where p' is obtained from p by changing x_1 into $1/x_1$. For example:

$$\pi_0^B(x_1^k) = -(x_1^k + x_1^{k-2} + \cdots + x_1^{-k+2} + x_1^{-k}) \quad .$$

```
ACE> r := BnIdcaOnPol(B[1], x1^6);

                              12     10     8     6     4     2
                             x1   + x1   + x1  + x1  + x1  + x1  + 1
                    r := - ------------------------------------
                                              6
                                             x1

ACE> expand(r);

                                 6     4     2        1     1     1
                               - x1  - x1  - x1  - 1 - --- - --- - ---
                                                       2     4     6
                                                      x1    x1    x1
```

The elements of the algebras can also be decoded in terms of either signed permutations (`BnaCode2Bar`) or palindromic permutations (`BnaCode2Palin`).

In both group algebra and nilCoxeter algebra of B_n, one defines a Yang-Baxter basis (`BnGaYang`, `BnNcaYang`), indexed by codes, depending upon parameters $x_1, \ldots, x_n$ (by default) which can be specialized to $1, \ldots, n$ by the option `'num'`:

```
ACE> BnGaYang(B[1,3]);

                                          3       2
        - 2 (x2 + x1) x2 B[0, 2] + (2 x1  - 2 x2  x1 - 2 x1 - 2 x2) B[1, 0]

                                                    2     2
        - 4 (x1 - x2) (x2 + x1) x1 x2 B[1, 3] + (- x1  + x2  + 1 + 4 x2 x1) B[0, 0]

                 2                                  2              2
        + (- 2 x2  - 2 x2 x1) B[1, 1] + (- 4 x2 x1  + 2 x2 + 4 x2  x1) B[0, 1]

        + 2 (x1 - x2) (x2 + x1) x2 B[0, 3] + 4 (x2 + x1) x1 x2 B[1, 2]

ACE> BnGaYang([1,3], 'num');

                  - 12 B[0, 2] - 12 B[1, 0] + 12 B[0, 0] - 12 B[1, 1]

                  + 12 B[0, 1] + 24 B[1, 2] - 12 B[0, 3] + 24 B[1, 3]

ACE> BnNcaYang([1,3]);

        - 2 (x2 + x1) x2 B[0, 2] + (- 2 x1 - 2 x2) B[1, 0]

                                                           2
        - 4 (x1 - x2) (x2 + x1) x1 x2 B[1, 3] + B[0, 0] + (- 2 x2  - 2 x2 x1) B[1, 1]

        + 2 x2 B[0, 1] + 2 (x1 - x2) (x2 + x1) x2 B[0, 3] + 4 (x2 + x1) x1 x2 B[1, 2]
```

2.11 Words and tableaux

They are two dimensional arrays obtained by filling the *Ferrers diagram* of a partition with numbers, in such a way that numbers are weakly increasing in rows from left to right, and strictly increasing from bottom to top. A tableau is *standard* if the numbers used are $1, 2, 3, \ldots$ without repetition. Tableaux also code linear bases of irreducible representations of the symmetric and linear groups.

We represent a tableau by the sequence of its rows:

3	5	
2	4	
1	1	3

will be coded in the package by `Tab:=[ [3,5] ,[2,4], [1,1,3] ]`. We recover from it the planar tableau using the `Tab2Mat` function:

```
ACE> Tab:=[ [3,5] ,[2,4], [1,1,3] ];

                          Tab := [[3, 5], [2, 4], [1, 1, 3]]

ACE> Tab2Mat(Tab);

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

A tableau can be considered as the *word* (`Tab2Word`) obtained by concatenating its rows. From this word, one reconstructs without ambiguity the tableau by cutting the word at each decrease.

More generally, Schensted has described an algorithm (a "bumping process") to transform any word into a tableau: if `Tab` is the tableau obtained by this algorithm from the word w, then the tableau corresponding to the word wx is obtained by `Bump(Tab, x)`:

```
ACE> r := Bump(Tab, w[7,8,6,7]);

                  r := [[3, 5], [2, 4, 7, 8], [1, 1, 3, 6, 7]]

ACE> Tab2Mat(r);

                                [ 3  5          ]
                                [               ]
                                [ 2  4  7  8    ]
                                [               ]
                                [ 1  1  3  6  7 ]
```

Two words which give the same tableau are said to be *plactically* equivalent. As shown by Knuth, this equivalence on words is induced from the following equivalence on words of length 3 ($i < j < k$):

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

By pushing, thanks to the "jeu de taquin", a planar tableau into a north-east corner, one gets a *contretableau* which will be coded by the sequence of its columns, from left to right, each column being read from top to bottom. The system provides the functions `CTab2Word`, `CTab2Mat`, ..., which are exactly the analogues of the functions for tableaux. There is also an algorithm to get from any word a contretableau (`Word2CTab`), the elementary step being given by `BumpC`:

```
ACE> CTab:=Word2CTab(w[4,2,3,4,1,2,3,2]);

                        CTab := [[2, 1], [4, 3, 2], [4, 3, 2]]

ACE> CTab2Mat(CTab);

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

ACE> CTab2Mat( BumpC(CTab, w[7,8,6,7]) );

                                  [ 2  4  4  7  8 ]
                                  [               ]
                                  [ 1  3  3  6  7 ]
                                  [               ]
                                  [          2  2 ]
```

Another way of coding standard tableaux is by *Yamanouchi words*, *i.e.* words w such that for each factorisation $w = u\,v$, then $|v|_1 \geq |v|_2 \geq |v|_3 \geq \cdots$, where $|v|_i$ denote the number of times the letter i appears in v. This coding just tells in which rows in the standard tableau are the letters $n, n-1, \ldots, 2, 1$. One has the inverse function, from Yamanouchi words to standard tableaux `Yama2StdTab`.

The *shape* of a tableau is the list of the lengths of its successive rows, and thus is a partition which is given by the function `Tab2Part`. The `ListTab` function returns the list of all tableaux of a given shape and a given evaluation:

```
ACE> r := ListTab([3,2], [0,1,0,0,1,2,0,0,1]);

          r := [[[6, 9], [2, 5, 6]], [[5, 9], [2, 6, 6]], [[6, 6], [2, 5, 9]]]

ACE> map(Tab2Mat, r);

                          [ 6  9    ]  [ 5  9    ]  [ 6  6    ]
                          [[        ], [        ], [        ]]
                          [ 2  5  6 ]  [ 2  6  6 ]  [ 2  5  9 ]
```

Given a partition, the function `Part2ListStdTab` enumerates all the standard tableaux of shape this partition:

```
ACE> r := Part2ListStdTab([3,1]);

               r := [[[2], [1, 3, 4]], [[3], [1, 2, 4]], [[4], [1, 2, 3]]]

ACE> map(Tab2Mat, r);

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

The equivalent enumeration of right Yamanouchi words is given by `ListYama`. It computes all right Yamanouchi words corresponding to a given partition part. These words are exactly the words which give the tableaux [..., [3...3], [2...2], [1...1]] by Schensted's algorithm:

```
ACE> r := ListYama([4,2,1], 'right');

    r := [w[1, 1, 2, 1, 3, 2, 1], w[1, 2, 1, 1, 3, 2, 1], w[2, 1, 1, 1, 3, 2, 1],

          w[2, 1, 1, 3, 1, 2, 1], w[2, 1, 3, 1, 1, 2, 1], w[2, 3, 1, 1, 1, 2, 1],

          w[3, 2, 1, 1, 1, 2, 1], w[3, 2, 1, 1, 2, 1, 1], w[3, 2, 1, 2, 1, 1, 1],

          w[3, 2, 2, 1, 1, 1, 1], w[2, 3, 1, 1, 2, 1, 1], w[2, 3, 1, 2, 1, 1, 1],

          w[2, 3, 2, 1, 1, 1, 1], w[2, 1, 3, 1, 2, 1, 1], w[2, 1, 3, 2, 1, 1, 1],

          w[2, 1, 1, 3, 2, 1, 1], w[1, 2, 1, 3, 1, 2, 1], w[1, 2, 3, 1, 1, 2, 1],

          w[1, 3, 2, 1, 1, 2, 1], w[3, 1, 2, 1, 1, 2, 1], w[3, 1, 2, 1, 2, 1, 1],

          w[3, 1, 2, 2, 1, 1, 1], w[1, 3, 2, 1, 2, 1, 1], w[1, 3, 2, 2, 1, 1, 1],

          w[1, 2, 3, 1, 2, 1, 1], w[1, 2, 3, 2, 1, 1, 1], w[1, 2, 1, 3, 2, 1, 1],

          w[1, 1, 2, 3, 1, 2, 1], w[1, 1, 3, 2, 1, 2, 1], w[1, 3, 1, 2, 1, 2, 1],

          w[3, 1, 1, 2, 1, 2, 1], w[3, 1, 1, 2, 2, 1, 1], w[1, 3, 1, 2, 2, 1, 1],

          w[1, 1, 3, 2, 2, 1, 1], w[1, 1, 2, 3, 2, 1, 1]]

ACE> r := map(Word2Tab, r):
```

```
ACE> Tab2Mat( op( {op(r)} ) );

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

A tableau t gives a word in non commuting letters $1, 2, \ldots$, but also a monomial x^t by replacing each letter i by the variable x_i. Now, through this transformation, the sum of all tableaux of a given shape *part* gives a polynomial $\sum x^t$ which is exactly the Schur function of index the partition conjugate to *part*. The best way to explain why the sum of all tableaux gives a symmetric function is to introduce an action of the symmetric group on words, which sends tableaux to tableaux of the same shape, and is compatible with plactic equivalence. This action is given by the `SgaOnFree` function:

```
ACE> SgaOnFree(A[1,3,2], w[2,2,3,1,1,2,2,3]);

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

Now, the set of tableaux of a given shape is globally invariant under this action. Of course, the induced action on the monomials x^t is the usual action of the symmetric group on the variables $x_1, x_2, \ldots$

Finally, to each word one associates two integral vectors, its *charge vector* (that can be computed by the `Word2Charge` function) and its *cocharge vector* (`Word2CoCharge`):

```
ACE> Word2Charge(w[3,2,2,1,4,1,1,3], 'vector');

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

ACE> Word2CoCharge(w[3,2,2,1,4,1,1,3], 'vector');

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

These two vectors are compatible with Schensted (`Schensted`) algorithm and the plactic equivalence, as well as with cyclic permutations on words. The sum of the entries in the cocharge vector is a number which is the height of the tableau, when the set of all tableaux is made into a rank poset by the operation of deleting the first letter of a tableau and inserting from the right:

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

```
ACE> lt:=ListTab([3,2,1], [1,2,2,1]);

             lt := [[[4], [3, 3], [1, 2, 2]], [[4], [2, 3], [1, 2, 3]],

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

ACE> map(Tab2Mat, lt);

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

ACE> r := map(Tab2Word, lt);

r := [w[4, 3, 3, 1, 2, 2], w[4, 2, 3, 1, 2, 3], w[3, 2, 4, 1, 2, 3], w[3, 2, 3, 1, 2, 4]]

ACE> map(Word2Charge, r);

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

On the other hand, the sum $\sum q^{chrg(t)} x^t$ on all tableaux of a given shape *part*, where $chrg(t)$ is the sum of the entries of the charge vector, is the *Hall-Littlewood* polynomial Q'_{part}.

2.12 Schubert polynomials

Let $\mathbf{x} := \{x_1, \ldots, x_n\}$ be a totally ordered set of variables. We denote by $s_i, i = 1, \ldots, n-1$ the elementary transposition seen as the operator on $\mathbb{Z}[\mathbf{x}]$ that interchanges x_i and x_{i+1} and fixes all other variables. Elementary transpositions satisfy the following relations:

$$s_i s_j = s_j s_i \qquad \text{if } |i-j| > 1 \quad , \tag{2.81}$$

$$s_i s_j s_i = s_j s_i s_j \qquad \text{if } j = i+1 \quad , \tag{2.82}$$

known as the famous *braid* relations. We also consider another operator $\partial_i, i = 1, \ldots, n-1$ on the polynomial ring defined by:

$$\mathbb{Z}[x_1, \ldots, x_n] \ni \quad p \longrightarrow \partial_i(p) := \frac{p - s_i\, p}{x_i - x_{i+1}} \quad , \tag{2.83}$$

where $s_i\, p$ stands for the polynomial p in which variables x_i and x_{i+1} have been exchanged. This linear operator was introduced by Newton and is called Newton's *divided difference*. In 1973, Bernstein, Gelfand & Gelfand and Demazure observed that divided differences fulfil, like elementary transpositions, the braid relations:

$$\partial_i \partial_j = \partial_j \partial_i \qquad \text{if } |i-j| > 1 \quad , \tag{2.84}$$

$$\partial_i \partial_j \partial_i = \partial_j \partial_i \partial_j \qquad \text{if } j = i+1 \quad . \tag{2.85}$$

It is well-known that the symmetric group $\mathfrak{S}_n$ of degree n is generated by elementary transpositions $s_i, i = 1, \ldots, n-1$, so that for any permutation $\mu \in \mathfrak{S}_n$, one can find an expression of μ as a

product $s_{i_1} \dots s_{i_k}$ of elementary transpositions. Such an expression is far from being unique—because elementary transpositions satisfy relations (2.81), (2.82) together with the extra relation $s_i^2 = 1, i = 1, \dots, n-1$—but is said to be *reduced* when k is exactly the *number of inversions* of the permutation μ. Now, relations (2.84) and (2.85) imply that for any permutation $\mu \in \mathfrak{S}_n$, there exists a divided difference ∂_μ (Newton's case corresponding to elementary transpositions). Indeed, we may define unambiguously for any reduced decomposition—because relation $s_i^2 = 1$ is replaced by $\partial_i^2 = 0$ in the case of divided differences—of $\mu = s_{i_1} \dots s_{i_k}$:

$$\partial_\mu := \partial_{i_1} \dots \partial_{i_k} \quad . \tag{2.86}$$

Schubert polynomials, introduced in 1982 by Lascoux and Schützenberger [76] as a combinatorial tool for some issues in algebraic geometry, are the images under divided differences of the monomial $x^\delta = x_1^{n-1} x_2^{n-2} \dots x_n^0$:

$$\mathbf{X}_\mu := \partial_{\mu^{-1}\omega}(x^\delta) \quad , \tag{2.87}$$

where $\omega = (n, n-1, \dots, 2, 1)$ denotes the longest element of $\mathfrak{S}_n$. These are polynomials in n variables, indexed by permutations $\mu \in \mathfrak{S}_n$ which provide a convenient basis of the polynomial ring regarded as a free module over the ring of symmetric polynomials. Schubert polynomials are non-negative integral linear combinations of the monomials of the obvious basis $\{x_1^{i_1} \cdots x_n^{i_n} : 0 \le i_k \le n-k,\ 1 \le k \le n\}$, over which they present many advantages. Their geometric interpretation is the source of many non trivial identities that are liftings at the level of polynomials of classical or recent formulas of the Schubert calculus, whose original version was only valid in a certain quotient of $\mathbb{Z}[x_1, \dots, x_n]$.

This quotient is the so-called ring of coinvariants of the symmetric group. It is the quotient $R = \mathbb{Z}[x_1, \dots, x_n]/I^+$ of the ring of polynomials by the ideal I^+ generated by the symmetric polynomials without constant term. Geometrically, R can be interpreted as the cohomology ring $H^*(\mathcal{F}_n, \mathbb{Z})$ of the variety of complete flags in $\mathbb{C}^n$. With this interpretation, the class of the Schubert polynomial $\mathbf{X}_\mu$ is the Poincaré dual of the Schubert cycle $[\overline{B}_\mu]$.

Schubert polynomials are compatible with the embedding $\mathfrak{S}_n \hookrightarrow \mathfrak{S}_{n+1}$ that maps $(\mu(1), \dots, \mu(n))$ onto $(\mu(1), \dots, \mu(n), n+1)$. When $\mathfrak{S}_n$ is identified with its image in $\mathfrak{S}_{n+1}$, Schubert polynomials form a basis of the polynomial ring $\mathbb{Z}[x_1, x_2, \dots]$ in an infinite sequence of indeterminates. But the most important key is that Schubert polynomials are compatible with divided differences, which turn to be the fundamental operators of the Schubert calculus. This was realized independently by Bernstein, Gelfand & Gelfand and by Demazure, who introduced such operators for any Weyl group and applied them to the description of the cohomology of generalized flag manifolds G/P where G is a complex semisimple Lie group and P a parabolic subgroup.

Algebraic calculation on Schubert polynomials involve both manipulations of combinatorial objects such as permutations, codes or diagrams, and a scalar product with self-adjoint operators. This combinatorics includes the combinatorics of Schur functions—partitions being considered as *Lehmer codes* of special permutations—and is moreover very similar to it. With Schubert polynomials, one can compute directly on polynomials, not passing to the quotient by the ideal I^+. The resulting theory appears as the natural generalization of the theory of Schur functions (which correspond to Grassmann manifolds) and provides some deep insights in the combinatorics of permutations.

2.12.1 Symmetric groups and divided difference operators

Consider a polynomial p in the variables $\dots, x_i, \dots, x_j, \dots$, then the divided difference of p with respect to the variables x_i and x_j is defined by:

$$\partial_{x_i, x_j}(f) := \frac{f(\dots, x_i, \dots, x_j, \dots) - f(\dots, x_j, \dots, x_i, \dots)}{x_i - x_j} \quad . \tag{2.88}$$

This operator which maps polynomials onto polynomials and decreases the degrees by 1, satisfies the following properties:

$$\partial_{x_i,x_j} s_{x_i,x_j} = -\partial_{x_i,x_j} \quad , \tag{2.89}$$

$$s_{x_i,x_j} \partial_{x_i,x_j} = \partial_{x_i,x_j} \quad , \tag{2.90}$$

$$\partial^2_{x_i,x_j} = 0 \quad , \tag{2.91}$$

$$\partial_{x_i,x_j} \partial_{x_j,x_k} \partial_{x_i,x_j} = \partial_{x_j,x_k} \partial_{x_i,x_j} \partial_{x_j,x_k} \quad , \tag{2.92}$$

where s_{x_i,x_j} denotes the transposition that interchanges the variables x_i and x_j.

Consider now the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates. For each $i \geq 1$, one can define the two operators s_i and ∂_i by:

$$s_i := s_{x_i,x_{i+1}} \quad ,$$

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

so that, if $p \in \mathbb{Z}[x_1, x_2, \ldots]$ and $i \geq 1$, then:

$$\partial_i(p) := \frac{p - s_i\, p}{x_i - x_{i+1}} \quad . \tag{2.93}$$

The resulting is a polynomial which is symmetric in x_i and x_{i+1}. It is very helpful to remark that both operators s_i and ∂_i separately satisfy the braid relations. This allows to define operators ∂_μ for any permutation $\mu \in \mathfrak{S}_n$. When considering that $\mathfrak{S}_n$ is embedded into $\mathfrak{S}_{n+1}$ by adding a fixed point $n+1$, then we can define the group:

$$\mathfrak{S}_\infty := \lim_{i \to \infty} \mathfrak{S}_i \quad ,$$

of permutations of the set of positive integers that fix all but a finite number of them. Thus, one can more generally consider operators ∂_μ for any permutation $\mu \in \mathfrak{S}_\infty$.

Let $p \in \mathbb{K}[x_1]$ be a polynomial in the unique variable x_1 of degree less than n and let also $\{y_1, \ldots, y_{n-1}\}$ denote any set of elements of $\mathbb{K}$. Then, the Newton interpolation formula can be written:

$$p(x_1) = p(y_1) + \sum_{i=1}^{n-1} \partial_i \ldots \partial_1(p)(x_1 - y_1) \ldots (x_1 - y_i) \quad , \tag{2.94}$$

in which divided differences act on the alphabet $\{y_1, \ldots, y_{n-1}\}$.

A couple (i, j) of indices such that $1 \leq i < j \leq n$ and $\mu(i) > \mu(j)$ is called an *inversion* of μ. The *code* of μ is the vector $c(\mu) := (c_1, \ldots, c_n)$ of non-negative integers, c_i being the number of j such that (i, j) is an inversion of μ. The code of a permutation describes its inversions and is another coding of the permutation. In other words, there is a one-to-one correspondence between elements of $\mathfrak{S}_n$ and vectors of $\mathbb{N}^n$ or more generally between permutations of $\mathfrak{S}_\infty$ and vectors of $\mathbb{N}^{(\mathbb{N}^*)}$. It is also useful to consider the partition $\lambda(\mu)$ associated with the permutation μ. The parts of this partition are the non-zero entries of the vector $c(\mu)$, arranged in weakly decreasing order.

We introduce a special class of permutations known as the vexillary class. A permutation μ is said to be *vexillary* if there do not exist four positions $1 \leq i < j < k < l$ such that $\mu(j) < \mu(i) < \mu(l) < \mu(k)$. For instance, all permutations of $\mathfrak{S}_1, \mathfrak{S}_2$ and $\mathfrak{S}_3$ are vexillary, while the only permutation in $\mathfrak{S}_4$ which is not vexillary is $\mu = (2, 1, 4, 3)$. Testing whether a permutation is vexillary or not is not so trivial but among vexillary permutations one has sub-families for which this test is easier. Let $\mu \in \mathfrak{S}_n$ and $c(\mu) = (c_1, \ldots, c_n)$ denote its code. Then, μ is said to be *dominant* when $c(\mu)$ is a partition, *i.e.* a weakly decreasing sequence of non-negative integers. On another hand, μ is said to be *Grassmannian*

if and only if $c_1 \leq \cdots \leq c_k$ and $\forall i > k, c_i = 0$ for some k. In fact, dominant and Grassmannian permutations are vexillary ones.

We recall that $s_1, \ldots, s_{n-1}$ generate the symmetric group $\mathfrak{S}_n$ *i.e.* each permutation $\mu \in \mathfrak{S}_n$ can be expressed as a product of elementary (simple) transpositions. The *length* $l(\mu)$ of the permutation μ is the minimal number of simple transpositions required to express μ as such a product: this product is then called a *reduced decomposition* of μ. It is a classical result that the length of a permutation μ is also equal to the number of its inversions or equivalently to the sum of all components of its code.

Because of relations (2.81) and (2.82) the reduced decomposition of a permutation $\mu \in \mathfrak{S}_n$ is in general not unique, but thanks to relations (2.84) and (2.85) an operator:

$$\partial_\mu := \partial_{i_1} \ldots \partial_{i_k} \quad , \tag{2.95}$$

can be unambiguously defined for any reduced decomposition $s_{i_1} \ldots s_{i_k}$ of μ. Moreover, considering any sequence of positive integers $\mathbf{i} = (i_1, \ldots, i_p)$, the property $\partial_i^2 = 0, i = 1, \ldots, n-1$ shows that if $\mathbf{i}$ is not a reduced decomposition of a certain permutation, then $\partial_\mathbf{i} = \partial_{i_1} \ldots \partial_{i_p} = 0$.

2.12.2 Schubert polynomials

Schubert polynomials possess many interesting combinatorial properties related to the symmetric group. They contain as a subfamily Schur functions which are the fundamental symmetric polynomials. This section gives a brief exposition of some properties of Schubert polynomials that lead to efficient computations.

For each $\mu \in \mathfrak{S}_n$, the (simple) Schubert polynomial $\mathbf{X}_\mu$ is defined by:

$$\mathbf{X}_\mu := \partial_{\mu^{-1}\omega}(x^\delta) \quad ,$$

where ω denotes the longest—with respect to the number of inversions—element of $\mathfrak{S}_n$, μ^{-1} the inverse of μ and $x^\delta := x_1^{n-1} x_2^{n-2} \ldots x_n^0$. For instance, we compute $\mathbf{X}_\mu$ for $\mu = (1,4,2,3)$ as follows: we have $\omega = (4,3,2,1)$ and $\mu^{-1} = (1,3,4,2)$ so that $\mu^{-1}\omega = (2,4,3,1) = s_1 s_2 s_3 s_2$. Thus, by definition, $\mathbf{X}_\mu = \partial_{2,4,3,1}(x^\delta) = \partial_1\partial_2\partial_3\partial_2(x_1^3 x_2^2 x_3)$. Now, remember that, thanks to relation (2.85), $\partial_2\partial_3\partial_2 = \partial_3\partial_2\partial_3$:

$$\begin{aligned}
\mathbf{X}_{1,4,2,3} &= \partial_1\partial_3\partial_2\partial_3(x_1^3 x_2^2 x_3) \\
&= \partial_1\partial_3\partial_2(x_1^3 x_2^2) \\
&= \partial_1\partial_3(x_1^3 x_2 + x_1^3 x_3) \\
&= \partial_1(x_1^3) = x_1^2 + x_1 x_2 + x_2^2 \quad .
\end{aligned}$$

Some basic properties of Schubert polynomials are the following:

1. $\forall \mu \in \mathfrak{S}_n$, $\mathbf{X}_\mu$ is a homogeneous polynomial of degree $l(\mu)$ in the variables $x_1, \ldots, x_{n-1}$.
2. $\forall \mu \in \mathfrak{S}_n$, $\mathbf{X}_\mu$ is symmetric in x_i, x_{i+1} if and only if $\mu(i) < \mu(i+1)$.
3. $\mathbf{X}_\omega = x^\delta$ and $\mathbf{X}_{1,\ldots,n} = 1$.
4. Whenever μ belongs to a *Young subgroup* $\mathfrak{S}_i \times \mathfrak{S}_j$ *i.e.* $\mu = \mu'\mu''$, $\mu' \in \mathfrak{S}_i \times \{id\}$, $\mu'' \in \{id\} \times \mathfrak{S}_j$, we have $\mathbf{X}_\mu = \mathbf{X}_{\mu'}\mathbf{X}_{\mu''}$.

Schubert polynomials are sometimes indexed by sequences $\mathbf{i} = (i_1, \ldots, i_n) \subset \mathbb{N}^n$ by taking the code of μ instead of the permutation itself. We denote by $\mathbf{Y}_\mathbf{i}$ the Schubert polynomial corresponding to the sequence $\mathbf{i} \subset \mathbb{N}^n$. On account of the above properties, it is quite easy to deduce that $\mathbf{Y}_\mathbf{i}$, $\mathbf{i} \subset \mathbb{N}^n$ is homogeneous of degree $|\mathbf{i}| = i_1 + \cdots + i_n$. Calculations may be simplified by the following properties:

1. Let μ be a dominant permutation, then $\mathbf{X}_\mu = x^{\lambda(\mu)}$.
2. If μ is a Grassmannian permutation, then $\mathbf{X}_\mu = s_{\lambda(\mu)}(x_1, \ldots, x_r)$ where r is the unique descent of μ and $s_{\lambda(\mu)}$ is the Schur function corresponding to the partition $\lambda(\mu)$.
3. $\mathbf{Y}_{\underbrace{0\ldots0i}_{r}} = s_i(x_1, \ldots, x_r)$, $i > 0$.
4. $\mathbf{Y}_{\underbrace{0\ldots0}_{r-k}\,\underbrace{1\ldots1}_{k}} = s_{\underbrace{1\ldots1}_{k}}(x_1, \ldots, x_r)$.
5. In particular, $\mathbf{X}_{s_i} = x_1 + \cdots + x_i$, for $i = 1, \ldots, n-1$.

The function `TableX` in the `SP` package computes all Schubert polynomials corresponding to a given symmetric group. The following *Maple* example illustrates such a computation for $\mathfrak{S}_3$, the table being indexed by the 6 permutations of the group $\mathfrak{S}_3$:

```
ACE> TableX(3);
                                table([
                                  [1, 2, 3] = 1
                                  [2, 1, 3] = x1
                                  [1, 3, 2] = x1 + x2
                                  [2, 3, 1] = x1 x2
                                                2
                                  [3, 1, 2] = x1
                                                2
                                  [3, 2, 1] = x1  x2
                                ])
```

Schubert polynomials $\mathbf{X}_\mu$, $\mu \in \mathfrak{S}_n$ form a $\mathbb{Z}-$basis of the subspace of $\mathbb{Z}[x_1, \ldots, x_n]$ spanned by the monomials $x^\alpha, \alpha \subseteq (n-1, n-2, \ldots, 0)$, consequently Schubert polynomials $\mathbf{X}_\mu$, $\mu \in \mathfrak{S}_\infty$ form a $\mathbb{Z}-$basis of the polynomial ring $\mathbb{Z}[x_1, x_2, \ldots]$ in an infinite sequence of indeterminates (the function `ToX` expresses any general expression in the basis of Schubert polynomials). The function `x2X` converts any expression from the basis of monomials to the basis of Schubert polynomials while the function `Tox` realizes the reverse operation as illustrated by the following example:

```
ACE> r := x2X(x1^5 + x1*x2^3*x3);

      r := X[2, 5, 3, 1, 4] - X[3, 4, 2, 1] - X[4, 2, 3, 1] + X[6, 1, 2, 3, 4, 5]

ACE> Tox(r);
                                        5         3
                                      x1  + x1 x2  x3
```

Schubert polynomials $\mathbf{X}_\mu, \mu \in \mathfrak{S}_n$ also form a basis of the algebra of polynomials $\mathbb{Z}[x_1, \ldots, x_n]$ regarded as a free module over the ring of symmetric polynomials $\mathbf{Sym}[x_1, \ldots, x_n]$. This point exceeds the scope of this paper hence we refer the interested reader to [70] for more details and solely give an example of this interpretation of $\mathbb{Z}[x_1, \ldots, x_n]$. Considering the previous polynomial $p = x_1^5 + x_1 x_2^3 x_3$ for

which the expansion in terms of Schubert polynomials involves permutations in $\mathfrak{S}_4$, $\mathfrak{S}_5$ and $\mathfrak{S}_6$, we have another expression of p that only involves Schubert polynomials in $\mathfrak{S}_3$ but now with coefficients that are symmetric functions in the variables x_1, x_2, x_3 expressed for instance in the Schur basis: $p = (s_{3,1,1} - s_{2,2,1})\mathbf{X}_{1,2,3} - s_{3,1}\mathbf{X}_{2,1,3} - s_{1,1,1}\mathbf{X}_{2,3,1} + (s_3 - s_{1,1,1})\mathbf{X}_{3,1,2} + s_{2,1,1}\mathbf{X}_{1,3,2}$.

The multiplicative structure in $\mathbf{Sym}[x_1, \ldots, x_n]$ is determined by *Pieri's formula* that expresses in the basis of Schur functions the product $h_i s_\lambda$ (respectively $e_i s_\lambda$), $i \geq 0$, λ being a partition, and where h_i (resp. e_i) denotes the i-th complete (resp. elementary) symmetric function. In the Schubert basis, the multiplicative structure relies on the multiplication by one variable. It involves a certain combinatorics on permutations described by *Monk's formula* [80], [79, formula (4.15')]:

$$x_r \mathbf{X}_\mu = \sum_{\nu = \mu\, t_{ri},\, i>r,\, l(\nu)=l(\mu)+1} \mathbf{X}_\nu - \sum_{\eta = \mu\, t_{ir},\, i<r,\, l(\eta)=l(\mu)+1} \mathbf{X}_\eta \quad , \tag{2.96}$$

where t_{ij} denotes the transposition that interchanges i and j. Because $\mathbf{X}_{s_r} = x_1 + \cdots + x_r$, it can be equivalently stated by:

$$\mathbf{X}_{s_r} \mathbf{X}_\mu = \sum_{\nu = \mu\, t_{ij},\, i \leq r < j,\, l(\nu)=l(\mu)+1} \mathbf{X}_\nu \quad .$$

For instance, to compute $x_4 \mathbf{X}_{3,6,2,4,8,7,1,5}$:

```
ACE> ToX(x4*X[3,6,2,4,8,7,1,5]);

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

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

A more general Pieri formula [77] provides a way to express the product $h_i \mathbf{X}_\mu$ (resp. $e_i \mathbf{X}_\mu$) in the Schubert basis. For the time being, Monk's formula appears as the fundamental tool for calculation on Schubert polynomials. *Exempli gratia*, we may choose r so that there is only one positive term in the product $x_r \mathbf{X}_\mu$. In that case, we obtain the following so-called *transition*:

$$\mathbf{X}_\nu = x_r \mathbf{X}_\mu + \sum_\eta \mathbf{X}_\eta \quad . \tag{2.97}$$

By iterating this same principle on $\mathbf{X}_\mu$ and all $\mathbf{X}_\eta$, it gives an efficient way to develop a Schubert polynomial $\mathbf{X}_\nu$ in the basis of monomials (this algorithm is used by the function `X2x`). Now, starting from an identity of type (2.97), one writes:

$$\mathbf{X}_\nu \longrightarrow \sum_\eta \mathbf{X}_\eta \quad , \tag{2.98}$$

and calls it a *truncated transition*. This amounts to ignore the extra term $x_r \mathbf{X}_\mu$ (this can be of course realized by putting $x_r = 0$). To define the *maximal transition* of ν, we start from equation (2.97) in which one chooses r to be the last descent of the permutation—this ensures the validity of the equation (2.98) except for the identity permutation.

To any permutation ν, one can associate a tree defined as follows: if ν is vexillary, the tree is reduced to one leaf indexed by ν; otherwise, considering the truncated maximal transition $\mathbf{X}_\nu \longrightarrow \sum \mathbf{X}_\eta$, one builds all edges from ν to all sub-trees corresponding to each η.

The action $\longrightarrow$ that corresponds in fact to an alphabet restriction, allows to compute the symmetric part of a Schubert polynomial as a sum of Schur functions (which are Schubert polynomials

corresponding to Grassmannian permutations). A special case of the algorithm [74] gives the decomposition of a product of two Schur functions.

In relation with the geometry of flag manifolds, it is natural to define a scalar product on $\mathbb{Z}[x_1, \ldots, x_n]$ for which elements of the ring of symmetric polynomials $\mathbf{Sym}[x_1, \ldots, x_n]$ are scalars. The use of the ∂_ω, $(\omega \in \mathfrak{S}_n)$ operator seems to be the best adapted so that the scalar product may be defined by:

$$\langle f, g \rangle := \partial_\omega(fg) \quad . \tag{2.99}$$

For this scalar product, divided differences ∂_i, $i = 1, \ldots, n-1$ are self-adjoint and more generally we get the next property on Schubert polynomials: let $\mu, \nu \in \mathfrak{S}_n$ such that $l(\mu) + l(\nu) = \binom{n}{2}$, then:

$$\langle \mathbf{X}_\mu, \mathbf{X}_\nu \rangle = \begin{cases} 1 & \text{if } \nu = \omega\mu \quad , \\ \\ 0 & \text{otherwise} \quad . \end{cases} \tag{2.100}$$

Thus, Schubert polynomials of degree k are adjoint to Schubert polynomials of degree $\binom{n}{2} - k$. The following *Maple* instructions show how to compute all scalar products between Schubert polynomials in $\mathfrak{S}_3$ using the function `ScalarPol` that efficiently implements definition (2.99), expressing the result in terms of Schur functions:

```
ACE> Sn:=3:
ACE> scal:=array(1..Sn!, 1..Sn!): l:=ListPerm(Sn):
ACE> for i from 1 to Sn! do
      for j from i to Sn! do
        scal[i, j] := SpScalarPol(Tox(X[op(op(i, 1))]), Tox(X[op(op(j, 1))]), Sn);
        scal[j, i] := scal[i, j]
      od
    od:
ACE> print(l);

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

```
ACE> print(scal);

                    [ 0    0     0     0      0        1    ]
                    [                                      ]
                    [ 0    0     1     0      0       s[1]  ]
                    [                                      ]
                    [ 0    1     0     0     s[1]     s[2]  ]
                    [                                      ]
                    [ 0    0     0     0      1       s[1]  ]
                    [                                      ]
                    [ 0    0   s[1]    1      0     s[1, 1] ]
                    [                                      ]
                    [ 1  s[1]  s[2]  s[1]  s[1, 1]  s[2, 1] ]
```

The next section is devoted to the study of double Schubert polynomials and our point of view will

be to shed some new light on simple Schubert polynomials.

2.12.3 Double Schubert polynomials

Let us take two independent sets of indeterminates $\mathbf{x} := \{x_1, \ldots, x_n\}$ and $\mathbf{y} := \{y_1, \ldots, y_n\}$. The *maximal double Schubert polynomial* $\mathbb{X}_\omega$ is defined by:

$$\mathbb{X}_\omega(\mathbf{x}, \mathbf{y}) := \prod_{i+j \leq n} (x_i - y_j) \quad , \tag{2.101}$$

where ω denotes the longest element of $\mathfrak{S}_n$, and in general for any permutation $\mu \in \mathfrak{S}_n$ the *double Schubert polynomial* $\mathbb{X}_\mu$ is defined to be the image under divided differences of $\mathbb{X}_\omega(\mathbf{x}, \mathbf{y})$:

$$\mathbb{X}_\mu(\mathbf{x}, \mathbf{y}) := \partial_{\mu^{-1}\omega}(\mathbb{X}_\omega(\mathbf{x}, \mathbf{y})) \quad , \tag{2.102}$$

in which divided differences act only on the $\mathbf{x}$ variables. These polynomials which belong to the polynomial ring $\mathbb{Z}[x_1, \ldots, x_{n-1}, y_1, \ldots, y_{n-1}]$ have the following properties:

1. $\forall \mu \in \mathfrak{S}_n$, $\mathbb{X}_\mu$ is homogeneous of degree $l(\mu)$ in the variables $x_1, \ldots, x_{n-1}, y_1, \ldots, y_{n-1}$.
2. Specializing all y_i's to 0, $\mathbb{X}_\mu(\mathbf{x}, \mathbf{y}) = \mathbf{X}_\mu(\mathbf{x})$ because $\mathbb{X}_\omega(\mathbf{x}, 0) = x^\delta$.
3. $\mathbb{X}_{1,\ldots,n}(\mathbf{x}, \mathbf{y}) = 1$.
4. $\forall \mu \in \mathfrak{S}_n$, $\mathbb{X}_{\mu^{-1}}(\mathbf{x}, \mathbf{y}) = \mathbb{X}_\mu(-\mathbf{y}, -\mathbf{x})$.

The `SP` package provides the function `TableXX` to compute the table of all double Schubert polynomials corresponding to $\mathfrak{S}_n$. The example shown below illustrates the use of this function for $n = 3$:

```
ACE> TableXX(3);
                  table([
                      [1, 2, 3] = 1
                      [2, 1, 3] = x1 - y1
                      [1, 3, 2] = x1 - y1 + x2  - y2
                      [2, 3, 1] = (x1 - y1) (x2 - y1)
                      [3, 1, 2] = (x1 - y1) (x1 - y2)
                      [3, 2, 1] = (x1 - y1) (x1 - y2) (x2 - y1)
                  ])
```

Double Schubert polynomials generalize the coefficients $(x_1 - y_1) \ldots (x_1 - y_i)$ in the Newton interpolation formula for functions of one variable (2.94) to the case of any number of variables. In other words, considering one set of indeterminates $\mathbf{x} := \{x_1, x_2, \ldots\}$ and $\mathbf{y} := \{y_1, \ldots, y_{n-1}\}$ denoting any set of elements of a field $\mathbb{K}$, the Newton interpolation formula for $p \in \mathbb{K}[x_1, x_2, \ldots]$ becomes:

$$p(\mathbf{x}) \equiv \sum_\mu \partial_\mu(p)(\mathbf{y}) \mathbb{X}_\mu(\mathbf{x}, \mathbf{y}) \quad , \tag{2.103}$$

in which divided differences act on the $\mathbf{y}$ variables. This formula becomes an exact formula if one takes a sufficiently big symmetric group: it then coincides with the result of the `x2XX` function. The following *Maple* commands explicit the function `NewtonInterp` that realizes the Newton interpolation, the last argument being the degree of the symmetric group:

```
ACE> r := NewtonInterp(x1^2, 2);

                        2
                 r := y1  XX[1, 2] + (y1 + y2) XX[2, 1]
ACE> Tox(r);
                             y1 x1 + y2 x1 - y2 y1

ACE> Tox( NewtonInterp(x1^5 + x1*x2^3*x3, 6) );

                                  5        3
                                 x1  + x1 x2  x3
```

in which `Tox` develops any expression in the basis of monomials.

Because in definition (2.102) divided differences act on the $\mathbf{x}$ variables, double Schubert polynomials are sums of monomials x^α, $\alpha \subseteq \delta$. Thus, double Schubert polynomials can be expressed as linear combinations of simple Schubert polynomials in the variables $\mathbf{x}$ with coefficients in $\mathbb{Z}[\mathbf{y}]$. Double Schubert polynomials may be considered as generating functions for families of simple Schubert polynomials as illustrated by the relation:

$$\mathbb{X}_\mu(\mathbf{x},\, \mathbf{y}) = \sum_\nu \mathbf{X}_\nu(\mathbf{x})\, \mathbf{X}_{\nu\omega}(-\mathbf{y}) \quad , \tag{2.104}$$

in which μ and ν lie in $\mathfrak{S}_n$. As in the case of simple Schubert polynomials (2.96), we have a Monk formula to multiply a double Schubert polynomial by a variable x_r:

$$x_r \mathbb{X}_\mu \quad = \quad y_{\mu(r)} \mathbb{X}_\mu + \sum_{\nu = \mu\, t_{rj},\, j>r,\, l(\nu)=l(\mu)+1} \mathbb{X}_\nu - \sum_{\eta = \mu\, t_{jr},\, j<r,\, l(\eta)=l(\mu)+1} \mathbb{X}_\eta \quad . \tag{2.105}$$

Thus, working with two sets of variables involves only adding an extra term [70] in Monk's formula (2.96), as shown below:

```
ACE> ToXX(x3*XX[2,3,5,1,4]);

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

Furthermore, as in the case of simple Schubert polynomials, r may be chosen such that:

$$\mathbb{X}_\nu = (x_r - y_{\mu(r)})\mathbb{X}_\mu + \sum_\eta \mathbb{X}_\eta \quad . \tag{2.106}$$

This provides a convenient way to compute the development of a double Schubert polynomial in the basis of monomials but the most interesting point to notice is that it gives an efficient manner to specialize double Schubert polynomials. Indeed, in relation (2.106) the term $(x_r - y_{\mu(r)})\mathbb{X}_\mu$ will be very soon specialized to zero whenever the specialization of the second alphabet to a permutation of the first one maps the factor $x_r - y_{\mu(r)}$ onto zero. The `TableXX` function also gives the specialization of all double Schubert polynomials by adding a second argument which is a permutation:

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

             table([
               [2, 4, 3, 1] = 0
               [2, 1, 3, 4] = x1 - x2
               [2, 3, 4, 1] = 0
               [4, 2, 1, 3] = 0
               [2, 3, 1, 4] = 0
               [3, 4, 1, 2] = 0
               [3, 1, 4, 2] = (x3 - x4) (- x4 + x1) (x1 - x2)
               [1, 2, 3, 4] = 1
               [3, 4, 2, 1] = 0
               [4, 2, 3, 1] = 0
               [3, 2, 4, 1] = 0
               [3, 2, 1, 4] = 0
               [2, 4, 1, 3] = 0
               [4, 1, 2, 3] = 0
               [1, 3, 2, 4] = - x4 + x1
               [1, 4, 2, 3] = (x2 - x1) (- x4 + x1) + (- x4 + x1) (x1 - x2)
               [4, 3, 2, 1] = 0
               [1, 2, 4, 3] = x3 - x4
               [2, 1, 4, 3] = (x3 - x1) (x1 - x2) + (- x4 + x1) (x1 - x2)
               [4, 1, 3, 2] = 0
               [3, 1, 2, 4] = (- x4 + x1) (x1 - x2)
               [1, 4, 3, 2] = (x3 - x4) ((x2 - x1) (- x4 + x1) + (- x4 + x1) (x1 - x2))
               [4, 3, 1, 2] = 0
               [1, 3, 4, 2] = (x3 - x4) (- x4 + x1)
             ])
```

More generally, the specialization of the second alphabet (`SpSpecialize`) is related to many interesting properties, which justify using double Schubert polynomials instead of their specialization $y_i = 0$.

2.12.4 Examples from algebraic geometry

Chern classes of flag manifolds

According to Borel, the cohomology ring of the flag manifold $\mathcal{F}_n$ is isomorphic to $\mathbb{Z}[x_1, \ldots, x_n]/I^+$ where I^+ is the ideal generated by the symmetric polynomials in the variables $x_1, \ldots, x_n$ without constant term.

The flag manifold is equipped with tautological line bundles $\mathrm{L}_1, \ldots, \mathrm{L}_n$, 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^+. It follows from the splitting principle (see [68], §13) that the tangent bundle has the same Chern classes as the direct sum $\bigoplus_{i>j} \mathrm{L}_i \mathrm{L}_j^*$, that is:

$$c(\mathcal{F}_n) = \prod_{i<j} (1 + \gamma_i - \gamma_j) \quad .$$

The following sequence of instructions gives the expansion of $c(\mathcal{F}_4)$ in the basis of Schubert cycles:

```
ACE> chern:=proc(n) local i, j, res;
       res:=1;
       for j to n do
         for i to j-1 do
           res:=res * (1+x.i-x.j)
         od
       od;
       RETURN(res)
     end:
ACE> Flag(4):
ACE> r := chern(4);

               r := (1 + x1 - x2) (1 + x1 - x3) (1 + x2 - x3)

                    (1 + x1 - x4) (1 + x2 - x4) (1 + x3 - x4)

ACE> ToX(r);

      6 X[2, 3, 1, 4] +  6 X[3, 1, 2, 4]  + 4 X[2, 1, 4, 3] +  6 X[1, 4, 2, 3]

      + 6 X[3, 2, 1, 4] + 16 X[3, 1, 4, 2] + 2 X[1, 3, 2, 4] + 2 X[1, 2, 4, 3]

      + X[1, 2, 3, 4] + 6 X[1, 3, 4, 2]  +  2 X[2, 1, 3, 4] + 22 X[4, 1, 2, 3]

      + 24 X[4, 3, 2, 1] + 22 X[2, 3, 4, 1] + 20 X[2, 4, 1, 3] + 28 X[3, 2, 4, 1]

      + 44 X[3, 4, 1, 2] + 24 X[4, 2, 1, 3] + 24 X[2, 4, 3, 1] + 28 X[4, 1, 3, 2]

      +  6 X[1, 4, 3, 2] + 48 X[4, 2, 3, 1] + 36 X[4, 3, 1, 2] + 36 X[3, 4, 2, 1]
```

Moreover, Schubert polynomials are compatible with the embeddings $H^*(\mathcal{F}_{i_1\cdots i_r}) \hookrightarrow H^*(\mathcal{F}_n)$ of the cohomology rings of incomplete flag varieties given by the fibrations:

$$(V_0 \subset V_1 \subset \cdots \subset V_n) \longrightarrow (V_{i_1} \subset V_{i_1+i_2} \subset \cdots \subset V_{i_1+\cdots+i_r}) \quad .$$

For example, the Chern classes of the Grassmannian $\mathrm{G}(2,4)$ are given by:

```
ACE> Flag(4):
ACE> res:=1:
ACE> for i from 1 to 2 do for j from 3 to 4 do res:=res * (1+x.i-x.j) od od:
ACE> r := ToX(res);

          r := 7 X[2, 3, 1, 4] + 7 X[1, 4, 2, 3]  + 4 X[1, 3, 2, 4]

               + X[1, 2, 3, 4] + 12 X[2, 4, 1, 3] + 6 X[3, 4, 1, 2]
```

To obtain the result in the usual basis of Schur functions (indexation of Schubert cycles of $\mathrm{G}(k,n)$ by partitions contained in the rectangle $k \times (n-k)$), we can use the function `ToY`:

```
ACE> ToY(r);

                7 Y[1, 1, 0, 0] + 7 Y[0, 2, 0, 0]  + 4 Y[0, 1, 0, 0]

                + Y[0, 0, 0, 0] + 12 Y[1, 2, 0, 0] + 6 Y[2, 2, 0, 0]
```

When the code of a permutation has the form $I = J.0^{n-1}$, where J is a partition (weakly increasing vector), the Schubert polynomial $\mathbf{Y}_I$ corresponds to the Schubert cycle s_J of $\mathrm{G}(k, n-k)$. The properties of the Chern classes of flag manifolds are discussed in [76], [73].

Projective degrees of Schubert varieties

Let $i : \mathcal{F}_n \hookrightarrow \mathbb{P}^N$ be a projective embedding of $\mathcal{F}_n$, corresponding to the line bundle:

$$\eta = \mathrm{L}_1^{m_1}\mathrm{L}_2^{m_2}\ldots\mathrm{L}_n^{m_n} = i^*\mathcal{O}_{\mathbb{P}^N}(1) \quad ,$$

and let $h \in \mathrm{H}^1(\mathbb{P}^N)$ be the class of a hyperplane. The projective degree of a subvariety Z of codimension k in $\mathcal{F}_n$ is:

$$d(\mathrm{Z}) = \varepsilon\left(i^*([i\mathrm{Z}].h^k)\right) = \varepsilon([\mathrm{Z}].Y^k) \quad ,$$

where $\varepsilon : \mathrm{H}^{top}(\mathcal{F}_n) \longrightarrow \mathbb{Z}$ is the morphism sending to 1 the class of a point, and $Y = i^*h = \sum m_i x_i$.

For example, with the Plücker embedding we get $Y = (n-1)x_1 + \cdots + x_{n-1}$ and for this embedding, the degree of the Schubert variety $\underline{\mathbf{X}}_{2,1,4,3}$ of $\mathcal{F}_4$ is computed by the following sequence of commands:

```
ACE> z:=3*x1 + 2*x2 + x3:
ACE> ToX(z^4 * X[2,1,4,3]);

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

so that the required degree is 78. For more details about how to effectively compute the projective degrees of Schubert cycles, the reader may refer to [75].

The double Schubert polynomials also have a geometrical interpretation [67]. When the variables x_i and y_j are interpreted as Chern classes of certain line bundles over a variety M, they give the cohomology classes corresponding to various degeneracy loci of maps of flagged bundles over M.

2.13 Computations in the ring of polynomials in several variables

Computations in the ring $\mathbb{Z}[x_1, \ldots, x_n]$ of polynomials in several variables are not so easy because dimensions grow very quickly with the number of variables and with the degree. The problem is to find linear bases adapted to some specific problems without losing track of the multiplicative structure. For $n = 1$, such bases are provided by interpolation theory. In higher dimensions, a polynomial can have some symmetry, which cannot be efficiently exploited if the polynomial is just written as a sum of monomials.

For example, if a polynomial is totally symmetrical, we can have recourse to many combinatorial objects which render computations independent of the number of variables. To be able to use symmetry methods in the case of general polynomials, we shall look at the ring of polynomials in n variables

$x_1, \ldots, x_n$ as a module over the ring $\mathbf{Sym}[x_1, \ldots, x_n]$ of symmetric polynomials. It is easy to show that the set of monomials of degree less than $(n-1, \ldots, 2, 1, 0)$ (componentwise) is a linear basis of this module, which is free of rank $n!$.

We shall use a more interesting basis for this free module over $\mathbf{Sym}[x_1, \ldots, x_n]$, consisting of Schubert polynomials in the x_i and y_i (where the y_i are a second set of indeterminates). These Schubert polynomials are the universal coefficients in the generalized Newton interpolation formula (they generalize Newton's interpolation polynomials $(x_1 - y_1) \ldots (x_1 - y_n)$). Moreover they are the natural basis in the geometric interpretation of the ring of polynomials as the equivariant cohomology ring of the flag manifold. Last, but not least, their combinatorics extends the one of Schur functions (the natural basis of the ring $\mathbf{Sym}[x_1, \ldots, x_n]$), with constructions on permutations replacing the handling of partitions and Young tableaux.

2.13.1 Schubert bases

Let w be a permutation in $\mathfrak{S}_n$ and $c(w) = (c_1, c_2, \ldots, c_n)$ its code, then:

$$\mathbf{X}_w = x_1^{c_1} x_2^{c_2} \ldots x_n^{c_n} + \sum \alpha_i \; x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} \; , \tag{2.107}$$

where $\alpha_i > 0$, $i_1 \leq n-1, i_2 \leq n-2, \ldots, i_n \leq 0$, and $(i_1, i_2, \ldots, i_n)$ lexicographically smaller than $(c_1, c_2, \ldots, c_n)$.

Schubert polynomials are compatible with the embedding $\mathfrak{S}_n \hookrightarrow \mathfrak{S}_{n+1}$. Thus, $\{\mathbf{X}_w, w \in \mathfrak{S}_\infty\}$ is a $\mathbb{Z}$-basis of $\mathbb{Z}[x_1, x_2, \ldots]$. More precisely, let $\eta : \mathbb{Z}[x_1, x_2, \ldots] \to \mathbb{Z}$ be the constant term homomorphism. Then [79, (4.14)], one has:

$$\forall p \in \mathbb{Z}[x_1, x_2, \ldots], \qquad p = \sum_{w \in \mathfrak{S}_\infty} \eta(\partial_w p) \, \mathbf{X}_w \; . \tag{2.108}$$

Example 2.13.1 Let $p := x_1^5 + x_1 x_2^3 x_3$. Then p is equal to:

$$\begin{aligned} p &= \mathbf{X}_{6,1,2,3,4,5} - \mathbf{X}_{3,4,2,1} - \mathbf{X}_{4,2,3,1} + \mathbf{X}_{2,5,3,1,4} \\ &= x_1^5 - x_1^2 x_2^2 x_3 - x_1^3 x_2 x_3 + (x_1 x_2^3 x_3 + x_1^2 x_2^2 x_3 + x_1^3 x_2 x_3) \; . \end{aligned}$$

It should be noted that this expression of p involves simple Schubert polynomials indexed by permutations in $\mathfrak{S}_4, \mathfrak{S}_5$ and $\mathfrak{S}_6$ while the original polynomial only depends upon the three variables x_1, x_2 and x_3. It is possible to give an expression of p involving only simple Schubert polynomials in $\mathfrak{S}_3$, but the coefficients are now symmetric functions in x_1, x_2, x_3:

$$\begin{aligned} p = {} & \left(x_1 x_2 x_3^2 + x_1 x_2^2 x_3 + x_1^2 x_2 x_3 - x_1^2 x_2^2 x_3 - x_1^2 x_2 x_3^2 - x_1 x_2^2 x_3^2\right) \mathbf{X}_{1,2,3} - \Big(x_1^3 x_2 \\ & + x_1^3 x_3 + x_1^2 x_2^2 + 2x_1^2 x_2 x_3 + x_1^2 x_3^2 + x_1 x_2^3 + 2x_1 x_2^2 x_3 + 2x_1 x_2 x_3^2 + x_1 x_3^3 + x_2^3 x_3 \\ & + x_2^2 x_3^2 + x_2 x_3^3\Big) \mathbf{X}_{2,1,3} + \Big(x_1^3 + x_1^2 x_2 + x_1^2 x_3 + x_1 x_2^2 + x_1 x_3^2 + x_2^3 + x_2^2 x_3 + x_2 x_3^2 \\ & + x_3^3\Big) \mathbf{X}_{3,1,2} - \Big(x_1 x_2 x_3\Big) \mathbf{X}_{2,3,1} + \Big(x_1^2 x_2 x_3 + x_1 x_2^2 x_3 + x_1 x_2 x_3^2\Big) \mathbf{X}_{1,3,2} \; . \end{aligned}$$

For now on, we fix the integer n and use only Schubert polynomials $\mathbf{X}_w$ or $\mathbb{X}_w, w \in \mathfrak{S}_n$. Let $\mathbf{Sym}[\mathbf{x}] := \mathbf{Sym}[x_1, \ldots, x_n]$ be the subring of symmetric polynomials in $x_1, \ldots, x_n$. In fact, one has [79, Chap. V] the following proposition:

Proposition 2.13.1 *The space* $\mathbb{Z}[\mathbf{x}]$ *is a free module of dimension* $n!$ *over the ring* $\mathbf{Sym}[\mathbf{x}]$ *with basis the simple Schubert polynomials* $\mathbf{X}_w, w \in \mathfrak{S}_n$*:*

$$\mathbb{Z}[x_1, \ldots, x_n] \quad = \quad \bigoplus_{w \in \mathfrak{S}_n} \mathbf{Sym}[x_1, \ldots, x_n]\ \mathbf{X}_w\ . \tag{2.109}$$

In other words, any polynomial $p \in \mathbb{Z}[\mathbf{x}]$ may be uniquely expressed as a sum:

$$p = \sum_{w \in \mathfrak{S}_n} c_w\, \mathbf{X}_w\ ,$$

where the c_w's belong to $\mathbf{Sym}[\mathbf{x}]$, as illustrated by the previous expansion of $x_1^5 + x_1 x_2^3 x_3$.

2.13.2 Multiplicative structure

We study the ring $\mathbb{Z}[x_1, \ldots, x_n]$ of polynomials seen as a free module of dimension $n!$ over the ring $\mathbf{Sym}[x_1, \ldots, x_n]$ of symmetric polynomials, with basis the simple Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_n$. For this purpose, we have to recover the multiplicative structure on this module. As explained previously, Monk's formula gives the multiplicative structure of the ring in an infinite number of variables, and does not preserve the subspace with basis the $\mathbf{X}_w, w \in \mathfrak{S}_n$. This formula does not use the fact that coefficients can belong to $\mathbf{Sym}[x_1, \ldots, x_n]$. As there is at most one term (in Monk's formula) indexed by a permutation μ outside of $\mathfrak{S}_n$, we are thus led to express certain simple Schubert polynomials as a sum with coefficients in $\mathbf{Sym}[x_1, \ldots, x_n]$ of simple Schubert polynomials indexed by permutations of $\mathfrak{S}_n$. More generally, every simple Schubert polynomial in $\mathbb{Z}[x_1, \ldots, x_n]$ has an expression in the Schubert basis $\mathbf{X}_w, w \in \mathfrak{S}_n$.

We need only consider the w' of the type $(n+1, n, \ldots, \widehat{m}, \ldots, 2, 1, m)$ because using the divided differences $\partial_1, \ldots, \partial_{n-1}$, we get all others. We consider operators from $\mathfrak{S}_{n-1}$ into $\mathfrak{S}_n$, $P_0^n, \ldots, P_k^n$ defined by:

$$\begin{aligned}
P_0^n(\mu) &\longrightarrow n\ \mu \\
P_1^n(\mu) &\longrightarrow \mu_1\ n\ \mu_2 \ldots \\
\vdots \quad &\longrightarrow \quad \vdots \\
P_k^n(\mu) &\longrightarrow \mu_1 \ldots \mu_{k-1}\ n\ \mu_k \ldots
\end{aligned}$$

Proposition 2.13.2 *Let* n, m *be two integers such that* $m \leq n$*. Let* $\mathcal{R}(n, m)$ *be the polynomial recursively defined by:*

$$\mathcal{R}(n,m) := \begin{cases} e_1 \mathbf{X}_{n,n-1,\ldots,2,1} - \displaystyle\sum_{k=1}^{n-1} P_{n-k}^n \Big(\mathcal{R}(n-1,k)\Big)_{e_j \leftarrow e_{j+1}}\ , & \text{if } m = n, \\ \\ P_0^n \Big(\mathcal{R}(n-1,m)\Big)_{e_j \leftarrow e_{j+1}} - P_1^n \Big(\mathcal{R}(n-1,m)\Big)_{e_j \leftarrow e_{j+2}}\ , & \text{otherwise.} \end{cases}$$

Then, $\mathcal{R}(n, m)$ *is exactly equal to the Schubert polynomial which is indexed by the permutation* $(n+1, n, \ldots, \widehat{m}, \ldots, 2, 1, m)$*.*

Using the divided differences $\partial_1, \ldots, \partial_{n-1}$, which commute with the e_i, one can deduce from proposition 2.13.2 the expression of any Schubert polynomial in $\mathfrak{S}_{n+1}$.

2.13.3 Applications

The FM package provides functions to compute with this structure of free module over symmetric polynomials. Proposition 2.13.2 has modified Monk's formula (2.96) into a formula involving only permutations in $\mathfrak{S}_n$ and thus, allows to express the product of any Schubert polynomial $\mathbf{X}_w, w \in \mathfrak{S}_n$ by any polynomial in $x_1, \ldots, x_n$. The FM_n function is used to fix the number of variables:

```
ACE> ToX(x3*X[3,1,4,2]);

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

ACE> FM_n(4);

                                          4

ACE> ToXfix(x3*X[3,1,4,2]);

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

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

More generally, it allows to multiply any linear combination of Schubert polynomials with coefficients in $\mathbf{Sym}[x_1, \ldots, x_n]$ by a polynomial with coefficients also in $\mathbf{Sym}[x_1, \ldots, x_n]$. To reduce the symmetric coefficients, one uses distinguished bases of symmetric functions and corresponding algorithms. These algorithms are furnished by combinatorial properties of partitions or Young tableaux and are independent of the number of variables. The ToXfix function can be used to convert any expression in the basis of Schubert polynomials indexed by permutations of $\mathfrak{S}_n$:

```
ACE> FM_n(4):
ACE> ToXfix(x3^3*X[2,4,1,3]^2, 'collect');

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

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

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

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

We also provide the ToXfixScal function which gives the same result as ToXfix but uses scalar products rather than the modified Monk formula.

As an application of the methods described in the preceeding section, we shall mention the cohomology ring of a relative *Grassmann* variety. More explicitely, let A and B be two disjoint alphabets, $A := \{x_1, \ldots, x_m\}$ and $B := \{x_{m+1}, \ldots, x_n\}$ of respective cardinal m and $n - m$. We shall denote by $A + B$ the alphabet $\{x_1, \ldots, x_m, x_{m+1}, \ldots, x_n\}$.

The above mentioned ring is the space of polynomials in $x_1, \ldots, x_n$ which are symmetrical separately in $x_1, \ldots, x_m$ and $x_{m+1}, \ldots, x_n$. This space is a free module over the ring of symmetric polynomials $\mathbf{Sym}[x_1, \ldots, x_n]$. It has a canonical basis (the *Schubert cycles*) of cardinal $\binom{n}{m}$, consisting of

Schur functions [88] indexed by partitions smaller than $(n-m)^m$. For example, in the case $m=1$, the basis consists in $x_1^0, x_1^1, \ldots, x_1^{n-1}$ and the decomposition of a polynomial into this basis is given by *Euclid*'s algorithm. For a more general m, the product of two Schur functions belonging to the basis, given by the *Littlewood-Richardson* rule [88], will involve partitions not smaller than $(n-m)^m$.

However, one can express this product of Schur functions, thanks to the preceding section, in terms of Schubert polynomials $\mathbf{X}_w, w \in \mathfrak{S}_n$. These Schubert polynomials must be symmetrical in $x_1, \ldots, x_m$ and therefore are Schur functions [79]. Because the indexing permutations are in $\mathfrak{S}_n$, then the corresponding Schur functions are indexed by partitions smaller or equal to $(n-m)^m$, *i.e.* belong to the canonical basis. In other words, the previous section gives a method to decompose any element of the cohomology ring in the basis of Schubert cycles. For instance, for $m=3$, $n=5$, the square of the Schur function $s_{21}(x_1,x_2,x_3)$ is equal to $e_4 s_2 - e_3 s_{21} + e_1^2 s_{22} + (e_4 - e_3 e_1) s_{11} + (e_4 e_1 + e_5) s_1 + (e_1^2 - e_2) s_{211} - (e_2 e_1 + e_3) s_{111} + s_{222} + 2 e_1 s_{221} + e_5 e_1$:

```
ACE> FM_n(5);

                                        5

ACE> CLG_n(3);

                                        3

ACE> sf := Tox_n(s[2,1]);

                    2       2                       2       2    2          2
          sf := x3  x2 + x3  x1 + 2 x3 x2 x1 + x3 x2  + x3 x1  + x2  x1 + x2 x1

ACE> r := x2XfixScal(sf^2, 'collect');

        r := s[2, 1, 1, 1, 1] X[1, 2, 3, 4, 5]

           - (- 2 s[1, 1, 1, 1, 1] - s[2, 1, 1, 1]) X[1, 2, 4, 3, 5]

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

           - s[1, 1, 1] X[1, 3, 5, 2, 4] - (2 s[1, 1, 1] + s[2, 1]) X[2, 3, 4, 1, 5]

           + (s[2] + s[1, 1]) X[1, 4, 5, 2, 3] + s[2] X[2, 3, 5, 1, 4]

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

ACE> simplify( Xfix2x(r) );

                     2   3       3   3             4     2   4        2   2   2
           6 x3 x2  x1  + 2 x2  x1  + 2 x3 x2 x1  + x2  x1  + 10 x3  x2  x1

                 2      3     2   4     4   2       4            4            3   3
          + 6 x3  x2 x1  + x3  x1  + x2  x1  + 2 x2  x3 x1 + 2 x3  x2 x1 + 2 x3  x1
```

```
                3   3     4   2     4   2     4   2       3   2          3      2
          + 2 x3  x2  + x2  x3  + x3  x2  + x3  x1  + 6 x3  x2  x1 + 6 x3  x2 x1

                2   3             3   2
          + 6 x3  x2  x1 + 6 x3 x2  x1
```

while the Littlewood-Richardson rule gives $s_{42}+s_{411}+s_{33}+2s_{321}+s_{222}$ that involves partitions outside of (222):

```
ACE> CLG_n(3);

                                          3

ACE> Tos_n(s[2,1]^2);

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

Here is a bigger example of a polynomial involving only x_1, x_2, x_3 and x_4 in terms of Schubert polynomials in $\mathfrak{S}_4$:

```
ACE> x2XfixScal(x1^3*x2^3*x3^5 + x1^4*x3^4*x4^3, 'collect');

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

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

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

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

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

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

     - (- s[4, 2, 2] - 2 s[3, 2, 2, 1] - s[2, 2, 2, 2]) X[2, 3, 4, 1]

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

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

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

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

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

The package FOCK provides an environment for computing in the Fock representation of the quantum affine algebra $U_q(\widehat{sl}_n)$. It is a satellite package of ACE, that is it is not included in the ACE environment but this is an example of a package that uses the ACE system.

Following [41], we identify the (q-deformed) Fock space with the vector space $\mathcal{F}$ spanned by Young diagrams λ. We also find convenient to regard $\mathcal{F}$ as the space of symmetric functions, by identifying λ with the Schur function s_λ. The main routines compute:

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

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

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

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

One motivation for implementing these routines was their conjectured application to the representation theory of Hecke algebras at roots of unity [38]. Indeed, as explained in Section 2.14.7 below, they yield the first efficient general algorithm for computing their decomposition matrices (over a field of characteristic 0).

In [38], canonical bases of the whole Fock space $\mathcal{F}$ were introduced, extending the global crystal bases of $M(\Lambda_0)$. The main idea is to extend to $\mathcal{F}$ the canonical involution of $M(\Lambda_0)$. Routines computing this involution and the associated canonical bases are included in the package (see Section 2.14.6).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

0		
1		
2	0	
0	1	2

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

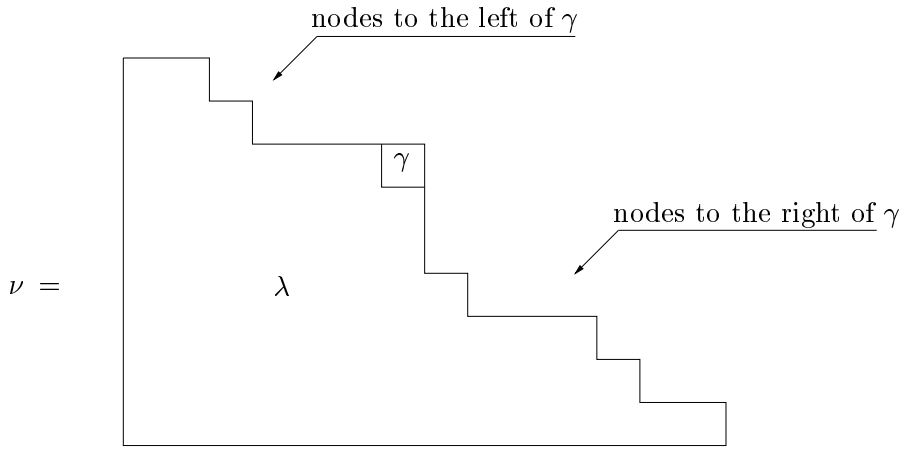
Let $i \in \{0, 1, \ldots, n-1\}$ and let λ, ν be two partitions such that ν is obtained from λ by filling an indent i-node γ. We set:

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

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

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

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



The algebra $U_q(\widehat{sl}_n)$ acts on $\mathcal{F}$ by [31] [41]:

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

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

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

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

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

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

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

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

                                      0

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

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

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

                                  s[4, 2, 1, 1]

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

                                  s[3, 3, 1, 1]

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

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

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

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

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

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

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

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

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

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

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

```
ACE> ListPart(6);

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

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

ACE> ListPart(6, 'regular'=2);

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

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

2.14.3 Crystal graph of the Fock space

This is an infinite graph with vertices the partitions $\lambda \in \mathcal{P}$ and whose edges record the action of Kashiwara's renormalized lowering operators $\tilde{f}_i$ on these partitions. To describe these edges we introduce the notion of *good node* [41, 37]. Fix a residue $r \in \{0, 1, \ldots, n-1\}$ and consider the sequence of removable and indent r-nodes obtained by reading the boundary of λ from left to right. Thus, for $\lambda = (3\,2\,1^2)$ and $n = 3$ one has:

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

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

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

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

For example, this algorithm applied to the sequence:

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

goes through the following steps:

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

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

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

and the penultimate R corresponds to a good node.

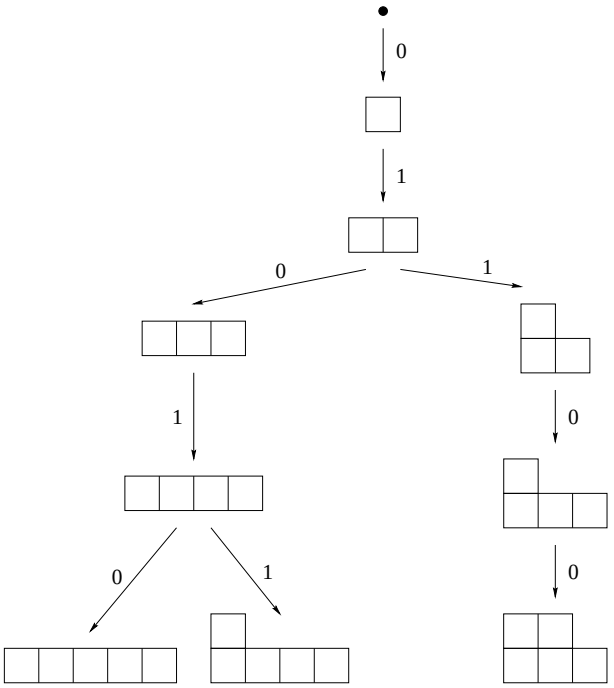
With this rule, we see that $\lambda = (3\,2\,1^2)$ has one good 2-node, no good 1-node, and one good 0-node (the leftmost removable 0-node).

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

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

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

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



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

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

                                    0

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

                                [3, 3, 1, 1]

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

                                [3, 2, 2, 1]

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

                                 [3, 2, 1]

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

                                    0

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

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

The set of vertices of the crystal graph of a $U_q(\mathfrak{g})$-module is endowed with a natural action of the Weyl group of the Lie algebra $\mathfrak{g}$ [35]. In the case $\mathfrak{g} = \widehat{sl}_n$, the Weyl group is the affine symmetric

group. The action of a simple reflection s_i of W on λ is defined by:

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

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

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

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

                              [3, 2, 1]

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

                             [3, 3, 1, 1]

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

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

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

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

where the indices i are understood modulo n. The corresponding involution on the set of partitions is called the Mullineux-Kleshchev involution because of its interpretation in the modular representation theory of the symmetric groups [37].

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

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

                                [5, 2]
```

2.14.4 Canonical basis of the basic representation

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

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

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

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

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

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

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

By [34], there exists a unique $\mathbb{Q}[q, q^{-1}]$-basis $\{G(\mu), \mu \in \mathcal{P}_n\}$ of $M(\Lambda_0)_{\mathbb{Q}}$ such that:

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

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

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

A simple and efficient algorithm to compute $\{G(\mu)\}$ has been proposed in [38]. This algorithm is directly inspired by the techniques described in [33] for computing the decomposition matrices of symmetric groups.

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

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

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

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

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

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

2.14.5 Kashiwara-Miwa-Stern algebra and ribbon tableaux

For $q = 1$, the operators:

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

commute with the action of $\widehat{sl}_n$ on the Fock space. The quantum analogues A_k, B_k of these operators have been found by Kashiwara-Miwa-Stern [36]. In [39], other generators U_k, V_k of the associative

algebra generated by the A_k, B_k are described in terms of ribbon tableaux. V_k is a q-analogue of multiplication by the plethysm $p_n[h_k]$, and U_k corresponds to $D_{p_n[h_k]}$. More generally, for a partition λ, one defines $V_\lambda = V_{\lambda_1} V_{\lambda_2} \dots V_{\lambda_r}$, and similarly U_λ. The vectors $v_\lambda = V_\lambda \emptyset$ form a basis of $U_q(\widehat{sl}_n)$-highest weight vectors in the Fock space, which can be conveniently computed by enumerating n-ribbon tableaux of weight λ and recording their spins. A recursive formula for the spin polynomials avoids the full enumeration.

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

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

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


ACE> r := BOp(s[2,1], 2, 2):
ACE> Tos(r, 'collect');

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

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

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

ACE> r := VOp(s[1], [1,1], 2):
ACE> Tos(r, 'collect');

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

2.14.6 Canonical basis of the Fock space representation

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

the involution of Section 2.14.4 [40].

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

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

as illustrated by the examples below:

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

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

ACE> InvolMat(2, 5);

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

Using this extended involution and imitating Section 2.14.4 one obtains a canonical basis $\{G(\lambda)\}$ of $\mathcal{F}$. The subset $\{G(\mu)\}$ labelled by n-regular partitions μ coincides with the canonical basis of $M(\Lambda_0)$.

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

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

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

ACE> CBMat(2, 6);

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

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

It was conjectured in [38] and proved independently by Ariki [29] and Grojnowski (building on results announced in [30]) that the matrices expressing the global lower crystal basis of $M(\Lambda_0)$ on the natural basis of the Fock space, are q-analogues of the decomposition matrices of the Hecke algebras $H_m(v)$, for v a primitive n-th root of unity.

Similarly, it was conjectured in [40] that the matrices expressing the canonical basis of $\mathcal{F}$ on the natural basis of the Fock space, are q-analogues of the decomposition matrices of the v-Schur algebras $\mathcal{S}_m(v)$ at $v = \sqrt[n]{1}$.

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

```
ACE> CBMat(2, 6);
                    [  1   0   0   0   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [  q   1   0   0   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [  0   q   1   0   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [      2                                ]
                    [  q   q   q   1   0  0   0  0  0  0  0 ]
                    [                                       ]
                    [  0   0   q   0   1  0   0  0  0  0  0 ]
                    [                                       ]
                    [  0   0   0   0   0  1   0  0  0  0  0 ]
                    [                                       ]
                r := [  2       2                           ]
                    [ q   q   q   q   q  0   1  0  0  0  0 ]
                    [                                       ]
                    [          2                            ]
                    [  0   0   q   q   q  0   0  1  0  0  0 ]
                    [                                       ]
                    [      2   3   2   2                    ]
                    [  0   q   q   q   q  0   q  q  1  0  0 ]
                    [                                       ]
                    [  2   3                   2            ]
                    [ q   q   0   q   0  0   q   0  q  1  0 ]
                    [                                       ]
                    [  3           2                        ]
                    [ q   0   0   q   0  0   0  0  0  q  1 ]

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

3

Installation

Être ce que nous sommes et devenir ce que nous sommes capables de devenir, tel est le seul but de la vie.

Robert Louis Balfour Stevenson.

This chapter is devoted to the installation of `ACE`. Before starting the installation, we should be aware of the following remark: even if `ACE` is available for any UNIX, MS-DOS or MACINTOSH system, for any release 1, 2, 3 or 4 of *Maple V*, we get different specific versions of the environment. That is to say, we get the UNIX version for *Maple V* release 2 which is not exactly the same as the UNIX version for *Maple V* release 3, and which is quite different from any MS-DOS version. We cannot say that sources differ between the UNIX, MS-DOS and MACINTOSH versions, but some of them have special instructions for instance for the releases 3 and 4 of *Maple V*. In any case, there are some specific programs (given only with UNIX versions) to compile the environment on a UNIX system, whereas there are other programs (given only with MS-DOS versions of `ACE`) for the installation on a MS-DOS system.

Remember also that `ACE` is available on a CD-ROM which provides several pre-installed versions of `ACE` for different systems. This can be useful to quickly install `ACE` on your system.

Before installing the environment, we should know the version which corresponds to our system (UNIX, MS-DOS or MACINTOSH) and its *Maple V* release (1, 2, 3, 4), and then refer to the corresponding section. The `ACE` environment is distributed (in its non-preinstalled version) for each UNIX or MS-DOS version, as one single archive file which contains sources and includes pre-computed data. In the case of the MACINTOSH versions, the archive file is different for some technical reasons.

3.1 Installing ACE on a UNIX system

This section is devoted to the installation of `ACE` in a UNIX environment.

The `ACE` environment is distributed as a compressed archive of files. First of all, we have to restore these files. According to the *Maple V* release `ACE` has to be installed for, we have the corresponding compressed archive file, `ACE1UNIX.tgz` (*Maple V* release 1), `ACE2UNIX.tgz` (*Maple V* release 2), `ACE3UNIX.tgz` (*Maple V* release 3) or `ACE4UNIX.tgz` (*Maple V* release 4). These files contain the sources and on-line help files of `ACE` together with the installation programs.

In the following, we will suppose that we get the `ACE3UNIX.tgz`, or in other words, we have to install the `ACE` environment on a UNIX operating system for *Maple V* release 3. Anyway, the same work has to be done for the other archive files.

3.1.1 Restoring sources

The `ACE3UNIX.tgz` file is a "`gzip`ed" file, that is to say, it has been compressed by the UNIX program `gzip`. At first, we have to uncompress this file by the following command:

```
gunzip ACE3UNIX.tgz
```

Note that the `gunzip` command can be replaced by `gzip -d`. This command creates a new file named `ACE3UNIX.tar` while the `ACE3UNIX.tgz` has disappeared.

This file has been created by using the `tar` command. To see the detail on the files contained in the archive, we run the UNIX command:

```
tar tvf ACE3UNIX.tar
```

and to restore the `ACE` sources, we run the following UNIX command:

```
tar xvf ACE3UNIX.tar
```

that creates a directory `ACE_PACK` in which all files will be extracted from the archive file.

3.1.2 Description of the `ACE_PACK` directory

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

The `BIN` subdirectory contains the `.modelinit` file that is a model of the `.mapleinit` *Maple* file that we should modify before installing `ACE`. It also contains the installation program, *i.e.* the `INSTALL.ACE` file (which is a C-shell program), the `INSTALL.SR1`, `INSTALL.SR2` and `INSTALL.HL1`, `INSTALL.HL2` files that describe how to respectively install source files and help files. We also give the `INSTALL.DAT` file which describes how to install pre-computed data. Finally, we provide in the `BIN` subdirectory the `ACE` program (that is a shell script) which can be used to run *Maple* and to automatically load `ACE`.

The `CPRIGHTS` subdirectory contains the `rights` file that gives the copyright notice.

The `PACKAGES` subdirectory contains the declarations of all packages included in `ACE`, whereas the `SOURCES` subdirectory contains the sources of all functions, together with some files devoted to the installation. Moreover, the `data` subdirectories of each directory of the `SOURCES` directory contain the corresponding pre-computed tables of data.

3.1.3 Description of a typical package

Let us choose the `FM` package as a typical one to present the organization of any package on the disk. It is essentially stored in the `FM` directory of respectively the `SOURCES` directory for the *Maple* sources,

installation files and data, and the `PACKAGES` subdirectory for the *Maple* declaration of the package.

The `SOURCES/FM` directory contains the `src` directory in which the *Maple* code of all functions of `FM` is, the `help` directory that clusters the *Maple* help files for each function of the package. Moreover, the `files` directory is used by the installation programs like `COMPILE.FM`. Finally, the `data` subdirectory contains the pre-computed tables of data for the `FM` package and the `test` directory contains a file `FM.tst` to test the installation (`read 'FM.tst';`).

The `PACKAGES/FM` directory contains three files: `FM` that declares the package, `FMext` which declares all functions used in the package and finally, `FMhlp` that declares to *Maple* where to find the compiled help files for the `FM` package.

3.1.4 Configuration file

Having restored all the environment on the disk, we are now able to make the installation.

First of all, we have to tell *Maple* where to store the compiled files (*i.e.* the `.m` files). This is done in the `.mapleinit` file in the `HOME` directory, by the use of a variable per package.

Since it is a non-trivial work to set all the variables, `ACELib, SGLib, SPLib, TABLib` and so on, this procedure has been integrated into the installation process so that we don't need to do it ourself. In other words, the installation program now asks whether we want it to modify or create our `.mapleinit` file.

However, if we still want to do it by hand, then here are the different steps to follow. We provide a model of the `.mapleinit` file, so-called `.modelinit` in the `BIN` directory. Here is this file for *Maple V* release 3:

```
##########
# File @(#) .modelinit (Maple) Sun Jan  1 17:01:45 Frh 1995
##########
#
#################################################################
#                                                               #
# Insert the following lines in the $HOME/.mapleinit file.      #
#                                                               #
#################################################################
#
## ACE package...
#
#######################################################################
#                                                                     #
# Modify the following line so that the ACELib variable  contains    #
# the directory where one wants maple to store the compiled files.    #
#                                                                     #
#######################################################################
```

```
#
ACELib:=‘/people/roger/maple/lib‘:
#
ACELibSeq:=SGLib, SPLib, SYMFLib, TABLib, TYPLib, HEKALib,  ERRLib,
           CLGLib, TEXLib, SGALib, NCALib, IDCALib, NCSFLib, FMLib,
           BNLib, BNALib, CGLib, COMPLib, PARTLib, FREELib, SFALib:
#
BNLib  :=‘‘.ACELib.‘/ACE‘:              BNALib :=‘‘.ACELib.‘/ACE‘:
CLGLib :=‘‘.ACELib.‘/ACE‘:              CGLib  :=‘‘.ACELib.‘/ACE‘:
COMPLib:=‘‘.ACELib.‘/ACE‘:              ERRLib :=‘‘.ACELib.‘/ACE‘:
FMLib  :=‘‘.ACELib.‘/ACE‘:              FREELib:=‘‘.ACELib.‘/ACE‘:
HEKALib:=‘‘.ACELib.‘/ACE‘:              IDCALib:=‘‘.ACELib.‘/ACE‘:
NCALib :=‘‘.ACELib.‘/ACE‘:              NCSFLib:=‘‘.ACELib.‘/ACE‘:
PARTLib:=‘‘.ACELib.‘/ACE‘:              SFALib :=‘‘.ACELib.‘/ACE‘:
SGALib :=‘‘.ACELib.‘/ACE‘:              SGLib  :=‘‘.ACELib.‘/ACE‘:
SPLib  :=‘‘.ACELib.‘/ACE‘:              SYMFLib:=‘‘.ACELib.‘/ACE‘:
TABLib :=‘‘.ACELib.‘/ACE‘:              TEXLib :=‘‘.ACELib.‘/ACE‘:
TYPLib :=‘‘.ACELib.‘/ACE‘:
#
## Where to find the packages...for Maple V Release 1
#
# _liblist:=[ACELib, ACELibSeq]:
#
## Where to find the packages...for Maple V Release > 1
#
  libname:=ACELib, ACELibSeq, libname:
```

Thus, to tell *Maple* where to compile (and later, where to find) the packages, we copy the `.modelinit` file into the existing `.mapleinit` file in the `HOME` directory. If we don't have any `.mapleinit` file, then we create it by directly copying the `.modelinit` file into a new `.mapleinit` file. Then, we modify the `ACELib` variable in the `.mapleinit` file, setting it to an **existing** directory. Note that, according to the given `.modelinit` file, all packages will be compiled into subdirectories of the `ACELib` directory but even if it is not necessary, it is easier to manage.

Global installation

Having modified the `.mapleinit` file, or having decided that this file will be automatically modified by the installation program, 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` (or simply `INSTALL.ACE`) which starts the installation:

```
INSTALL.ACE
```

The program asks several questions, in particular it asks whether we want to modify (update) our `.mapleinit` file. It also automatically creates a `ACE` file which can be used to run `maple` and load the `ACE` environment.

Individual installation

We also provide the possibility to compile each package individually. For this purpose, we give for each package its own installation program in its corresponding directory. For instance, we can compile the CLG package by running a UNIX command that looks like:

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

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

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

3.1.5 Having some troubles

The automatic installation program asks for the UNIX command to run to load *Maple*, so that now, we should not have any problem if *Maple* is not in the PATH variable. However, if there is a problem, that is to say, if the *Maple* command is unknown for the system, please follow the next advices. Since we run the csh shell to interpret the installation program, we should have the *Maple* program in the PATH variable while running this shell so that this trouble could be due to an incomplete .cshrc file (which is the configuration file of the csh shell program) in the HOME directory. Please refer to a documentation about csh to tell it where to find the *Maple* program: for instance, enter the UNIX command `man csh` that should give some hints about the problem.

Whenever there is a `writting error` message from *Maple* that looks like:

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

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

3.1.6 Using the environment

If you want someone to be able to use ACE, the first thing to do is to modify its .mapleinit to tell *Maple* where to find packages. For that, you need to copy the .mapleinit used to install ACE. It should contain the definitions of all Lib variables needed to use ACE. Note that the advantage of ACE is that even if the ACE compiled files have been moved in another directory (different from the one used for the installation), we can still use ACE by simply modifying the .mapleinit file without compiling again ACE.

Now, several possibilities are available. First of all, we can run *Maple* by the (usual) UNIX command maple, and then to load the complete ACE environment, that is to load all packages, we enter the *Maple* command `with(ACE);`. We can load a single package, say SYMF, by the *Maple* command `with(SYMF);`.

Another way to use the ACE environment is to execute the ACE UNIX command located in the BIN directory. It runs *Maple*, automatically prints the ACE banner and loads the ACE environment. When running ACE with the -x option, it runs *XMaple* instead of *Maple*. We can add the BIN directory to our PATH variable to have the ACE command accessible from anywhere. It can be done for instance in the corresponding shell configuration file. We also have to put the correct rights on the ACE file, with a command like: `chmod 755 ACE` if bad rights forbid the program to be executed. Normally, the automatic installation program must have correctly set the rights on this file.

3.2 Installing ACE on a MS-DOS system

This section deals with the installation of ACE in a MS-DOS environment. First, we have to extract files from the compressed archive files. According to the *Maple V* release we want ACE to be installed for, we get the corresponding compressed archive file, ACE1DOS.tgz (*Maple V* release 1), ACE2DOS.tgz (*Maple V* release 2), ACE3DOS.tgz (*Maple V* release 3) or ACE4DOS.tgz (*Maple V* release 4).

In the following, we will suppose that we get the ACE3DOS.tgz. ACE environment on a MS-DOS operating system for *Maple V* release 3. In any case, the same work has to be done for other releases, except for *Maple V* release 4 for which we don't need to modify any MAPLE.INI file.

3.2.1 Restoring sources

The ACE3DOS.tgz is a "gziped" file, that is to say, it has been compressed by the UNIX program gzip. We assume that this file is stored on a floppy disk named a: in the following.

To simplify the restoring phase on a MS-DOS system, we provide a program that can be used to uncompress the archive file and extract from it, in the same time, all files: this is the tar.exe file given on the floppy disk which is a "special" version of a UNIX compatible tar program including the gunzip program.

At first, being on the hard disk in the right directory, and assuming that we get this "special" tar program, we can uncompress the archive and extract all files by the following command:

```
a:tar.exe .xvf a:ACE3DOS.tgz
```

that creates a directory ACE_PACK. If we do not have this tar.exe program, files can be restored by first uncompressing the archive file and then extracting them from it (refer to the UNIX installation section).

3.2.2 Description of the ACE_PACK directory

This directory contains all sources of the ACE environment.

The BIN subdirectory contains the MAPLE.MOD file which is a model of the MAPLE.INI *Maple* file that we should modify to install ACE. It also contains the installation program, *i.e.* the INSTALL.BAT file (that is a batch file), the INSTALL.SR1, INSTALL.SR2 files and INSTALL.HL1, INSTALL.HL2 files which describe how to respectively install source files and help files. We also give the INSTALL.DAT file which describes how to install pre-computed data.

The CPRIGHTS subdirectory contains the rights file that gives the copyright notice.

The PACKAGES subdirectory contains the declarations of all packages included in ACE, whereas the SOURCES subdirectory contains the sources of all functions, together with some files devoted to the

installation. Moreover, the `data` subdirectories of each directory of the `SOURCES` directory contain the corresponding pre-computed tables of data.

3.2.3 Description of a typical package

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

The `SOURCES/FM` directory contains the `src` directory in which the *Maple* code of all functions of `FM` is, the `help` directory that clusters the *Maple* help files for each function of the package. Moreover, the `files` directory is used by the installation programs like `COMPFM.BAT`. The `FM.ini` file gives the lines that must be added to the `MAPLE.INI` file in the *Maple* library directory[(1)]. Finally, the `data` subdirectory contains the pre-computed tables of data for the `FM` package and the `test` directory contains a file `FM.tst` to test the installation (`read 'FM.tst';`).

The `PACKAGES/FM` directory contains three files: `FM` that declares the package, `FMEXT` which declares all functions used in the package and finally, `FMHLP` that declares to *Maple* where to find the compiled help files for the `FM` package.

3.2.4 Configuration file

Having restored all the environment on the disk, we are now able to make the installation.

If we use *Maple V* release 4, we can skip this section and continue at the next one. In other words, we don't need to modify any file before installing `ACE`.

First of all, we have to tell *Maple* where to store the compiled files (*i.e.* the `.m` files). This is done in the `MAPLE.INI` file in the *Maple* library directory (by default `\MAPLEV\LIB`), by using one variable per package. Since it is a non-trivial work to set all the variables, `ACELib, SGLib, SPLib, TABLib` and so on, we provide a model of rows that must be added in the `MAPLE.INI` file, so-called `MAPLE.MOD` in the `BIN` directory. Here is this file for *Maple V* release 3:

```
##########
# File @(#) MAPLE.MOD (Maple) Wed May 10 09:48:17 Fre 1995
##########
#
```

[(1)] which is usually the `\MAPLEV\LIB` directory.

```
######################################################################
#                                                                    #
# Insert the following lines in the \MAPLEV\LIB\maple.ini file.      #
#                                                                    #
######################################################################
#
## ACE package...
#
######################################################################
#                                                                    #
# Modify the following line so that the ACELib variable  contains  #
# the directory where one wants maple to store the compiled files.   #
#                                                                    #
######################################################################
#
ACELib:=`/MAPLEV/LIB`:
#
ACELibSeq:=SGLib, SPLib, SYMFLib, TABLib, TYPLib, HEKALib,  ERRLib,
           CLGLib, TEXLib, SGALib, NCALib, IDCALib, NCSFLib, FMLib,
           BNLib, BNALib, CGLib, COMPLib, PARTLib, FREELib, SFALib:
#
BNLib  :=``.ACELib.`/ACE`:              BNALib :=``.ACELib.`/ACE`:
CLGLib :=``.ACELib.`/ACE`:              CGLib  :=``.ACELib.`/ACE`:
COMPLib:=``.ACELib.`/ACE`:              ERRLib :=``.ACELib.`/ACE`:
FMLib  :=``.ACELib.`/ACE`:              FREELib:=``.ACELib.`/ACE`:
HEKALib:=``.ACELib.`/ACE`:              IDCALib:=``.ACELib.`/ACE`:
NCALib :=``.ACELib.`/ACE`:              NCSFLib:=``.ACELib.`/ACE`:
PARTLib:=``.ACELib.`/ACE`:              SFALib :=``.ACELib.`/ACE`:
SGALib :=``.ACELib.`/ACE`:              SGLib  :=``.ACELib.`/ACE`:
SPLib  :=``.ACELib.`/ACE`:              SYMFLib:=``.ACELib.`/ACE`:
TABLib :=``.ACELib.`/ACE`:              TEXLib :=``.ACELib.`/ACE`:
TYPLib :=``.ACELib.`/ACE`:
#
## Where to find the packages...for Maple V Release 1
#
# _liblist:=[ACELib, ACELibSeq]:
#
## Where to find the packages...for Maple V Release > 1
#
  libname:=ACELib, ACELibSeq, libname:
```

Thus, to tell *Maple* where to compile (and later, where to find) the packages, we insert all lines contained in the `MAPLE.MOD` file into the existing `MAPLE.INI` file in the *Maple* library directory (avoid to insert these lines at the end of the `MAPLE.INI` file)

Then, we modify the `ACELib` variable in the `MAPLE.INI` file, setting it to an **existing** directory that differs from the `\MAPLEV\LIB` one, but we recommand not to change this variable since one can have some troubles with the installation program to create the needed directories.

Note that, according to the given `MAPLE.MOD` file, all packages will be compiled into subdirectories of the `ACELib` directory but even if it is not necessary, it is easier to manage.

Global installation

For *Maple V* release 4, we simply go in the `BIN` directory and then we enter the MS-DOS command:

```
INSTALL.BAT
```

Then, we go to the next section.

For previous versions of *Maple V*, we have to use the following instructions. Having modified the `MAPLE.INI` file, we are ready to install all packages by using the `INSTALL.ACE` program in the `BIN` directory. Being in this directory, enter the MS-DOS command, `INSTALL.BAT` which prints some information about the installation among which it is said that this program takes as argument the directory where the packages must be installed, and this directory is exactly the same as the directory specified by the `ACELib` variable in the `MAPLE.INI` file. Then, to install the environment, enter a MS-DOS command like:

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

3.2.5 Using the environment

We can use `ACE` by running *Maple* and then loading the complete `ACE` environment (all packages) by the *Maple* command `with(ACE);`. We can load a single package, say `SYMF`, by the *Maple* command `with(SYMF);`.

3.3 Installing ACE on a MACINTOSH system

This section deals with the installation of `ACE` on a MACINTOSH. `ACE` is both available as a "`gzip`ed" tar archive file and a BinHexed archive. First, we have to extract files from the archive files. According to the *Maple V* release we want `ACE` to be installed for, we get either the corresponding compressed tar file `ACE1MAC.tgz` (*Maple V* release 1), `ACE2MAC.tgz` (*Maple V* release 2), `ACE3MAC.tgz` (*Maple V* release 3) or `ACE4MAC.tgz` (*Maple V* release 4) or BinHexed archive file, `ACE1MAC.sit.hqx` (*Maple V* release 1), `ACE2MAC.sit.hqx` (*Maple V* release 2), `ACE3MAC.sit.hqx` (*Maple V* release 3) or `ACE4MAC.sit.hqx` (*Maple V* release 4). Once downloaded, we have to extract all files from the archive using the `gunzip` and `tar` programs for MACINTOSH or `Stuffit Expander` application (for `.sit.hqx` files). This last application is shareware available on every **infomac archive** on the Internet.

Remark. Note that we have not tested the installation of the MACINTOSH version on several systems. We only have installed `ACE` on a MACINTOSH running a very old release of *Maple V*. Do not hesitate to contact us if you have any trouble with the MACINTOSH installation and furthermore please send us emails if you have ideas to improve the installation of `ACE` on a MACINTOSH system. Because we had not a recent version of *Maple V*, we were not able to do a better automatic installation program.

In the following, we will suppose that we get the `ACE3MAC.sit.hqx`, or in other words, we have to install the `ACE` environment on a MACINTOSH system for *Maple V* release 3.

3.3.1 Restoring sources

The `ACE3MAC.sit.hqx` file is a BinHexed archive. To extract all files from the archive, we have to **drop** the `ACE3MAC.sit.hqx` file on the `Stuffit Expander` icon. It will create a folder `ACE_PACK` in which all files will be extracted from the archive file. It will also create a folder `ACE` (containing subfolders but without any file), and a `MAPLEINIT` file. These two folders and the `MAPLEINIT` have to be moved into the `lib` folder next to the `Maple` folder, before installing the `ACE` environment.

3.3.2 Description of the `ACE_PACK` folder

The `ACE_PACK` folder contains all sources of the `ACE` environment.

The `ACE_PACK/BIN` folder contains the installation program, *i.e.* the `INSTALL.ACE` file (which is a *Maple* file), the `INSTALL.SR1`, `INSTALL.SR2` files and `INSTALL.HL1`, `INSTALL.HL2` files which describe how to respectively install source files and help files. We also give the `INSTALL.DAT` file which describes how to install pre-computed data.

The `ACE_PACK/CPRIGHTS` subfolder contains the `rights` file that gives the copyright notice.

The `ACE_PACK/PACKAGES` subfolder contains the declarations of all packages included in `ACE`, whereas the `ACE_PACK/SOURCES` subfolder contains the sources of all functions, together with some files devoted to the installation. Moreover, the `data` subfolders of each folder of the `ACE_PACK/SOURCES` folder contain the corresponding pre-computed tables of data.

3.3.3 Description of a typical package

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

The `SOURCES/FM` folder contains the `src` folder in which the *Maple* code of all functions of `FM` is, the `help` folder that clusters the *Maple* help files for each function of the package. Finally, the `data` subfolder contains the pre-computed tables of data for the `FM` package and the `test` subfolder contains a file `FM.tst` to test the installation (`read 'FM.tst';`).

The `PACKAGES/FM` folder contains three files: `FM` that declares the package, `FMext` which declares all functions used in the package and finally, `FMhlp` that declares to *Maple* where to find the compiled help files for the `FM` package.

3.3.4 Configuration file

Having restored all the environment on the disk, we are now able to make the installation.

First of all, we have to tell *Maple* where to store the compiled files (*i.e.* the `.m` files). This is done in the `MAPLEINIT` file in the *Maple* library directory (by default `:lib`), by using one variable per package. Since it is a non-trivial work to set all the variables, `ACELib, SGLib, SPLib, TABLib` and so on, we provide a model of the `MAPLEINIT` file that we have previously moved into the `lib` folder next to the `Maple` folder. Here is this file for *Maple V* release 3:

```
##########
# File @(#) MAPLEINIT (Maple) Wed Feb  7 17:29:13 MET 1996
##########
#
#########################################################
#                                                       #
# Insert the following lines in the MAPLEINIT file in #
# the folder  given  by  the libname MAPLE variable.  #
# It should be by default :lib.                         #
#                                                       #
#########################################################
#
## ACE package...
#
#############################################################
#                                                           #
# DO NOT modify the following line  except  if  Maple   #
# library has been moved and is not in the folder :lib. #
#                                                           #
#############################################################
#
ACELib:=``.libname:
#
ACELibSeq:=SGLib, SPLib, SYMFLib, TABLib, TYPLib, HEKALib,  ERRLib,
           CLGLib, TEXLib, SGALib, NCALib, IDCALib, NCSFLib, FMLib,
           BNLib, BNALib, CGLib, COMPLib, PARTLib, FREELib, SFALib:
#
BNLib  :=``.ACELib.`:ACE`:              BNALib :=``.ACELib.`:ACE`:
CLGLib :=``.ACELib.`:ACE`:              CGLib  :=``.ACELib.`:ACE`:
COMPLib:=``.ACELib.`:ACE`:              ERRLib :=``.ACELib.`:ACE`:
FMLib  :=``.ACELib.`:ACE`:              FREELib:=``.ACELib.`:ACE`:
HEKALib:=``.ACELib.`:ACE`:              IDCALib:=``.ACELib.`:ACE`:
NCALib :=``.ACELib.`:ACE`:              NCSFLib:=``.ACELib.`:ACE`:
PARTLib:=``.ACELib.`:ACE`:              SFALib :=``.ACELib.`:ACE`:
SGALib :=``.ACELib.`:ACE`:              SGLib  :=``.ACELib.`:ACE`:
SPLib  :=``.ACELib.`:ACE`:              SYMFLib:=``.ACELib.`:ACE`:
TABLib :=``.ACELib.`:ACE`:              TEXLib :=``.ACELib.`:ACE`:
TYPLib :=``.ACELib.`:ACE`:
#
## Where to find the packages...for Maple V Release 1
#
# _liblist:=[ACELib, ACELibSeq]:
#
## Where to find the packages...for Maple V Release > 1
#
  libname:=ACELib, ACELibSeq, libname:
```

We do not in general have to modify `ACELib`. *Maple* should find installed libraries.

Global installation

We are ready to install all packages by using the `INSTALL.ACE` *Maple* program in the `ACE_PACK/BIN` folder. Run *Maple* and enter the following *Maple* command:

```
read ‘:lib:ACE_PACK:BIN:INSTALL.ACE‘;
```

which prints instructions to install `ACE`. To follow the instructions, we recommand to select all lines (output of the previous *Maple* command), put them as `input` lines, and save them as a *Maple* `Worksheet`. Then, quit and click on the created icon to run *Maple* and **execute one block of commands each time**. For that purpose, we recommand `cut` and `copy` technic because we have to run *Maple* about 20 times[(2)]. Blocks of commands are separated by an empty line and each block tells *Maple* to terminate the session each time.

Remark. We can see the result of each block of commands by clicking on `Cancel` when *Maple* asks us to confirm we want to terminate the session. This can be useful to detect that there is a problem.

3.3.5 Using the environment

We can use `ACE` by running *Maple* and then loading the complete `ACE` environment (all packages) by the *Maple* command `with(ACE);`. We can load a single package, say `SYMF`, by the *Maple* command `with(SYMF);`.

[(2)] We did not succeed in having an easier way to install `ACE`, but fill free to inform us if you have better ideas.

4

Examples

This chapter presents some examples provided with the environment. These examples are given in the subdirectories of the `EXAMPLES` directory. First, we compute Grothendieck polynomials and Key polynomials, then a section deals with projective degrees of Schubert varieties and Todd classes of flag varieties. The next section is devoted to computations with the `NCSF` package. The following section deals with generating functions for Schubert polynomials. The last section presents the action of divided differences on so-called modified powers and its implementation through the `SFA` package.

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 this 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(Perm2Inv(perm), GenPerm(n, 'last'))...
  n:=nops(perm);
  wperm:=MultPerm(Perm2Inv(perm), GenPerm(n, 'last'));
  RETURN(simplify(IdcaOnPol(Idca2Demazure(A[op(wperm)]), GrothMax(n))))
end;
```

The next function computes the table of all Grothendieck polynomials for $\mathfrak{S}_n$:

```
TableGG:=proc(n)
  local
    p, # a permutation...
    t; # the result...
  t:=table();
  for p in ListPerm(n) do
    t[p]:=Groth(p)
  od;
  RETURN(eval(t))
end;
```

The following command demonstrates the use of this function:

```
ACE> TableGG(3);
```

```
                        table([
                                          (x1 - y2) (- x1 + y1)
                            [3, 1, 2] = - ---------------------
                                                    2
                                                  x1

                                          - x1 + y1
                            [2, 1, 3] = - ---------
                                              x1
```

```
                                    (- x1 + y1) (- x2 + y1)
                        [2, 3, 1] = -----------------------
                                            x1 x2

                                    (- x1 + y1) (x1 - y2) (- x2 + y1)
                        [3, 2, 1] = ---------------------------------
                                                    2
                                                  x1  x2

                        [1, 2, 3] = 1

                                      - x2 x1 + y2 y1
                        [1, 3, 2] = - ---------------
                                            x1 x2
    ])
```

When specializing $y_i \longrightarrow 1$ and $x_i \longrightarrow 1/(1-x_i)$, Grothendieck polynomials become equal to Schubert polynomials $\mathbf{X}_{perm}+$ terms of higher degree. The following function computes the table of all Grothendieck polynomials for $\mathfrak{S}_n$ expressed in the $\mathbf{X}$ Schubert basis:

```
TableGG2X:=proc(n)
  local
    i, # variable for sequence...
    t; # a table...
  t:=TableGG(n);
  t:=subs(seq(y.i=1,i=1..n-1), eval(t));
  t:=subs(seq((x.i)=1/(1-(x.i)),i=1..n-1), eval(t));
  RETURN(map(x2X, map(simplify, eval(t))))
end;
```

The next example illustrates the use of this function for the symmetric group $\mathfrak{S}_3$:

```
ACE> TableGG2X(3);

                        table([
                            [3, 1, 2] = X[3, 1, 2]
                            [2, 1, 3] = X[2, 1]
                            [2, 3, 1] = X[2, 3, 1]
                            [3, 2, 1] = X[3, 2, 1]
                            [1, 2, 3] = X[1]
                            [1, 3, 2] = - X[2, 3, 1] + X[1, 3, 2]
                        ])
```

4.2 Key polynomials

A key polynomial is the image of a dominant monomial $x_1^{i_1}x_2^{i_2}\ldots$, $i_1 \geq i_2 \geq \cdots$, under a Demazure operator. The following program gives a way to compute these polynomials using the `ACE` environment.

The `ListDom` function returns the list of all partitions which are smaller than $[n-1, n-2, ..., 0]$:

```
ListDom:=proc(n)
  local
    i;
  RETURN(map(proc(l,n) [op(l),0$(n-nops(l))] end,
        ListPartIn([seq(-i,i=-(n-1)..-1)]), n))
end;
```

`KeyPol`(*cod*) computes the key polynomial corresponding to a permutation given by its code:

```
KeyPol:=proc(cod)
  local
    i,    # variable for sequence...
    j,    # variable for loop...
    pos,  # a position...
    perm, # a permutation...
    rcod, # a code...
    tmp;  # a temporary variable...
  rcod:=sort(cod,proc(x,y) evalb(x>=y) end);
  perm:=NULL;
  tmp:=rcod;
  for j from 1 to nops(cod) do
    member(cod[j], tmp, 'pos');
    perm:=perm, pos;
    tmp:=subsop(pos=-1, tmp);
  od;
  perm:=[perm];
  RETURN(IdcaOnPol(
           Idca2Demazure(A[op(Perm2Inv(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 Perm2Interval(GenPerm(n, 'last')) 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 $\mathbf{Y}$ Schubert basis.

```
TableKY:=proc(n)
  RETURN(eval(map(ToY, TableK(n))))
end;
```

Since the permutation $w := [2, 1, 4, 3]$ (whose Lehmer code is $[1, 0, 1, 0]$) is the only non-vexillary permutation of $\mathfrak{S}_4$, we have the following result:

```
ACE> t:=TableKY(4):
ACE> t[ [1,0,1,0] ];

                        - Y[2, 0, 0] + Y[1, 0, 1, 0]
```

4.3 Projective degree and Todd class

4.3.1 Projective degree of a Schubert variety

The class of an hyperplane section of the flag manifold $\mathcal{F}(\mathbb{C}^n)$, for the Plücker embedding, is equal to $h := (n-1)x_1 + (n-2)x_2 + \cdots + x_{n-1}$. The projective degree of a Schubert variety is by definition the coefficient of $\mathbf{X}_\omega$ in the product $\mathbf{X}_{perm} \times h^{\binom{n}{2}-l(perm))}$ in which ω denotes the maximal permutation of $\mathfrak{S}_n$ and $l(perm)$ the length of the permutation $perm$:

```
h:=proc(n)
  local
    i; # variable for sequence...
  option remember;
  RETURN(convert([seq((n-i)*x.i, i=1..n-1)],`+`))
end;

ProjDegree:=proc(perm)
  local
    r,     # the result...
    n,     # degree of the symmetric group...
    sFlag, # a value...
    d;     # the projective degree of the Schubert variety...
  n:=nops(perm);
  d:=n*(n-1)/2 - Perm2Length(perm);
  sFlag:=_Sn;
  Flag(n);
  r:=subs(X[op(GenPerm(n, 'last'))]=1, ToX(h(n)^d*X[op(perm)]));
  if (sFlag<>-1) then Flag(sFlag) else Flag() fi;
  RETURN(r)
end;
```

Here is an example showing how to compute all projective degrees for all permutations of $\mathfrak{S}_4$:

```
ACE> ListPerm(4);

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

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

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

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

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

ACE> seq(ProjDegree(perm), perm=ListPerm(4));

  720, 46, 220, 6, 280, 16, 48, 3, 48, 2, 14, 1, 220, 12, 78, 3, 46, 3, 6, 1, 16, 1, 3, 1
```

4.3.2 Todd class of the flag variety

The total Todd class of the flag variety $\mathcal{F}(\mathbb{C}^n)$ is by definition equal to the class of $e^{\sum_{1\leq i<j\leq n}(x_i-x_j)}$, in the cohomology ring, and thus coincides with $e^{h(n)}$ where $h(n)$ has been defined in the previous subsection. The following function computes this Todd class:

```
Todd:=proc(n)
  local
    pol,   # a polynomial...
    hk,    # power of h...
    sFlag, # a value...
    k;     # variable for loop...
  pol:=1;
  hk:=1;
  sFlag:=_Sn;
  Flag(n);
  for k from 1 to n*(n-1)/2 do
    hk:=ToX(hk*h(n));
    pol:=pol + hk/factorial(k)
  od;
  if (sFlag<>-1) then Flag(sFlag) else Flag() fi;
  RETURN(pol)
end;
```

Here is the Todd class of the flag variety $\mathcal{F}(\mathbb{C}^4)$:

```
ACE> Todd(4);

 1 + X[2, 1, 3, 4] + X[1, 3, 2, 4] + X[1, 2, 4, 3] + 3/2 X[3, 1, 2, 4]

      + 3/2 X[2, 3, 1, 4] + X[2, 1, 4, 3] + 3/2 X[1, 4, 2, 3]

      + 3/2 X[1, 3, 4, 2] + 8/3 X[4, 1, 2, 3] + X[3, 2, 1, 4] + 2 X[3, 1, 4, 2]

      + 7/3 X[2, 4, 1, 3] + 8/3 X[2, 3, 4, 1] + X[1, 4, 3, 2] + 2 X[4, 2, 1, 3]

         23                                        23
      + ---- X[4, 1, 3, 2] + 13/4 X[3, 4, 1, 2] + ---- X[3, 2, 4, 1]
         12                                        12

      + 2 X[2, 4, 3, 1] + 11/6 X[4, 3, 1, 2] + 7/3 X[4, 2, 3, 1]

      + 11/6 X[3, 4, 2, 1] + X[4, 3, 2, 1]
```

4.4 Some computations with NCSF

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

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

is a Lie idempotent. In terms of noncommutative symmetric functions, this element is the specialization at $q = \varepsilon$ of the symmetric function $\frac{1}{n} K_n(q)$, where $K_n(q)$ is given by the generating series:

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

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

```
ACE> C:=ListCompo(4):
ACE> K:=0:
ACE> for x in C do K:=K+ q^Compo2MajorIndex(x)*R[op(x)] od:
ACE> K:=K/4:
ACE> K:=subs(q=I, K):
```

To check that it is an idempotent, we enter:

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

We have to compare with `ToS(K)` because the output of `Internal` is given in the S-basis.

To see that κ_4 projects onto the Free Lie algebra, we may compute its left and right products with a known Lie idempotent, for example Dynkin's element $\theta_4 := \frac{1}{4}[[[1,2],3],4]$, which corresponds to the symmetric function $\frac{1}{4}\Psi_4$:

```
ACE> T:=ToS(Ps[4]/4):
ACE> r := NcsfInternal(T, K) - K:
ACE> ToR(r);
                                    0

ACE> r := NcsfInternal(K, T) - T:
ACE> ToR(r);
                                    0
```

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

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

where A is just a label. This transformation is computed by the `NcsfTransAlpha` function:

```
ACE> r := NcsfTransAlpha(S[3], 'direct', 'q'):
ACE> simplify(ToR( (1-q)*(1-q^2)*(1-q^3)*r));

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

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

```
ACE> r := NcsfTransAlpha(S[4], 'inverse', 'q'):
ACE> simplify(ToR(r/(1-q)));

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

One can show that the element:

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

is a Lie idempotent [45, 53], interpolating between several known ones:

```
ACE> varphi:=proc(n ,q)
       local r;
       r := (1-q^n)/n*NcsfTransAlpha(Ps[n], 'direct', q);
       r := ToR(r, 'collect');
       r := map(normal, r);
       RETURN(r);
     end:

ACE> P:=varphi(4, 'q');

                                                 2
                       R[1, 3]          q  R[3, 1]          q R[1, 2, 1]
 P := 1/4 R[4] - 1/4 ---------- - 1/4 ---------- + 1/4 ------------
                      2                 2                   2
                     q  + q + 1        q  + q + 1          q  + q + 1
```

```
                              2
           q R[2, 2]       q  R[2, 1, 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> r := subs(q=1, P): ToPh(r);

                                                 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(NcsfTransAlpha(S[n], 'inverse', 'q'));
  res:=split(res,t);
  res:=NcsfTransAlpha(res, 'direct', 'q');
end;
#
## Individual q-Eulerians on the R-basis.
#
qEuler:=proc(n,k,q);
  ToR( coeff(expand(GFqE(n,q,`t`)), `t`, k) );
end;
```

For instance:

```
ACE> r := qEuler(3, 1, 'q'):
ACE> map(simplify, ToR(r, 'collect'));

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

ACE> r := qEuler(3, 2, 'q'):
ACE> map(simplify, ToR(r, 'collect'));

       R[2, 1] (- 1 + q)                                  R[1, 2] (- 1 + q)
   1/6 ----------------- - 1/2 R[1, 1, 1] + 1/2 R[3] - 1/6 -----------------
             q + 1                                              q + 1

ACE> r := qEuler(3, 3, 'q'):
ACE> map(simplify, ToR(r, 'collect'));

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

4.5 Generating functions for Schubert polynomials ($\mathfrak{S}_n$ and B_n)

Schubert polynomials for the symmetric group can be expressed as certain coefficients in the algebra of divided differences [78] or nilCoxeter algebra [85]. The generating function of Fomin and Kirillov is:

$$\mathfrak{S}(\mathbf{x};\mathbf{y}) = C_1(x_1)\,C_2(x_2)\ldots C_{n-1}(x_{n-1}) \quad , \tag{4.1}$$

$$= \sum_{\mu\in\mathfrak{S}_n} \mathbb{X}_\mu(\mathbf{x},\mathbf{y})\ \partial_\mu \quad , \tag{4.2}$$

in which $C_i(x) = \Big(1+(x-y_{n-i})\partial_{n-1}\Big)\Big(1+(x-y_{n-i-1})\partial_{n-2}\Big)\ldots\Big(1+(x-y_1)\partial_i\Big)$ and the x_i and y_i commute with the ∂_μ.

Here is a program computing this generating function. The function `C` computes a $C_i(x_i)$ by evaluating the product of all `(1 + (x - y_alphabet[j-i+1])*Transpo(j, n))` in the nilCoxeter algebra (`NcaMult`):

```
C:=proc(n, i, x, y_alphabet)
  local
    j, # variable for loop...
    r; # the result...
  r:=1;
  for j from i to n-1 do
    r := NcaMult((1 + (x - y_alphabet[j-i+1])*A[op(SgTranspo(j, n))]), r)
  od;
  RETURN(r)
end;
```

The call `Transpo(j, n)` returns the elementary transposition exchanging j and $j+1$ in $\mathfrak{S}_n$. Then,

the generating function of double Schubert polynomials corresponds to the following instructions:

```
An_Schubertfg:=proc(x_alphabet, y_alphabet)
  local
    i, # variable for loop...
    n, # degree of the symmetric group...
    r; # the result...
  n:=nops(x_alphabet);
  r:=1;
  for i from 1 to n-1 do
    r := NcaMult(r, C(n, i, x_alphabet[i], y_alphabet))
  od;
  RETURN(r)
end;
```

For instance, we have for $\mathfrak{S}_3$:

```
ACE> An_Schubertfg([x1, x2, x3], [y1, y2, y3]);

     A[1, 2, 3] + (x1 - y1) A[2, 1, 3] + (x1 - y2) (x1 - y1) A[3, 1, 2]

      + (x2 - y1 + x1 - y2) A[1, 3, 2] + (x1 - y1) (x2 - y1) A[2, 3, 1]

      + (x1 - y1) (x1 - y2) (x2 - y1) A[3, 2, 1]
```

B_n can be canonically embedded into $\mathfrak{S}_{2n}$. More explicitly, it amounts to identify the generators $u_1, \ldots, u_{n-1}$ of the B_n-nilCoxeter algebra with:

$$\begin{aligned} u_{n-1} &= \partial_1 \partial_{2n-1} \quad , \\ u_{n-2} &= \partial_2 \partial_{2n-2} \quad , \\ \vdots \quad &\vdots \qquad\quad \vdots \\ u_1 &= \partial_{n-1} \partial_{n+1} \quad , \end{aligned}$$

and the extra generator u_0 to ∂_n.

The generating function of B_n-Schubert polynomials of Fomin and Kirillov is:

$$\mathrm{FK}(\mathbf{x}) = B(x_1)\,B(x_2)\ldots B(x_n)\,C'_1(-x_1)\,C'_2(-x_2)\ldots C'_{n-1}(-x_{n-1}) \tag{4.3}$$

$$= \sum_{\nu \in B_n} \mathfrak{X}_\nu(\mathbf{x})\; u_\nu \tag{4.4}$$

$$= \sum \mathfrak{X}_\nu(\mathbf{x})\; \partial_\mu \quad , \tag{4.5}$$

μ being the image of ν in $\mathfrak{S}_{2n}$, $B(x) = (1+xu_{n-1})\ldots(1+xu_1)\,(1+xu_0)\,(1+xu_1)\ldots(1+xu_{n-1})$ and $C'_i(x) = (1+xu_{n-1})\,(1+xu_{n-2})\ldots(1+xu_i)$. First of all, the `BnTranspoPalin` function realizes the embedding of B_n into $\mathfrak{S}_{2n}$, for a simple transposition. Now, the call `B(n, x)` stands for $B(x)$ whereas `Bn_C(n, i, x)` computes $C'_i(x)$:

```
B:=proc(n, x)
  local
    j, # variable for loop...
    r; # the result...
  r:=1;
  for j from n-1 by -1 to 0 do
    r := BnNcaMult(r,
           (1 + x*B[op(`BN/BnPalin2Code`(`BN/BnTranspoPalin`(j, n)))]))
  od;
  for j from 1 to n-1 do
    r := BnNcaMult(r,
           (1 + x*B[op(`BN/BnPalin2Code`(`BN/BnTranspoPalin`(j, n)))]))
  od;
  RETURN(r)
end;

Bn_C:=proc(n, i, x)
  local
    j, # variable for loop...
    r; # the result...
  r:=1;
  for j from i to n-1 do
    r := BnNcaMult(
           (1 + x*B[op(`BN/BnPalin2Code`(`BN/BnTranspoPalin`(j, n)))]), r)
  od;
  RETURN(r)
end;

Bn_Schubertfg:=proc(x_alphabet)
  local
    i, # variable for loop...
    n, # B(n)...
    r; # the result...
  n:=nops(x_alphabet);
  r:=1;
  for i from 1 to n do
    r := BnNcaMult(r, B(n, x_alphabet[i]))
  od;
  for i from 1 to n-1 do
    r := BnNcaMult(r, Bn_C(n, i, -x_alphabet[i]))
  od;
  RETURN(r)
end;
```

For instance, we obtain for $n = 2$:

```
ACE> r := map(factor, Bn_Schubertfg([x1, x2]));
```

```
                                          2
r := x2 (x1 + x2) B[1, 1] + (x1 + x2)  B[0, 2] + (x1 + x2) B[1, 0] + B[0, 0]

                                        2
      + x1 x2 (x1 + x2) B[1, 2] + x2  (x1 + x2) B[0, 3] + (x1 + 2 x2) B[0, 1]

             2
      + x1 x2  (x1 + x2) B[1, 3]

ACE> BnaCode2Palin(r);
                                         2
  x2 (x1 + x2) A[3, 1, 4, 2] + (x1 + x2)  A[2, 4, 1, 3] + (x1 + x2) A[1, 3, 2, 4]

     + A[1, 2, 3, 4] + x1 x2 (x1 + x2) A[3, 4, 1, 2]

         2
     + x2  (x1 + x2) A[4, 2, 3, 1] + (x1 + 2 x2) A[2, 1, 4, 3]

           2
     + x1 x2  (x1 + x2) A[4, 3, 2, 1]
```

There are other families of B_n-Schubert polynomials. Fomin and Kirillov have defined a second one and shown the link with a third one due to Biley and Haiman. Pragacz and Ratajski [81] describe still another one related to geometrical considerations, and a fifth one is due to Fulton.

4.6 Divided differences on modified powers

Divided differences are operators acting on functions of several variables. We describe this action on the basis of modified powers (that is x^k is replaced by $(x-a_0)(x-a_1)..(x-a_{n-1})$). Special cases are of interest when $a_i = q^i$ or $a_i = i$. We use the `SFA` package.

4.6.1 Complete Functions of Alphabets

Let $A = \{a_1, a_2, ...\}$ be an infinite totally ordered alphabet, A_n denoting the finite subalphabet $\{a_1, a_2, .., a_n\}$.

Let $v = [v_1, v_2, ...]$ be an infinite vector of $\mathbb{N}^\infty$.

Then $v \odot A$ denotes the alphabet $\{a_1, .., a_1, a_2, .., a_2, ..., a_i, .., a_i, ...\}$, each a_i occurring v_i times. More generally one could take $v \in \mathbb{R}^\infty$, $v_1a_1 + v_2a_2 + ... + v_ia_i + ...$ to be another notation for $v \odot A$.

Let p^n be the vector $[p, .., p, 0, .., 0, ...]$, with n occurrences of p. In particular, $A_n = 1^n \odot A$.

Let us illustrate the definitions :

$$\sum_{k\geq 0} z^k S_k\Big(([3,3,-2,0,..,0,...] \odot A) - 5\Big) = \frac{(1-za_3)^2(1-z)^5}{(1-za_1)^3(1-za_2)^3} \quad . \tag{4.6}$$

4.6.2 Action of Divided Differences on Modified Powers

To each totally ordered alphabet X, Newton has associated operators ∂_1, ∂_2, ... acting on any function of A (on the left) according to the rule:

$$f\partial_i = \frac{f(..,x_i,x_{i+1},..) - f(..,x_{i+1},x_i,..)}{x_i - x_{i+1}} \quad . \tag{4.7}$$

We shall write in short $\partial^k = \partial_1\partial_2..\partial_k$, where ∂^0 is the identity operator.

Definition 4.6.1 *Let A be an alphabet. $x^{[n]} := \prod_{i=0}^{n-1}(x - a_i)$ is called the nth modified power of x.*

Let $\lambda = [\lambda_1, \lambda_2, .., \lambda_\ell]$ be a partition. Let Φ be the morphism which sends any monomial $\mathbf{x}^\lambda = x_1^{\lambda_1}x_2^{\lambda_2}..x_\ell^{\lambda_l}$ onto $\Phi(\mathbf{x}^\lambda) := \mathbf{x}^{[\lambda]} = {x_1}^{[\lambda_1]}{x_2}^{[\lambda_2]}..{x_\ell}^{[\lambda_\ell]}$.

We put $m_\emptyset = 0$, $m_{[\,]} = 1$ for the monomial functions.

The result of the action of divided differences on a modified power is:

Theorem 4.6.1 *Let $A = \{a_1, a_2, ...\}$ be an infinite totally ordered alphabet and a_0 an extra variable. Let ${x_1}^{[n]} := \prod_{i=0}^{n-1}(x_1 - a_i)$. Then:*

$${x_1}^{[n]}\partial^k = \sum_\lambda S_{n-|\lambda|-k}\left(ka_0 - A_{n-1} + \sum_{j=1}^{\ell(\lambda)} A_{\lambda_j}\right)\Phi(m_\lambda(x_1,..,x_{k+1})) \quad . \tag{4.8}$$

Further explainations can be found in [82].

We first program some shorthands:

```
# Summing list elements
SumIt := proc(l) convert(l, `+`); end:

# Creating alphabet An=a1..an
Alphab := proc(a,n) local i; SumIt([seq(a.i, i=1..n)]); end:

# Tranforming a list of lists of lists into a list of lists.
# [[x1],[x2,x3],[x4],[x5,x6,x7]] -> [x1,x2,x3,x4,x5,x6,x7]
Flat := l -> map(op, l):
```

Let us implement the formula in theorem (4.6.1):

```
# Declaring valid variables for SfAExpand
SfAVars({{a}}):

DdOnMp := proc(n,k)
  local
    lp,  # list of partitions...
    i,j; # vars for seq...
  lp := Flat([seq(ListPart(i), i=0..n-k)]);
  SumIt(
  [seq(
    h[subs(0=NULL, n-SumIt(lp[i])-k)]
            (k*'a0'-Alphab('a',n-1)+
             SumIt([seq(Alphab('a',op(j,lp[i])), j=1..nops(lp[i]))])
            )
    * Phi(m[op(lp[i])](Alphab('x',k+1))),
  i=1..nops(lp))]);
end:
```

Let us take an example:

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

          h[2](2 a0 - a1 - a2 - a3) Phi(m[](x1 + x2 + x3))

              + h[1](2 a0 - a2 - a3) Phi(m[1](x1 + x2 + x3))

              + h[](2 a0 - a3) Phi(m[2](x1 + x2 + x3))

              + h[](2 a0 + a1 - a2 - a3) Phi(m[1, 1](x1 + x2 + x3))
```

Two specializations of this result are of interest: $A = \{1, q, q^2, ...\}$ and $A = \{0, 1, 2, ...\}$. The first one is related to the theory of hypergeometric functions, and the second one is related to Stirling's interpolation.

4.6.3 q-specialization

We set $a_i = q^i$, which we call the q-specialization of the $x^{[k]}$'s. Let $x^{[n]} = \prod_{i=0}^{n-1}(x - q^i)$. Let $\mathbf{q}_n = \{q^0, .., q^{n-1}\}$. Application of formula (4.8) for $k > 0$ leads to:

$$x_1{}^{[n]}\partial^k = \sum_\lambda S_{n-|\lambda|-k}\left(\sum_{j=2}^{k+1}\mathbf{q}_{\lambda_j+1} - (\mathbf{q}_n - \mathbf{q}_{\lambda_1+1})\right)\Phi(m_\lambda(x_1, .., x_{k+1})) \quad . \tag{4.9}$$

For $k = 1$, we get:

$$x_1{}^{[n]}\partial_1 = \sum_\lambda S_{n-|\lambda|-1}(\mathbf{q}_{\lambda_2+1} - (\mathbf{q}_n - \mathbf{q}_{\lambda_1+1}))\,\Phi(m_\lambda(x_1, x_2)) \quad . \tag{4.10}$$

Proposition 4.6.1 *In equation* (4.10)*, the coefficients factorize. More precisely:*

$$S_{n-|\lambda|-1}(\mathbf{q}_{\lambda_2+1} - (\mathbf{q}_n - \mathbf{q}_{\lambda_1+1})) = S_{n-|\lambda|-1}(\mathbf{q}_{\lambda_2+1}) \prod_{i=\lambda_1+1}^{n-\lambda_2-1} (1-q^i) \quad . \tag{4.11}$$

Let us verify this proposition with `SFA`:

```
# Multiplying list elements
MultIt := proc(l) convert(l, `*`); end:

# A new op function on partitions that returns 0 if not found
PartOp := proc(i,l) if (i>nops(l)) or (i<0) then 0 else op(i,l) fi end:

# Creating alphabet An=a0..a{n-1}
qAlphab := proc(a,n) local i; SumIt([seq(a.i, i=0..n-1)]); end:

# q-specialization
qSpec := proc(f) local i; subs(seq(a.i=q^i, i=0..100), f) end:
```

Here is the implementation of formula (4.11):

```
qDdOnMp := proc(n)
  local
    lp, # list of partitions...
    i,  # var for seq...
    j;  # var for seq...
  lp := Flat([seq(ListPart(i), i=0..n-1)]);
  subs(seq(a.i=q^i,i=0..n-1),
  SfAExpand(
  SumIt(
  [seq(
    h[subs(0=NULL, n-SumIt(lp[i])-1)] (qAlphab('a',PartOp(2,lp[i])+1))
    * MultIt([seq(1-a.j, j=PartOp(1,lp[i])+1..n-PartOp(2,lp[i])-1)])
    * Phi(m[op(lp[i])](x1+x2)),
  i=1..nops(lp))]))));
end:
```

We easily check the proposition for the first values of n:

```
ACE> seq(qDdOnMp(i)-qSpec(SfAExpand(DdOnMp(i,1))),i=1..7);

                              0, 0, 0, 0, 0, 0, 0
```

Of course, `qDdOnMp(i)` is much faster than `qSpec(SfAExpand(DdOnMp(i,1)))`.

4.6.4 $\mathbb{N}$-specialization: generalized Stirling numbers

Let us study the $\mathbb{N}$-specialization of theorem **4.6.1**, *i.e.* what are the coefficients when $a_i = i$.

For $a_i = i$, the modified powers $x^{[n]} := \prod_{i=0}^{n-1}(x-i)$ are also called factorial polynomials. Theorem **4.6.1** may be rewritten:

$$x_1{}^{[n]}\partial^k = \sum_\lambda S_{n-|\lambda|-k}\left(\sum_{j=1}^{\ell(\lambda)} A_{\lambda_j} - A_{n-1}\right)\Phi(m_\lambda(x_1,..,x_{k+1})) \quad . \tag{4.12}$$

Definition 4.6.2 (Generalized Stirling numbers) *Let $\lambda = [\lambda_1, \lambda_2, .., \lambda_\ell]$ be a partition. We define the generalized Stirling numbers $[\lambda]_i^n$ by:*

$$\begin{aligned}
[\lambda]_n^{n+|\lambda|} &= 1 && n > 0,\\
[\lambda]_i^n &= 0 && i > n - |\lambda|,\\
[\lambda]_0^n &= 0 && \ell = 0,\\
&= [\lambda - 1^\ell]_{\ell-1}^{n-1} && otherwise \quad ,\\
[\lambda]_i^{n+1} &= [\lambda]_{i-1}^n - n[\lambda]_i^n && otherwise \quad .
\end{aligned} \tag{4.13}$$

```
GenStirling := proc(lambda, n, m)
  local
    wght, # weight of lambda...
    l,    # length of lambda...
    i;    # var for seq...
option remember;
  wght := SumIt(lambda);
  l    := nops(lambda);
  if m>(n-wght) then
    0;
  elif m=(n-wght) then
    1;
  elif m=0 then
    if l<=1 then
      0;
    else
      GenStirling(subs(0=NULL,[seq(lambda[i]-1,i=1..l)]), n-1, l-1);
    fi;
  else
    GenStirling(lambda, n-1, m-1) - (n-1)*GenStirling(lambda, n-1, m);
  fi;
end:
```

Note that $[\ ]_i^n$ are the classical Stirling numbers of first kind $\mathcal{S}_i^n$.

Proposition 4.6.2 *The coefficients in theorem* **4.6.1** *for $A = \{0, 1, 2, ...\}$ are the generalized Stirling numbers:*

$$x_1{}^{[n]}\partial^k = \sum_\lambda [\lambda]_k^n \ \Phi(m_\lambda(x_1,..,x_{k+1})) \quad . \tag{4.14}$$

Let us again verify the proposition, using `SFA`:

```
# N-specialization
NSpec := proc(f) local i; subs(seq(a.i=i, i=0..100), f) end:

NDdOnMp := proc(n,k)
  local
    lp, # list of partitions...
    i;  # var for seq...
  lp := Flat([seq(ListPart(i), i=0..n-k)]);
  SumIt([seq(
   GenStirling(lp[i],n,k)*Phi(m[op(lp[i])](Alphab('x',k+1))),i=1..nops(lp))]);
end:
```

And we have:

```
ACE> seq(seq(NDdOnMp(i,j)-NSpec(SfAExpand(DdOnMp(i,j))),i=1..6),j=1..3);

              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

5

Main routines

The reasonable man
adapts himself to the world;
the unreasonable one
persists in trying to adapt the world to himself.
Therefore all progress depends on the unreasonable man.

George Bernard Shaw.

This chapter describes the main routines of the Algebraic Combinatorics Environment. Each section is devoted to a package that defines an entire collection of functions dealing with a specific subject such as the symmetric group, symmetric functions, noncommutative symmetric functions, Schubert polynomials, hyperoctahedral groups, algebras related to hyperoctahedral groups, classical Lie groups, computations in the ring of polynomials in several variables, ...

Each function description has the following format: after a one-line description showing how the function can be called, we find informations about the expected arguments. Then, we give a brief exposition about the function, and finally we provide some examples of the use of the function. Some related functions or topics finish the description.

In each description, names in a `typewriter font` as for instance in the **Calling Sequence** section, are parameters which are to be used verbatim, while *italicized* names are prototypes that may be replaced by any *Maple* expression according to the type specification noted in the **Parameters** section. The optional parameters are so-called extra parameters and may therefore be omitted when the function is called.

The first section of this chapter is devoted to present the `ACE` package. It includes all other packages so that it is suffcient to load `ACE` to have access to any function of the "*environment*". Note that all *Maple* functions are still available after having loaded the `ACE` package.

Note also that the notion of package is a very powerful *Maple* concept but we will talk about an "environment" (in algebraic combinatorics) to refer to the `ACE` package since it clusters various packages dealing with specific objects, all related to algebraic combinatorics. Thus, it sounds a little bit more general than a simple package.

5.1 About the ACE package

HELP FOR: An environment for Algebraic Combinatorics.

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 structures in algebraic combinatorics (partitions, compositions, permutations, words, tableaux, divided differences, symmetric functions), to compute in several algebras related to the symmetric group and to the hyperoctahedral group. It also gives functions to handle characters and representations of several classical groups, and to manipulate more recent objects like Schubert polynomials and non commutative symmetric functions.

- The available packages are:

BN	BNA	CG	CLG
COMP	FM	FREE	HEKA
IDCA	NCA	NCSF	PART
SFA	SG	SGA	SP
SYMF	TAB	TEX	TYP

- Another package ERR is used to print errors.

- Caution, the names A, B, E, F, L, M, O, Pe, Ph, Ps, QPh, QPs, R, S, Sp, X, XX, Y, YY, c, cl, e, h, m, p, s, w, x, y must not have any value assigned to have ACE work properly. These symbols are used as formal names in ACE, to denote either (noncommutative) symmetric functions, Schubert polynomials, or bases of different algebras and rings.

- 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>].

- ACE package available functions are:

AceBanner	AceBuildData	AceLoadData	AceSysInfo
AceTime	AceUnloadData	AceWhichIs	

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

EXAMPLES:

```
> with(ACE):
> AceWhichIs(Tos, 'str');

                                    true

> str;

                                 SYMF[Tos]

> AceSysInfo();

number of CPU seconds used     : 5.65
total number of words requested: 1000012
total number of words allocated: 851929
number of garbage collection   : 1
```

FUNCTION: `AceBanner` - print a banner about ACE

CALLING SEQUENCE:

```
AceBanner()
ACE[AceBanner]()
```

PARAMETERS:

SYNOPSIS:

- The `AceBanner` function prints a banner about `ACE`. It includes the release, the `ACE`'s address, the email address, the Web URL, and the list of all packages available with `ACE`.
- Whenever there is a conflict between the function name `AceBanner` and another name used in the same session, use the long form `ACE['AceBanner']`.

EXAMPLES:

```
> with(ACE):
> AceBanner();
############################################################
# Algebraic Combinatorics Environment,  3.0,  June 2, 1998 #
############################################################
# Alain Lascoux, Sebastien Veigneau & Co                   #
# Institut Gaspard Monge, Universite de Marne-la-Vallee,   #
# Cite Descartes, 5 Boulevard Descartes, Champs-sur-Marne, #
# 77454, Marne-la-Vallee, Cedex 2, FRANCE.                 #
# Email: ACE@univ-mlv.fr                                   #
# URL: http://phalanstere.univ-mlv.fr/~ace                 #
####################################################
# ACE  Algebraic Combinatorics Environment         #
#                                                  #
```

```
# SFA  Symmetric Functions on different Alphabets #
# FM   Free Module over symmetric polynomials   #
# NCSF NonCommutative Symmetric  Functions    #
# CLG  Characters of Linear Groups          #
# SP   Schubert Polynomials               #
# SYMF SYMmetric Functions              #  __/__/ __/_/_/ __/_/__/
# TAB  TABleaux                       #  __/ __/ __/     __/
# FREE FREE algebra                 #  __/_/__/ __/     __/_/
# BNA  hyperoctahedral Algebras   #  __/   __/ __/     __/
# SGA  Symmetric Group Algebra  #  __/    __/ __/_/_/ __/_/__/
# NCA  NilCoxeter Algebra     #
# IDCA IDCoxeter Algebra     #               _/_/_/    _/_/_/
# HEKA HEcKe Algebra          #               _/_/    _/  _/
#                               #           _/_/_/ _/ _/_/_/
# BN   hyperoctahedral groups    #
# SG   Symmetric Groups         #
# CG   Classical Groups       #
#                           #
# PART PARTitions         #
# COMP COMPositions      #
#                         #
# ERR  ERRors              #
# TEX  TEX output         #
# TYP  TYPe checking    #
########################
```

See Also:

Function: `AceBuildData` - build and store data

Calling Sequence:

```
AceBuildData(pack, data)
ACE[AceBuildData](pack, data)
```

Parameters:

`pack` = any name of package
`data` = any sequence of names

Synopsis:

- The `AceBuildData` function is used to build and store `data` on the disk. The first argument `pack` specifies the name of the package or more precisely the name of the sub-directory that contains the `data` on the disk. The second argument is in fact a sequence of arguments that gives the name of the `data` to be read on the disk.
- The `pack` name should be one of the list of package names `_ACELPN`.
- It returns true when it is able to store the computed `data` on the disk and false otherwise: in case of such a problem, the `data` which has been computed is usable but it (because it is not saved on the disk) should be recomputed each session.
- See `ACE['AceLoadData']` to see the list of `data` you can build.

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

Examples:

```
> with(ACE):
> AceBuildData(FM, XmXfix, 2);
```

```
                                    true
```

See Also: `AceLoadData` `AceUnloadData`

Function: `AceLoadData` - load data from disk

Calling Sequence:

```
AceLoadData(pack, data)
ACE[AceLoadData](pack, data)
```

Parameters:

`pack` = any name of package
`data` = any sequence of names

Synopsis:

- The `AceLoadData` function is used to load data from the disk. The first argument `pack` specifies the name of the package or more precisely the name of the sub-directory that contains the data on the disk. The second argument is in fact a sequence of arguments that specifies the name of the data to be read on the disk.
- The `pack` name should be one of the list of package names `_ACELPN`.
- Whenever the corresponding file does not exist on the disk, then it runs the `ACE['AceBuildData']` command to build the data and to store it on the disk.
- It returns true when the data is loaded and false otherwise, that is for instance when there is a problem to store the data built by `AceBuildData` on the disk: in case of such a problem, the data has been computed and is usable but it should be recomputed each session (because it has not been saved on the disk).
- It does not reload the data if it is already loaded: it then returns true.
- Here is the list of data you can load together with the corresponding command to load it:

  o Multiplication table between Schubert polynomials in a given symmetric group $\mathfrak{S}_n$. The result is expressed in the free module over symmetric polynomials in n variables (`FM` package). For instance, the command `AceLoadData(FM, XmXfix, 4);` is used to load the multiplication table for the symmetric group of degree 4: this table is `'FM/data/XmXfix4'`.

  o Multiplication table between Schubert polynomials in a given symmetric group $\mathfrak{S}_n$ and all variables $x_i, i = 1..n$. The result is expressed in the free module over symmetric polynomials in n variables (`FM` package). For instance, the command `AceLoadData(FM, MXfix, 4);` is

used to load the multiplication table for the symmetric group of degree 4: this table is then ‘FM/data/XMfix4‘.

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

Examples:

```
> with(ACE):
> AceLoadData(FM, XmXfix, 3);
```

```
                                    true
```

See Also: AceBuildData AceUnloadData

Function: AceSysInfo - give information on the system

Calling Sequence:

```
AceSysInfo()
ACE[AceSysInfo]()
```

Parameters:

Synopsis:

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

Examples:

```
> with(ACE):
> Tos(s[3,2,1]^2):
> AceSysInfo();
```

```
number of CPU seconds used     : 0.683
total number of words requested: 255616
total number of words allocated: 147429
number of garbage collections  : 1
```

See Also:

Function: `AceTime` - give time information

Calling Sequence:

```
AceTime(f, 't', a_1, ..., a_k)
ACE[AceTime](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 `AceTime` function runs $f(a_1, ..., a_k)$ and saves in the parameter `'t'` the computation time.
- The value of `AceTime(f, 't', a_1,..., a_k)` is $f(a_1, ..., a_k)$.
- For composite functions, transform it into a single function in order to use `AceTime`.
- Whenever there is a conflict between the function name `AceTime` and another name used in the same session, use the long form `ACE['AceTime']`.

Examples:

```
> with(ACE):
> AceTime(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;

                                  .266

> scal:=proc(p1,p2,p3) SfScalar(m[op(p1)]*s[op(p2)],m[op(p3)]) end:
> AceTime(scal, 't', [2,1], [3,1,1], [2]);

                                   0

> t;

                                  .167
```

See Also:

Function: `AceUnloadData` - unload data

Calling Sequence:

```
AceUnloadData(pack, data)
ACE[AceUnloadData](pack, data)
```

Parameters:

`pack` = any name of package
`data` = any sequence of names

Synopsis:

- The `AceUnloadData` function is used to unload `data` that is to forget the corresponding `data`, during the current session. Refer to the function `AceLoadData` to see the list of possible `data` to unload.
- The function returns true if the `data` was loaded before you run the `AceUnloadData` command and false otherwise.
- Note that the file (if it exists) that stores the corresponding `data` on disk is not removed.
- Whenever there is a conflict between the function name `AceUnloadData` and another name used in the same session, use the long form `ACE['AceUnloadData']`.

Examples:

```
> with(ACE):
> AceUnloadData(FM, XmXfix, 3);
```

```
                                false
```

See Also: `AceBuildData` `AceLoadData`

Function: `AceWhichIs` - look for a function in a list of packages

Calling Sequence:

```
AceWhichIs(f)
AceWhichIs(f, 'r')
ACE[AceWhichIs](f)
ACE[AceWhichIs](f, 'r')
```

Parameters:

`f` = any name of function
`r` = any name of variable (formal parameter)

SYNOPSIS:

- The `AceWhichIs` function looks for a function whose name is `f`. It returns true in the case it finds such a function and false otherwise. It looks for the name `f` among all functions of all packages contained in the list `_ACELPN`.
- When called with a second formal parameter, say `'r'`, `r` contains the long form name of the first function whose name matches the name `f` (this is the case when `AceWhichIs` returns true). When `AceWhichIs` returns false, `r` contains the list of functions whose names contain the substring `f`.
- Whenever there is a conflict between the function name `AceWhichIs` and another name used in the same session, use the long form `ACE['AceWhichIs']`.

EXAMPLES:

```
> with(ACE):
> AceWhichIs(NewtonInterp, 'n');

                                    true

> n;

                              SP[NewtonInterp]

> AceWhichIs(On, 'l');

                                   false

> l;

   [BNA[BnNcaOnPol], BNA[BnGaOnPol], BNA[BnIdcaOnPol], FREE[SgaOnFree],

    FREE[IdcaOnFree], HEKA[HekaOnXfix], HEKA[HekaOnPol], HEKA[HekaOnX],

    IDCA[IdcaOnPol],  IDCA[IdcaOnX],   IDCA[IdcaOnXfix],   NCA[NcaOnY],

    NCA[NcaOnYY],    NCA[NcaOnXX],     NCA[NcaOnX],      NCA[NcaOnPol],

    NCA[NcaOnXfix],     SGA[SgaOnXfix],   SGA[SgaOnPol],   SGA[SgaOnX],

    TAB[PermOnCTab], TAB[PermOnTab]]
```

SEE ALSO:

5.2 PART, a package to handle partitions

Help For: A package for partitions.

Calling Sequence:
```
PART[<function>](args)
<function>(args)
```

Synopsis:

- This package provides functions to work with partitions. Available functions deal with border, conjugate, diagonals, Frobenius notation, hooks, etc. of a partition, and also with lists of (skew) partitions.
- Whenever there is a conflict between a function name in `PART` and another name used in the same session, use the long form `PART`[<function>].
- The available functions are:

Border2Part	CompPart	Diagonal2Part	Exp2Part
Frob2Part	ListPart	ListPartIn	ListSkewDiag
PCore2Part	Part2Border	Part2Conjugate	Part2Diagonal
Part2Exp	Part2Frob	Part2ListHook	Part2Mat
Part2PCore	PartOrderMat	SkewPart2Mat	SplitPart

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

Examples:

```
> with(PART):
> Part2ListHook([4,2,2,1]);

                        [7, 5, 4, 3, 2, 2, 1, 1, 1]

> ListPartIn([40,30,15,10,8,7,5,3,1], 'nb');

                                  9631414
```

Function: `Border2Part` - partition associated to a given border

Calling Sequence:
```
Border2Part(border)
PART[Border2Part](border)
```

PARAMETERS:
`border` = a border

SYNOPSIS:

- The `Border2Part` function gives the partition associated to the `border`.
- A border delimits the diagram of a partition.
- The `border` is a list of 0 and 1, 0 denoting an horizontal step of the border of the diagram, 1 a vertical one.
- The following options are available:
  - `'alphabet'` = `m` where `m` is a list of length 2, returns the same, `m[1]` denoting horizontal steps, `m[2]` vertical ones.
- Whenever there is a conflict between the function name `Border2Part` and another name used in the same session, use the long form `PART['Border2Part']`.

EXAMPLES:

```
> with(PART):
> Border2Part([0,1,0,1,0,0,1,1,0,0,1]);

                              [6, 4, 4, 2, 1]

> Border2Part([a,b,a,b,a,a,b,b,a,a,b], 'alphabet'=[a,b]);

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

SEE ALSO: `Part2Border` `Part2Diagonal` `Part2Conjugate` `Part2Mat`

FUNCTION: `CompPart` - compare two partitions

CALLING SEQUENCE:
```
CompPart(part1, part2)
CompPart(part1, part2, kind)
PART[CompPart](part1, part2)
PART[CompPart](part1, part2, kind)
```

PARAMETERS:
`part1, part2` = any partitions
`kind` = `'natural'`, `'lexic'`, `'cixel'`

SYNOPSIS:

- The `CompPart` function compares two partitions for the given ordering.
- The considered ordering is given by the second argument `kind`:

  - `'natural'` : `part1` $\geq$ `part2` if $\sum_{\texttt{part1}} \leq \sum_{\texttt{part2}}$ for the lexicographic ordering ($\sum_I = [I_1, I_1+I_2, I_1+I_2+I_3, ...]$).
  - `'lexic'` : `part1`$\geq$`part2` for the lexicographic ordering.
  - `'cixel'` : `part1`$\geq$`part2` for the inverse lexicographic ordering.

- Without a third argument, the comparison is assumed to be natural.
- Whenever there is a conflict between the function name `CompPart` and another name used in the same session, use the long form `PART['CompPart']`.

Examples:

```
> with(PART):
> CompPart([3,2,1], [3,2,1,1], 'cixel');

                                    true

> CompPart([3,2,1], [3,2,1,1], 'lexic');

                                   false

> CompPart([3,2,1], [3,2,1,1]);

                                    true
```

See Also: `PartOrderMat` `ListPart`

Function: `Diagonal2Part` - partition from a list of diagonals

Calling Sequence:

```
Diagonal2Part(diago)
PART[Diagonal2Part](diago)
```

Parameters:

`diago` = a list with two components

Synopsis:

- The `Diagonal2Part` function takes a list `[v,pos]` as argument, where `v` is a list (lengths of the successive diagonals of the diagram of `part`) and `pos` an integer (position of the main diagonal), and returns the partition `part` such that `Part2Diagonal(part) = [v,pos]`.
- Whenever there is a conflict between the function name `Diagonal2Part` and another name used in the same session, use the long form `PART['Diagonal2Part']`.

Examples:

```
> with(PART):
> Diagonal2Part([[1,1,2,2,3,3,2,1,1,1], 5]);

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

See Also: Part2Diagonal Part2Border Part2Conjugate Part2Mat

Function: Exp2Part - partition from its exponential notation

Calling Sequence:

```
Exp2Part(vect)
PART[Exp2Part](vect)
```

Parameters:

vect = a vector of positive integers

Synopsis:

- The Exp2Part is the inverse function of Part2Exp.
- The exponential notation of a partition is a vector of positive integers that indicates the number of occurrences a part appears in the partition.
- Whenever there is a conflict between the function name Exp2Part and another name used in the same session, use the long form PART['Exp2Part'].

Examples:

```
> with(PART):
> Exp2Part([1, 0, 4, 0, 2]);

                          [5, 5, 3, 3, 3, 3, 1]

> Part2Exp([5, 5, 3, 3, 3, 3, 1]);

                              [1, 0, 4, 0, 2]
```

See Also: Part2Exp Part2Border Part2Diagonal Part2Conjugate

Function: Frob2Part - from Frobenius notation to standard notation for partitions

Calling Sequence:

```
Frob2Part(frob)
PART[Frob2Part](frob)
```

Parameters:

frob = a list of two decreasing lists of the same length

SYNOPSIS:

- The `Frob2Part` function computes a partition from its Frobenius symbol; that is the decomposition of its diagram into diagonal hooks. The inverse function is `Part2Frob`.
- The input is a list of two lists (list of diagonal arms, list of diagonal legs).
- The result is a list denoting a partition.
- Whenever there is a conflict between the function name `Frob2Part` and another name used in the same session, use the long form `PART['Frob2Part']`.

EXAMPLES:

```
> with(PART):
> Frob2Part([[6,3,0], [4,2,1]]);

                          [7, 5, 3, 3, 1]
```

SEE ALSO: `Part2Frob` `Part2Border` `Part2Exp` `Part2Mat` `Part2PCore`

FUNCTION: `ListPart` - list of partitions of a given weight

CALLING SEQUENCE:

```
ListPart(n)
PART[ListPart](n)
```

PARAMETERS:

`n` = any non negative integer

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`.
- The following options are available:
  - `'maxlg' = l` sets the maximal number of parts of each partition.
  - `'maxpart' = p` enumerates only the partitions `pa` such that `pa[1]` $\leq$ `p[1]`, ..., `pa[nops(p)]` $\leq$ `p[nops(p)]`.
  - `'regular' = m` only `m`-regular partitions are listed (no part is repeated strictly more than `m` times).
  - `'nb'` returns the number of partitions instead of the list thereof.
- 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 `PART['ListPart']`.

EXAMPLES:

```
> with(PART):
> ListPart(4);

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

> ListPart(5, 'maxlg'=3);

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

> ListPart(5, 'maxpart'=[3,2], 'maxlg'=3);
                         [[3, 2], [3, 1, 1], [2, 2, 1]]

> ListPart(100, 'nb');
                                    190569292
```

SEE ALSO: `ListPartIn` `ListSkewDiag`

FUNCTION: `ListPartIn` - list of partitions contained into a given partition or between two given partitions

CALLING SEQUENCE:

```
ListPartIn(part)
ListPartIn(part, part_in)
PART[ListPartIn](part)
PART[ListPartIn](part, part_in)
```

PARAMETERS:

`part, part_in` = any lists denoting partitions

SYNOPSIS:

- The `ListPartIn` function gives all partitions contained into the diagram of a given partition. The list is ordered by increasing length.
- Called with a second argument `part_in`, the function returns the list of all partitions contained into the diagram of `part` and containing the diagram of `part_in`.
- The following options are available:
  - `'weight' = w` imposes the weight of partitions.
  - `'nb'` returns the number of partitions instead of the list thereof.
- Whenever there is a conflict between the function name `ListPartIn` and another name used in the same session, use the long form `PART['ListPartIn']`.

EXAMPLES:

```
> with(PART):
> ListPartIn([3,1]);

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

> ListPartIn([3,2,1], [2,1,1]);

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

> ListPartIn([4,3,2], 'weight'=6);

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

> ListPartIn([40,30,15,10,8,7,5,3,1], 'nb');

                                    9631414
```

SEE ALSO: `ListPart` `ListSkewDiag`

FUNCTION: `ListSkewDiag` - list of skew diagrams

CALLING SEQUENCE:

```
ListSkewDiag(n)
PART[ListSkewDiag](n)
```

PARAMETERS:

`n` = any non negative integer

SYNOPSIS:

- The `ListSkewDiag` function gives all skew diagrams of `n`. Skew diagrams are special cases of skew partitions.
- A skew partition of `n` is a list of two partitions `part1` and `part2` such that `part1` is greater than `part2` (with respect to inclusion of diagrams) and `n` equals (weight of `part1`) minus (weight of `part2`).
- A skew diagram is a skew partition without empty rows or columns.
- When called with one argument, say `n`, the function returns the list of all skew diagrams of weight `n`.
- You may set some constraints on skew diagrams by using one or several options:
  - `'maxlg'=l` majorates the length of skew diagrams,
  - `'mininner'= p` imposes the minimal inner partition,
  - `'maxinner'= p` imposes the maximal inner partition,
  - `'minouter'= p` imposes the minimal outer partition,

  - 'maxouter'= p imposes the maximal outer partition,
  - 'nb' returns the number of skew diagrams instead of the list thereof.

- The outer partition corresponds to the outer shape of the skew diagram and the inner partition corresponds to the inner shape of the skew diagram.
- Given a skew diagram skd, convert(op(1,skd),`+`) - convert(op(2,skd),`+`) gives its weight.
- Whenever there is a conflict between the function name ListSkewDiag and another name used in the same session, use the long form PART['ListSkewDiag'].

EXAMPLES:

```
> with(PART):
> ListSkewDiag(2);

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

> ListSkewDiag(3, 'mininner'=[1], 'maxouter'=[3,1,1]);

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

> ListSkewDiag(3, 'maxinner'=[1], 'minouter'=[3]);

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

> ListSkewDiag(10, 'maxlg'=5, 'nb');

                                       5309
```

SEE ALSO: ListPart ListPartIn SkewPart2Mat

FUNCTION: PCore2Part - partition from p-core and p-quotient

CALLING SEQUENCE:

```
PCore2Part(pcore)
PART[PCore2Part](pcore)
```

PARAMETERS:

pcore = a list of lists

SYNOPSIS:

- The function PCore2Part is the inverse function of Part2PCore.
- pcore is a list [c, v0, v1, ..., vp-1]: c is the p-core (obtained by removing all possible outside ribbons of length p from the diagram of a partition; v0, v1, ..., vp-1 are the partitions obtained by grouping part[1], part[2]-1, part[3]-2, ..., according to their classes modulo p.

- Whenever there is a conflict between the function name `PCore2Part` and another name used in the same session, use the long form `PART['PCore2Part']`.

EXAMPLES:

```
> with(PART):
> Part2PCore([5,5,5,4,3,3,2,1,1,1],3);

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

> PCore2Part([[2, 2, 1, 1], [1, 1, 1], [1], [3, 1]]);

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

SEE ALSO: `Part2PCore`

FUNCTION: `Part2Border` - border of a given partition

CALLING SEQUENCE:

```
Part2Border(part)
PART[Part2Border](part)
```

PARAMETERS:

`part` = a partition

SYNOPSIS:

- The `Part2Border` function gives the north-east border of the diagram of a partition `part`.
- The border is a list of 0 (horizontal steps) and 1 (vertical steps).
- The following option is available:
  - `'alphabet' = m` where `m` is a list of length 2, returns the same, `m[1]` denoting horizontal steps, `m[2]` vertical ones.
- Whenever there is a conflict between the function name `Part2Border` and another name used in the same session, use the long form `PART['Part2Border']`.

EXAMPLES:

```
> with(PART):
> Part2Border([6,4,4,2,1]);

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

> Part2Border([6,4,4,2,1], 'alphabet'=[a,b]);

                    [a, b, a, b, a, a, b, b, a, a, b]
```

See Also: `Border2Part` `Part2Diagonal` `Part2Conjugate` `Part2Mat`

Function: `Part2Conjugate` - conjugate a partition

Calling Sequence:

```
Part2Conjugate(part)
PART[Part2Conjugate](part)
```

Parameters:

`part` = any list denoting a partition in (weakly) decreasing order

Synopsis:

- The `Part2Conjugate` function gives the conjugate (or transposed) of a partition. Geometrically, conjugation amounts to reflecting the diagram of the partition with respect to the main diagonal.
- Whenever there is a conflict between the function name `Part2Conjugate` and another name used in the same session, use the long form `PART['Part2Conjugate']`.

Examples:

```
> with(PART):
> Part2Conjugate([4,2,2]);
```

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

See Also: `Part2ListHook` `Part2Border` `Part2Diagonal`

Function: `Part2Diagonal` - diagonals of the given partition

Calling Sequence:

```
Part2Diagonal(part)
PART[Part2Diagonal](part)
```

Parameters:

`part` = a partition

Synopsis:

- The `Part2Diagonal` function gives a list composed of a list `v` and an integer `pos`; `v` is the list of lengths of the successive diagonals of the diagram of `part`; `pos` gives the position of the main diagonal in `v`.
- Whenever there is a conflict between the function name `Part2Diagonal` and another name used in the same session, use the long form `PART['Part2Diagonal']`.

EXAMPLES:

```
> with(PART):
> Part2Diagonal([6,4,4,2,1]);

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

SEE ALSO: `Diagonal2Part` `Part2Border` `Part2Conjugate` `Part2Mat`

FUNCTION: `Part2Exp` - multiplicities of a given partition

CALLING SEQUENCE:

```
Part2Exp(part)
PART[Part2Exp](part)
Part2Exp(part,m)
PART[Part2Exp](part,m)
```

PARAMETERS:

`part` = any partition
`m` = any positive integer

SYNOPSIS:

- The `Part2Exp` function gives the multiplicities of the parts of the partition `part`.
- The multiplicities of a partition is the list of occurrences of positive integers in the partition, starting from 1 up to the maximal part.
- When called with a second argument `m¿nops(part)`, it returns the list of occurences of 1, 2, ..., `m`.
- Whenever there is a conflict between the function name `Part2Exp` and another name used in the same session, use the long form `PART['Part2Exp']`.

EXAMPLES:

```
> with(PART):
> Part2Exp([6,4,4,2,1]);

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

> Part2Exp([6,4,4,2,1], 10);

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

See Also: Exp2Part Part2Border Part2Diagonal Part2Conjugate

Function: Part2Frob - from standard partition notation to Frobenius notation

Calling Sequence:
Part2Frob(part)
PART[Part2Frob](part)

Parameters:
part = any list denoting a partition

Synopsis:

- The Part2Frob function converts a partition into its Frobenius notation, which codes the decomposition of its diagram into diagonal hooks.
- The result is a list of two lists (list of diagonal arms, list of diagonal legs).
- Whenever there is a conflict between the function name Part2Frob and another name used in the same session, use the long form PART['Part2Frob'].

Examples:

```
> with(PART):
> Part2Frob([7,5,3,3,1]);
```

```
                          [[6, 3, 0], [4, 2, 1]]
```

See Also: Frob2Part Part2Border Part2Exp Part2Mat Part2PCore

Function: Part2ListHook - calculate the list of hook lengths

Calling Sequence:
Part2ListHook(part)
PART[Part2ListHook](part)

Parameters:
part = any list denoting a partition

Synopsis:

- The Part2ListHook function computes the list of all hook lengths of a given partition. The result is sorted into decreasing 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 `Part2ListHook(part,a)` computes the Jack-hook-product. `a` may be any expression. This is the product:
$$\texttt{a} * \texttt{arm_length}[\texttt{i},\texttt{j}] + \texttt{leg_length}[\texttt{i},\texttt{j}] + 1$$
over all cells of the diagram of the partition `part`.
- The call `Part2ListHook(part, q, t)` computes the two-variable hook polymial. Arguments `q` and `t` can be any expressions. This is the product:
$$1 - q^{\texttt{arm_length}[\texttt{i},\texttt{j}]} * t^{(1+\texttt{leg_length}[\texttt{i},\texttt{j}])}$$
over all cells of the diagram of the partition `part`.
- Whenever there is a conflict between the function name `Part2ListHook` and another name used in the same session, use the long form `PART['Part2ListHook']`.

EXAMPLES:

```
> with(PART):
> Part2ListHook([3,1]);
```

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

```
> 4!/convert([4, 2, 1, 1], `*`);
```

```
                                   3
```

```
> Part2ListHook([4,1,1], 'z');
```

```
                       2 (3 z + 3) (2 z + 1) (z + 1)
```

```
> Part2ListHook([4,1,1], 'q', 't');
```

```
                  3  3        2                    2       2
            (1 - q  t ) (1 - q  t) (1 - q t) (1 - t)  (1 - t )
```

SEE ALSO: `Part2Conjugate` `Part2Border` `Part2Diagonal` `Part2Mat` `ListPart`

FUNCTION: `Part2Mat` - a planar representation of a given partition

CALLING SEQUENCE:

```
Part2Mat(part)
PART[Part2Mat](part)
```

PARAMETERS:

`part` = a partition

SYNOPSIS:

- The `Part2Mat` function gives a planar representation of a partition, (called diagram of the partition) in the Cartesian plane.

- A partition is a decreasing list of positive integers.
- The following option is allowed:
  - `'alphabet' = l` where `l` is a list of 1 or 2 components, `l[1]` being the partition component, `l[2]` the outer component.
- Whenever there is a conflict between the function name `Part2Mat` and another name used in the same session, use the long form `PART['Part2Mat']`.

EXAMPLES:

```
> with(PART):
> Part2Mat([4,2,2]);
```

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

```
> Part2Mat([4,2,2], 'alphabet'=[`#`,` `]);
```

$$\begin{bmatrix} \# & \# & & \\ & & & \\ \# & \# & & \\ & & & \\ \# & \# & \# & \# \end{bmatrix}$$

SEE ALSO: `Part2Border` `Part2Conjugate` `Part2Diagonal` `ListPart`

FUNCTION: `Part2PCore` - p-core and p-quotient of a partition

CALLING SEQUENCE:

```
Part2PCore(part, p)
PART[Part2PCore](part, p)
```

PARAMETERS:

`part` = a partition
`p` = an integer

SYNOPSIS:

- To a partition `part` and an integer `p`, the function `Part2PCore` associates a list `[c, v0, v1, ..., vp-1]`: `c` is the `p`-core (obtained by removing all possible outside ribbons of length `p` from the diagram of `part`; `v0, v1, ..., vp-1` are the partitions obtained by sorting `part[1] part[2]-1, part[3]-2, ...`, according to their classes modulo `p`.
- Whenever there is a conflict between the function name `Part2PCore` and another name used in the same session, use the long form `PART['Part2PCore']`.

EXAMPLES:

```
> with(PART):
> Part2PCore([5,5,5,4,3,3,2,1,1,1],3);

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

SEE ALSO: `PCore2Part`

FUNCTION: `PartOrderMat` - order matrix on partitions

CALLING SEQUENCE:

```
PartOrderMat(n)
PartOrderMat(n, kind)
PART[PartOrderMat](n)
PART[PartOrderMat](n, kind)
```

PARAMETERS:

`n` = any positive integer
`kind` = `'natural'`, `'lexic'`, `'cixel'`

SYNOPSIS:

- The `PartOrderMat` function gives the order matrix on all partitions of `ListPart(n)`.
- The considered order is given by the second argument `kind`:
  - `'natural'` : `part1`$\geq$`part2` if $\sum_{\texttt{part1}} \leq \sum_{\texttt{part2}}$ for the lexicographic ordering ($\sum_I = [I_1, I_1 + I_2, I_1 + I_2 + I_3, ...]$).
  - `'lexic'` : `part1`$\geq$`part2` for the lexicographic ordering.
  - `'cixel'` : `part1`$\geq$`part2` for the inverse lexicographic ordering.
- The order is by default the natural order. The inverse matrix for this order is known as the Moebius matrix.
- Whenever there is a conflict between the function name `PartOrderMat` and another name used in the same session, use the long form `PART['PartOrderMat']`.

EXAMPLES:

```
> with(PART):
> PartOrderMat(6);

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

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

```
> linalg[inverse](PartOrderMat(6));
```

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

```
> PartOrderMat(4, 'cixel');
```

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

SEE ALSO: ListPart CompPart

FUNCTION: `SkewPart2Mat` - a planar representation of a skew partition

CALLING SEQUENCE:

```
SkewPart2Mat(skp)
PART[SkewPart2Mat](skp)
```

PARAMETERS:

`skp` = a skew part

SYNOPSIS:

- The `SkewPart2Mat` function gives a planar representation of a skew partition.
- A skew partition is a list of two partitions `p1` and `p2` such that `p1` is greater than `p2` (with respect to inclusion of diagrams).
- The following option is allowed:
  - `'alphabet' = l` where `l` is a list of 1 to 3 components, `l[1]` filling the skew diagram, `l[2]` the inner diagram and `l[3]` the outside.
- Whenever there is a conflict between the function name `SkewPart2Mat` and another name used in the same session, use the long form `PART['SkewPart2Mat']`.

EXAMPLES:

```
> with(PART):
> SkewPart2Mat([[3,2],[2]]);
```

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

```
> SkewPart2Mat([[3,2],[2]], 'alphabet'=[`#`,`.`,` `]);
```

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

```
> map(SkewPart2Mat, ListSkewDiag(3));
```

$$\left[\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ & & \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 0 \\ & & \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ & & \\ 0 & 1 & 0 \\ & & \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ & \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ & \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ & \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ & \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix}\right]$$

See Also: ListSkewDiag Part2Mat

Function: SplitPart - split a partition into several ones

Calling Sequence:
SplitPart(part)
PART[SplitPart](part)
SplitPart(part,n)
PART[SplitPart](part,n)

Parameters:
part = a partition
n = any positive integer

Synopsis:

- The SplitPart function returns the set of ways of splitting the partition part into two partitions.
- Called with a second argument n, the partition is splitted into n partitions.
- Whenever there is a conflict between the function name SplitPart and another name used in the same session, use the long form PART['SplitPart'].

Examples:

```
> with(PART):
> SplitPart([4,2,1]);

      {[[2], [4, 1]], [[4], [2, 1]], [[], [4, 2, 1]], [[4, 2, 1], []],

          [[2, 1], [4]], [[4, 1], [2]], [[1], [4, 2]], [[4, 2], [1]]}

> r := SplitPart([2,1], 3);

 r := {[[2, 1], [], []], [[1], [2], []], [[1], [], [2]], [[2], [1], []],

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

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

> map(proc(spt) sort(map(op, spt), proc(x,y) evalb(x>=y) end) end, r);

                                {[2, 1]}
```

See Also: ListSkewPart

5.3 COMP, a package to handle compositions

HELP FOR: A package for compositions.

CALLING SEQUENCE:
```
COMP[<function>](args)
<function>(args)
```

SYNOPSIS:

- This package provides functions to work with compositions. Available functions deal with comparisons, major indexes, conjugates, lists of compositions.
- Whenever there is a conflict between a function name in `COMP` and another name used in the same session, use the long form `COMP`[<function>].
- The available functions are:

`CompCompo`	`Compo2Conjugate`	`Compo2MajorIndex`	`Compo2Mat`
`ListCompo`	`ListCompoFatter`	`ListCompoFiner`	

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

EXAMPLES:

```
> with(COMP):
> Compo2Conjugate([1,4,1,1,2,2]);

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

> ListCompo(2, 'allowzeros', 'lg'=3);

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

FUNCTION: `CompCompo` - compare two compositions

CALLING SEQUENCE:
```
CompCompo(compo_1, compo_2, kind)
COMP[CompCompo](compo_1, compo_2, kind)
```

PARAMETERS:
`compo_1, compo_2` = any lists denoting compositions
`kind = 'lexic', 'cixel', 'finer'`

SYNOPSIS:

- The CompCompo function compares two compositions. The ordering is set by the argument kind, and the function returns true if compo_1≥compo_2 according to the chosen ordering.
- Decomposing parts of the first composition gives the list of 'finer' compositions.
- Whenever there is a conflict between the function name CompCompo and another name used in the same session, use the long form COMP['CompCompo'].

EXAMPLES:

```
> with(COMP):
> CompCompo([3,2,4], [3,1,1,4], 'finer');

                                    true

> CompCompo([3,1,1,4], [3,2,4], 'lexic');

                                   false

> CompCompo([3,2,4], [3,1,1,4], 'cixel');

                                    true
```

SEE ALSO: ListCompoFatter ListCompoFiner

FUNCTION: Compo2Conjugate - conjugate a composition

CALLING SEQUENCE:

```
Compo2Conjugate(comp)
COMP[Compo2Conjugate](comp)
```

PARAMETERS:

comp = any list denoting a composition

SYNOPSIS:

- The Compo2Conjugate function calculates the conjugate (or transposed) of a composition. Geometrically, a composition is represented by a ribbon (i.e. a skew diagram of width 1) and conjugation amounts to reflecting the diagram with respect to the main diagonal.
- Whenever there is a conflict between the function name Compo2Conjugate and another name used in the same session, use the long form COMP['Compo2Conjugate'].

EXAMPLES:

```
> with(COMP):
> Compo2Conjugate([1,4,1,1,2,2]);

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

See Also:

Function: `Compo2MajorIndex` - major index of a given composition

Calling Sequence:

```
Compo2MajorIndex(compo)
COMP[Compo2MajorIndex](compo)
```

Parameters:

`compo` = any list denoting a composition

Synopsis:

- The `Compo2MajorIndex` function computes the major index of a given composition. Let `I` $= [i_1, i_2, ..., i_m]$ be a composition, then `Compo2MajorIndex(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 `Compo2MajorIndex` and another name used in the same session, use the long form `COMP['Compo2MajorIndex']`.

Examples:

```
> with(COMP):
> Compo2MajorIndex([2,3,1,2]);
```

13

See Also:

Function: `Compo2Mat` - a planar representation of a composition

Calling Sequence:

```
Compo2Mat(compo)
COMP[Compo2Mat](compo)
```

Parameters:

`compo` = any list denoting a composition

Synopsis:

- The `Compo2Mat` function gives the diagram of a composition (i.e. a skew diagram of width 1, also called a ribbon), in the cartesian plane.
- A composition is a list of strictly positive integers.
- The following option is allowed:
  - `'alphabet' = l` where `l` is a list of 1 to 3 components, `l[1]` filling the skew diagram, `l[2]` the inner diagram and `l[3]` the outside.
- Whenever there is a conflict between the function name `Compo2Mat` and another name used in the same session, use the long form `COMP['Compo2Mat']`.

EXAMPLES:

```
> with(COMP):
> Compo2Mat([4,2,1,5]);
```

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

```
> Compo2Mat([4,2,1,5], 'alphabet'=[`#`,`.`,` `]);
```

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

SEE ALSO: `ListCompo`

FUNCTION: `ListCompo` - list of compositions of a given weight

CALLING SEQUENCE:

```
ListCompo(n)
COMP[ListCompo](n)
```

PARAMETERS:

`n` = any non negative integer

SYNOPSIS:

- The `ListCompo` function gives all compositions of `n`. A composition of `n` is list of positive integers, called parts, with sum `n`.
- When called with one argument, say `n`, the function returns the list of all compositions of `n`.
- The following options are available:
  - `'lg'=l` sets the number of parts of each composition.
  - `'lg'<=l` sets the maximal number of parts of each composition.
  - `'maxouter'=p` sets the outermost possible shape of the diagram (composition with possible zeros).
  - `'allowzeros'` compositions may include zeros.
  - `'nb'` returns the number of compositions instead of the list thereof.

- Given a composition comp, convert(comp, '+') gives its weight.
- Whenever there is a conflict between the function name ListCompo and another name used in the same session, use the long form COMP['ListCompo'].

EXAMPLES:

```
> with(COMP):
> ListCompo(3);

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

> ListCompo(2, 'allowzeros', 'lg'=3);

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

> ListCompo(5, 'maxouter'=[1,0,3], 'allowzeros', 'lg'<=4);

  [[1, 0, 3, 1], [1, 0, 2, 2], [1, 0, 1, 3], [1, 0, 0, 4], [0, 0, 3, 2],

               [0, 0, 2, 3], [0, 0, 1, 4], [0, 0, 0, 5]]

> ListCompo(20, 'lg'<=10, 'nb');

                                  262144
```

SEE ALSO: ListCompoFatter ListCompoFiner

FUNCTION: ListCompoFatter - all compositions fatter than a given one

CALLING SEQUENCE:

```
ListCompoFatter(compo)
COMP[ListCompoFatter](compo)
```

PARAMETERS:

compo = any list denoting a composition

SYNOPSIS:

- The ListCompoFatter function generates all compositions fatter than a a given composition compo (obtained by summing adjacent parts of compo). The result is a list of compositions.
- Whenever there is a conflict between the function name ListCompoFatter and another name used in the same session, use the long form COMP['ListCompoFatter'].

EXAMPLES:

```
> with(COMP):
> ListCompoFatter([2,1,3]);
```

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

See Also: ListCompo ListCompoFiner

Function: ListCompoFiner - all compositions finer that a given one

Calling Sequence:

```
ListCompoFiner(compo)
COMP[ListCompoFiner](compo)
```

Parameters:

compo = any list denoting a composition

Synopsis:

- The ListCompoFiner function generates all compositions finer than a given composition (obtained by decomposing parts of 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 COMP['ListCompoFiner'].

Examples:

```
> with(COMP):
> 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: CompCompo ListCompo ListCompoFatter

5.4 SG, a package for the symmetric group

HELP FOR: A package for the symmetric group.

CALLING SEQUENCE:
`SG[<function>](args)`
`<function>(args)`

SYNOPSIS:

- This package provides functions to work with the symmetric group.
- To get more information on permutations do either ?`TYP,Permutation` or ?`TYP[Permutation]`.
- 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:

Code2Perm	CompPerm	Cycle2Perm	Desc2ListPerm
FactorPerm	GenPerm	ListPerm	Mat2Perm
MultPerm	Part2SgCC	Perm2Betti	Perm2Code
Perm2Cycle	Perm2CycleType	Perm2Desc	Perm2Interval
Perm2Inv	Perm2Length	Perm2ListInv	Perm2ListRd
Perm2Mat	Perm2RRep	Perm2Rd	Perm2Rise
Perm2Rothe	Perm2Sign	RRep2Perm	RandPerm
Rd2Perm	SgCharTable	SgDimRep	SgTranspo
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]);
```

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

```
[                ]
[ x  .  .  .  .  .]
[                ]
[ .  .  x  .  .  .]
```

FUNCTION: `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 `[c1, c2, ...]` be the code of `perm`, then `ci` 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`

FUNCTION: `CompPerm` - compare two permutations

CALLING SEQUENCE:

```
CompPerm(perm_1, perm_2, kind)
SG[CompPerm](perm_1, perm_2, kind)
```

PARAMETERS:
`perm_1, perm_2` = any lists denoting permutations
`kind` = `'lexic', 'cixel', 'bruhat'`

SYNOPSIS:

- The `CompPerm` 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 from right to left.
- `kind='bruhat'` to compare the two permutations according to the Bruhat order.
- Whenever there is a conflict between the function name `CompPerm` and another name used in the same session, use the long form `SG['CompPerm']`.

EXAMPLES:

```
> with(SG):
> CompPerm([3,1,2], [1,3,2], 'lexic');

                                    true

> CompPerm([3,1,2], [1,3,2], 'cixel');

                                   false

> CompPerm([3,1,2], [1,3,2], 'bruhat');

                                    true
```

SEE ALSO:

FUNCTION: `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: Perm2Cycle Perm2CycleType Vect2Perm

FUNCTION: Desc2ListPerm - generate all permutations with a given descent set

CALLING SEQUENCE:

```
Desc2ListPerm(desc, n)
SG[Desc2ListPerm](desc, n)
```

PARAMETERS:

`desc` = any list of positive integer
`n` = any positive integer denoting the degree of a symmetric group

SYNOPSIS:

- The `Desc2ListPerm` function generates all permutations having a given descent set `desc`.
- The result is a list of permutations in lexicographic order.
- Whenever there is a conflict between the function name `Desc2ListPerm` and another name used in the same session, use the long form `SG['Desc2ListPerm']`.

EXAMPLES:

```
> with(SG):
> Desc2ListPerm([1,2,4], 5);

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

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

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

See Also: Perm2Desc

Function: 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:

Function: GenPerm - generate an element of the symmetric group

Calling Sequence:

```
GenPerm(n, typeperm)
GenPerm(perm, typeperm)
SG[GenPerm](n, typeperm)
SG[GenPerm](perm, typeperm)
```

Parameters:

n = any positive integer
perm = any list denoting a permutation
typeperm = 'first', 'previous', 'next', 'last'

Synopsis:

- The GenPerm function generates one element of a symmetric group in lexicographic order, according to the parameter typeperm.

- `GenPerm(n, 'first')` returns the first permutation of $\mathfrak{S}_n$.
- `GenPerm(n, 'last')` returns the last permutation of $\mathfrak{S}_n$.
- `GenPerm(perm, 'previous')` gives the previous permutation according to the lexicographic order. It returns the `NULL` value when `perm` is already the first permutation.
- `GenPerm(perm, 'next')` gives the next permutation according to the lexicographic order. It returns the `NULL` value when `perm` is already the last permutation.
- Whenever there is a conflict between the function name `GenPerm` and another name used in the same session, use the long form `SG['GenPerm']`.

EXAMPLES:

```
> with(SG):
> GenPerm([3, 1, 4, 2], 'next');

                              [3, 2, 1, 4]
> GenPerm([3, 2, 1, 4], 'previous');

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

SEE ALSO:

FUNCTION: `ListPerm` - generate all permutations

CALLING SEQUENCE:

```
ListPerm(n)
ListPerm(l)
SG[ListPerm](n)
SG[ListPerm](l)
```

PARAMETERS:

`n` = any positive integer denoting the degree of a symmetric group
`l` = any list or set of elements

SYNOPSIS:

- The `ListPerm` function generates or counts permutations.
- If the first argument is a list or a set `l`, it generates all distinct permutations of the elements of `l`.
- 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`.
- `ListPerm(n, 'lexic')` or `ListPerm(n, 'cixel')` returns all permutations in lexicographic or reverse lexicographic order.

- `ListPerm(n, 'level')` returns a list of lists. First list contains all permutations with 0 inversion, second one contains all permutations with 1 inversion, ..., last one contains all permutations with $\mathtt{n}(\mathtt{n}-1)/2$ inversions.
- `ListPerm(n, typerm)`, where `typerm` is either `'dominant'`, `'grassmannian'` or `'vexillary'`, returns the list of all dominant, grassmannian or vexillary permutations of the symmetric group of degree `n`. Vexillary permutations are generated using an algorithm due to J. West.
- When the first argument is an integer, one can use another argument `'code'`, to produce Lehmer codes instead of permutations. When generating dominant permutations, it is faster to produce codes. One can also add the argument `'nb'` to count instead of producing permutations.
- 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([2,2,4]);

                    [[2, 2, 4], [2, 4, 2], [4, 2, 2]]

> ListPerm(3, 'cixel');

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

> ListPerm(3, 'level');

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

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

> ListPerm(3, 'level', 'code');

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

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

> ListPerm(4, 'vexillary', 'nb');

                                23

> ListPerm(5, 'vexillary', 'nb', 'level');

                   [1, 4, 6, 11, 16, 18, 18, 15, 9, 4, 1]

> ListPerm(5, 'nb', 'level');

                   [1, 4, 9, 15, 20, 22, 20, 15, 9, 4, 1]
```

SEE ALSO: `GenPerm` `TYP[IsPerm]`

FUNCTION: 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]);
```

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

```
> Mat2Perm( Perm2Mat([2,4,3,1]) );
```

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

SEE ALSO: `Perm2Mat` `Perm2RRep` `RRep2Perm`

FUNCTION: 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]`

FUNCTION: `Part2SgCC` - compute a conjugacy class of the symmetric group

CALLING SEQUENCE:

```
Part2SgCC(part)
Part2SgCC(part, 'card')
SG[Part2SgCC](part)
SG[Part2SgCC](part, 'card')
```

PARAMETERS:

`part` = any list denoting a partition

SYNOPSIS:

- The `Part2SgCC` function computes the list of all permutations whose cycle type is the given partition `part`.
- When called with the second argument `'card'`, the function returns the number of elements in the conjugacy class indexed by the partition `part`.
- Whenever there is a conflict between the function name `Part2SgCC` and another name used in the same session, use the long form `SG['Part2SgCC']`.

EXAMPLES:

```
> with(SG):
> Part2SgCC([2,1]);
                    [[1, 3, 2], [2, 1, 3], [3, 2, 1]]
```

```
> Part2SgCC([5,4,3,1], 'card');

                                103783680
```

See Also:

Function: Perm2Betti - compute the Betti polynomial

Calling Sequence:

```
Perm2Betti(perm, 'q')
SG[Perm2Betti](perm, 'q')
```

Parameters:

perm = any list denoting any permutation

Synopsis:

- The Perm2Betti function computes the generating function of the lengths of all permutations that are smaller than a given permutation perm, according to Bruhat order.
- The second parameter 'q' can be arbitrarily choosen.
- Whenever there is a conflict between the function name Perm2Betti and another name used in the same session, use the long form SG['Perm2Betti'].

Examples:

```
> with(SG):
> Perm2Betti([3,4,1,2], 'q');

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

See Also: CompPerm Perm2Interval

Function: 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 `[c1, c2, ...]` be the code of `perm`, then `ci` 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([3, 0, 2, 0, 0]);
```

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

SEE ALSO: `Code2Perm`

FUNCTION: `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: Cycle2Perm Perm2CycleType Vect2Perm

Function: 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: Cycle2Perm Perm2Cycle

Function: 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`

FUNCTION: `Perm2Interval` - compute all codes of permutations below a given one

CALLING SEQUENCE:

```
Perm2Interval(perm)
SG[Perm2Interval](perm)
```

PARAMETERS:

`perm` = any list denoting a permutation

SYNOPSIS:

- The `Perm2Interval` 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 `Perm2Interval` and another name used in the same session, use the long form `SG['Perm2Interval']`.

EXAMPLES:

```
> with(SG):
> Perm2Interval([3,1,2]);

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

> map(Code2Perm, Perm2Interval([3,2,1]));

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

SEE ALSO: CompPerm Perm2Betti IDCA[Perm2Demazure]

FUNCTION: Perm2Inv - invert a permutation

CALLING SEQUENCE:
Perm2Inv(perm)
SG[Perm2Inv](perm)

PARAMETERS:
perm = any list denoting a permutation

SYNOPSIS:

- The Perm2Inv function inverts a permutation perm.
- Whenever there is a conflict between the function name Perm2Inv and another name used in the same session, use the long form SG['Perm2Inv'].

EXAMPLES:

```
> with(SG):
> Perm2Inv([3,1,2,4]);

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

SEE ALSO:

FUNCTION: 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`

FUNCTION: `Perm2ListInv` - calculate the set of inversions of a permutation

CALLING SEQUENCE:

```
Perm2ListInv(perm)
SG[Perm2ListInv](perm)
```

PARAMETERS:

`perm` = any list denoting a permutation

SYNOPSIS:

- The `Perm2ListInv` 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 `Perm2ListInv` and another name used in the same session, use the long form `SG['Perm2ListInv']`.

EXAMPLES:

```
> with(SG):
> Perm2ListInv(Perm2Inv([4,1,3,2]));

                    [{1, 4}, {2, 3}, {2, 4}, {3, 4}]

> Perm2ListInv([4,1,3,2]);

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


See Also: Perm2Inv Perm2Length

Function: Perm2ListRd - list all reduced decompositions of a given permutation

Calling Sequence:

```
Perm2ListRd(perm)
SG[Perm2ListRd](perm)
```

Parameters:

perm = any list denoting a permutation

Synopsis:

- The Perm2ListRd function returns the list of all reduced decompositions of a given permutation, i.e. all expressions of perm as a product (of minimal length) of simple transpositions.
- When used with the second parameter 'nb', it returns the number of such decompositions.
- Whenever there is a conflict between the function name Perm2ListRd and another name used in the same session, use the long form SG['Perm2ListRd'].

Examples:

```
> with(SG):
> Perm2ListRd([3,1,5,2,4]);

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

> Perm2ListRd([6,5,4,3,2,1], 'nb');

                                292864
```


See Also: Perm2Rd Rd2Perm

Function: 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]);
```

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

SEE ALSO: `Mat2Perm` `Perm2RRep` `RRep2Perm`

FUNCTION: `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]);
```

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

```
[                    ]
[ 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( Perm2RRep([2,3,1]) );
```

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

SEE ALSO: Mat2Perm Perm2Mat RRep2Perm

FUNCTION: 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

FUNCTION: Perm2Rise - compute the list of rises of a permutation

CALLING SEQUENCE:

```
Perm2Rise(perm)
SG[Perm2Rise](perm)
```

PARAMETERS:
`perm` = any list denoting a permutation

SYNOPSIS:

- The `Perm2Rise` function computes the list of rises of the permutation `perm`, i.e. the list of positions `i`, such that `perm[i]` < `perm[i+1]`.
- When called with the second argument `'nb'`, it returns the number of rises of the permutation perm.
- When called with the second argument `'rin'`, it returns the sum of the rise positions (rise index).
- When called with the second argument `'corin'`, it returns the sum of the values (`n` - rise position), where `n` is equal to `nops(perm)`.
- Whenever there is a conflict between the function name `Perm2Rise` and another name used in the same session, use the long form `SG['Perm2Rise']`.

EXAMPLES:

```
> with(SG):
> perm:=[3,1,4,2,6,5,7,8]:
> Perm2Rise(perm);
                              [2, 4, 6, 7]

> Perm2Rise(perm, 'nb');
                                   4

> Perm2Rise(perm, 'rin');
                                   9

> Perm2Rise(perm, 'corin');
                                  13
```

SEE ALSO: `Perm2Desc`

FUNCTION: `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 boxes of the diagram are numbered by consecutive numbers, starting from the row number.
- The matrix of the `x`'s is the matrix representing the permutation.
- Reading the entries from right to left and top to bottom, one obtains a canonical reduced decomposition of the permutation (`i` stands for the simple transposition exchanging `i` and `i+1`).
- Whenever there is a conflict between the function name `Perm2Rothe` and another name used in the same session, use the long form `SG['Perm2Rothe']`.

EXAMPLES:

```
> with(SG):
> Perm2Rothe([6,1,4,5,3,2]);
```

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

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

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

SEE ALSO: `Perm2Rd`

FUNCTION: `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` `SgTranspo`

FUNCTION: `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]);
```

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

```
> RRep2Perm( Perm2RRep([3,1,2]) );
```

$$[3, 1, 2]$$

See Also: Mat2Perm Perm2Mat Perm2RRep

Function: RandPerm - return a random permutation

Calling Sequence:

```
RandPerm(n)
RandPerm(n, typeperm)
SG[RandPerm](n)
SG[RandPerm](n, typeperm)
```

Parameters:

n = any positive integer denoting the degree of a symmetric group
typeperm = 'dominant', 'grassmannian'

Synopsis:

- The RandPerm function generates a random permutation belonging to the symmetric group of degree n.
- RandPerm(n, 'dominant') returns a random dominant permutation.
- RandPerm(n, 'grassmannian') returns a random grassmannian permutation.
- 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]
```

```
> r := RandPerm(20, 'dominant');
```

```
 r := [15, 11, 10, 12, 13, 7, 6, 8, 9, 14, 16, 5, 4, 17, 3, 18, 2, 19, 20, 1]
```

```
> Perm2Code(r);
```

```
       [14, 10, 9, 9, 9, 6, 5, 5, 5, 5, 5, 4, 3, 3, 2, 2, 1, 1, 1, 0]
```

See Also: Perm2Code

Function: Rd2Perm - calculate a permutation from a decomposition

Calling Sequence:

```
Rd2Perm(rd)
Rd2Perm(rd, n)
SG[Rd2Perm](rd)
```

`SG[Rd2Perm](rd, n)`

PARAMETERS:
`rd` = any 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,...]`, then `Rd2Perm(rd)` = `MultPerm(s_i, s_j, s_k, ...)` where `s_i` stands for `SgTranspo(i, n)`.
- When the second argument is not present, `n` is the smallest possible degree.
- 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`

FUNCTION: `SgCharTable` - compute the table of characters

CALLING SEQUENCE:

```
SgCharTable(n)
SgCharTable(n, 'index')
SgCharTable(n, part_1, part_2)
SG[SgCharTable](n)
SG[SgCharTable](n, 'index')
SG[SgCharTable](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 SgCharTable 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 SgCharTable(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.
- Partitions indexing rows and columns are also given by PART[ListPart](n).
- 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 SgCharTable and another name used in the same session, use the long form SG['SgCharTable'].

EXAMPLES:

```
> with(SG):
> SgCharTable(3);
```

$$\begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 2 \\ 1 & -1 & 1 \end{bmatrix}$$

```
> SgCharTable(3, 'index');
```

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

```
> SgCharTable(3, [2,1], [1,1,1]);
```

```
2
```

SEE ALSO: TYP[IsPart] PART[ListPart]

FUNCTION: SgDimRep - compute the dimension of a representation

CALLING SEQUENCE:

SgDimRep(part)
SG[SgDimRep](part)

PARAMETERS:

part = any list denoting a (skew) partition

SYNOPSIS:

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

- When part is a skew partition, it computes the dimension of the representation indexed by the skew partition `part[1]/part[2]`.
- Adding the parameter `'q'`, it computes the specialization of $s_{\texttt{part}}$ or $s_{\texttt{part[1]/part[2]}}$ in $x_1 = 1, x_2 = \texttt{q}, x_3 = \texttt{q}^2, ...$
- Whenever there is a conflict between the function name `SgDimRep` and another name used in the same session, use the long form `SG['SgDimRep']`.

EXAMPLES:

```
> with(SG):
> SgDimRep([4,4,3,2,1,1,1]);

                                512512

> SgDimRep([[4,2,1,1], [2,1]]);

                                  30

> SgDimRep([[4,2,1,1], [2,1]], 'q');

                2           2       4    3    2
              (q  + q + 1) (q  + 1) (q  + q  + q  + q + 1) q
```

SEE ALSO: `SgCharTable`

FUNCTION: `SgTranspo` - return a transposition

CALLING SEQUENCE:

```
SgTranspo(i)
SgTranspo(i, n)
SgTranspo(i, j, n)
SG[SgTranspo](i)
SG[SgTranspo](i, n)
SG[SgTranspo](i, j, n)
```

PARAMETERS:

`n` = the degree of a symmetric group
`i` = any positive integer
`j` = any positive integer

SYNOPSIS:

- The `SgTranspo` function returns a transposition, that is a permutation interchanging two values and fixing all others.
- `SgTranspo(i) =[1, ..., i+1, i]`.
- `SgTranspo(i,n) =[1, ..., i+1, i, ..., n]`.

- SgTranspo(i,j,n) =[1, ..., j, ..., i, ..., n].
- Whenever there is a conflict between the function name SgTranspo and another name used in the same session, use the long form SG['SgTranspo'].

EXAMPLES:

```
> with(SG):
> SgTranspo(2, 6);
```

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

```
> SgTranspo(3, 5, 7);
```

```
                    [1, 2, 5, 4, 3, 6, 7]
```

SEE ALSO:

FUNCTION: 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

5.5 Using SGA to work with the symmetric group algebra

HELP FOR: A package for 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.
- A general element is expressed as a linear combination of `A[perm]`, where `perm` is a permutation, with arbitrary coefficients.
- To get more information on permutations do either ?`TYP,Permutation` or ?`TYP[Permutation]`.
- Whenever there is a conflict between a function name in `SGA` and another name used in the same session, use the long form `SGA`[<function>].
- The available functions are:

`Part2SgaCIdemp`	`Rd2Sga`	`Sga2Carre`	`Sga2Nabla`
`Sga2Table`	`SgaAdd`	`SgaJucis`	`SgaMinus`
`SgaMult`	`SgaNormal`	`SgaOnPol`	`SgaOnX`
`SgaOnXfix`	`SgaYang`	`Table2Sga`	

- For help with a particular function do either ?`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):
> Sga2Carre(A[3,1,2]) &!* Sga2Nabla(A[3,1,2]);

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

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

FUNCTION: Part2SgaCIdemp - compute a central idempotent of the symmetric group algebra

CALLING SEQUENCE:

```
Part2SgaCIdemp(part)
SGA[Part2SgaCIdemp](part)
```

PARAMETERS:

`part` = any list denoting a partition

SYNOPSIS:

- The `Part2SgaCIdemp` 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 `Part2SgaCIdemp` and another name used in the same session, use the long form `SGA['Part2SgaCIdemp']`.

EXAMPLES:

```
> with(SGA):
> Part2SgaCIdemp([2,1]);

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

SEE ALSO:

FUNCTION: Rd2Sga - calculate a product of generators

CALLING SEQUENCE:

```
Rd2Sga(rd)
Rd2Sga(rd, n)
SGA[Rd2Sga](rd)
SGA[Rd2Sga](rd, n)
```

PARAMETERS:

`rd` = any list denoting a decomposition
`n` = any positive integer denoting the degree of a symmetric group

SYNOPSIS:

- The `Rd2Sga` function calculates the product of the generators given by the reduced decomposition `rd`. The product is taken in the symmetric group algebra.
- `Rd2Sga(rd)=SG['Rd2Perm'](rd)`.
- When the second argument is not present, `n` is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2Sga` and another name used in the same session, use the long form `SGA['Rd2Sga']`.

EXAMPLES:

```
> with(SGA):
> Rd2Sga([1,2,1], 5);

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

> Rd2Sga([1,2,1]);

                                 A[3, 2, 1]

> Rd2Sga([1,2,1,2]);

                                 A[3, 1, 2]
```

SEE ALSO: SG[Rd2Perm]

FUNCTION: Sga2Carre - Carré element of the symmetric group algebra

CALLING SEQUENCE:

Sga2Carre(e)
SGA[Sga2Carre](e)

PARAMETERS:

e = any element of the symmetric group algebra

SYNOPSIS:

- The Sga2Carre function converts an element of the symmetric group algebra into a Carré operator by replacing each A[perm] by the corresponding Carré element of the symmetric group algebra.
- The Sga2Carre together with the Sga2Nabla functions are the building blocks of idempotents in the symmetric group algebra.
- Given a permutation perm, Sga2Carre(A[perm]) is a certain product of factors of the type (SgTranspo(i,n) + 1/k) where the shifts 1/k are given by Yang-Baxter conditions. Special cases provide idempotents. For example, Sga2Carre(A[n, n-1, ..., 2, 1]) = sum of all permutations of $\mathfrak{S}_n$.
- They are defined by the recursion: for a simple transposition si and a permutation perm, such that length(perm si) > length(perm), then Sga2Carre(A[perm si])=Sga2Carre(A[perm]) &!* (si + 1/k), in which k=perm[i+1] - perm[i].
- Whenever there is a conflict between the function name Sga2Carre and another name used in the same session, use the long form SGA['Sga2Carre'].

EXAMPLES:

```
> with(SGA):
> Sga2Carre(q*A[2,3,1] + A[1,2,3]);

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

SEE ALSO: Sga2Nabla

FUNCTION: Sga2Nabla - Nabla element of the symmetric group algebra

CALLING SEQUENCE:

```
Sga2Nabla(e)
SGA[Sga2Nabla](e)
```

PARAMETERS:

`e` = any element of the symmetric group algebra

SYNOPSIS:

- The `Sga2Nabla` function converts an element of the symmetric group algebra into a Nabla operator by replacing each `A[perm]` by the corresponding Nabla element of the symmetric group algebra.
- The `Sga2Nabla` together with the `Sga2Carre` functions are the building blocks of idempotents in the symmetric group algebra.
- `Sga2Nabla(A[Identity]) = 0`.
- Given a permutation `perm`, `Sga2Nabla(A[perm])` is a certain product of factors of the type `(SgTranspo(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 `si` and a permutation `perm`, such that `length(perm si) > length(perm)`, then `Sga2Nabla(A[perm si])=Sga2Nabla(A[perm]) &!* (si - 1/k)`, in which `k=perm[i+1] - perm[i]`.
- Whenever there is a conflict between the function name `Sga2Nabla` and another name used in the same session, use the long form `SGA['Sga2Nabla']`.

EXAMPLES:

```
> with(SGA):
> Sga2Nabla(q*A[2,3,1] + A[1,2,3]);

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

See Also: Sga2Carre

Function: Sga2Table - convert an element of the symmetric group algebra into a table

Calling Sequence:

```
Sga2Table(e)
SGA[Sga2Table](e)
```

Parameters:

e = any element of the symmetric group algebra

Synopsis:

- The Sga2Table function converts an element of the symmetric group algebra into a table. The table is indexed by permutations and each entry corresponding to a permutation is the coefficient of this permutation in the element e.
- Note that all permutations in the indices of the table are adjusted to belong to the same minimal symmetric group.
- This function can be useful if one wants to act either on coefficients or on permutations. For instance, one can use the map function to act on coefficients.
- Whenever there is a conflict between the function name Sga2Table and another name used in the same session, use the long form SGA['Sga2Table'].

Examples:

```
> with(SGA):
> t := Sga2Table((x1+x2)*A[2,3,1] + x1^2*A[1,3,2] + A[1,2]);

                        t := table(sparse,[
                               [2, 3, 1] = x1 + x2
                                                2
                               [1, 3, 2] = x1
                               [1, 2, 3] = 1
                             ])

> t := map(SP['ToX'], t);

                       t := table(sparse,[
                              [2, 3, 1] = X[1, 3, 2]
                              [1, 3, 2] = X[3, 1, 2]
                              [1, 2, 3] = X[1]
                            ])

> Table2Sga(t);

    X[1, 3, 2] A[2, 3, 1] + X[3, 1, 2] A[1, 3, 2] + X[1] A[1, 2, 3]
```

See Also: `Table2Sga SP[ToX]`

Function: `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*A[3,1,2]:
> b:=t - A[1,3,2]:
> SgaAdd(a, b);
```

$$-A[1,3,2] + \frac{u\,A[3,1,2]}{t} + (q+t)\,A[1,2,3]$$

```
> a &!+ b;
```

$$-A[1,3,2] + \frac{u\,A[3,1,2]}{t} + (q+t)\,A[1,2,3]$$

See Also:

Function: `SgaJucis` - Jucis-Murphy element of the symmetric group algebra

Calling Sequence:

```
SgaJucis(i)
SGA[SgaJucis](i)
```

Parameters:

`i` = any positive integer

SYNOPSIS:

- The `SgaJucis` function computes the `i`-th Jucis-Murphy element inside the algebra of the symmetric group of degree `n` (`i`≤`n`), where `n` is taken to be `i` by default or the second argument if available.
- 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: `SgaJucis(1)` = 0, `SgaJucis(2)` = `T(1,2)`, `SgaJucis(3)` = `T(1,3)` + `T(2,3)`, ..., `SgaJucis(k)` = `T(1,k)` + `T(2,k)` + ... + `T(k-1,k)`.
- Whenever there is a conflict between the function name `SgaJucis` and another name used in the same session, use the long form `SGA['SgaJucis']`.

EXAMPLES:

```
> with(SGA):
> SgaJucis(2);
```

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

```
> SgaJucis(4, 6);
```

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

SEE ALSO: `SG[SgTranspo]`

FUNCTION: `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*A[3,1,2]:
> b:=t - A[1,3,2]:
> SgaMinus(a, b);
```

$$A[1, 3, 2] + \frac{u\ A[3, 1, 2]}{t} + (q - t)\ A[1, 2, 3]$$

```
> a &!- a;
```

$$0$$

SEE ALSO:

FUNCTION: `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*A[3,1,2]:
> b:=t - A[1,3,2]:
> SgaMult(a, b);
```

$$-\frac{u\ A[3, 2, 1]}{t} - q\ A[1, 3, 2] + u\ A[3, 1, 2] + q\ t\ A[1, 2, 3]$$

```
> a &!* b;
```

```
      u A[3, 2, 1]
- ------------ - q A[1, 3, 2] + u A[3, 1, 2] + q t A[1, 2, 3]
       t
```

See Also:

Function: `SgaNormal` - reduce each permutation inside an expression to the minimum symmetric group containing it

Calling Sequence:

```
SgaNormal(e)
SgaNormal(e, m)
SGA[SgaNormal](e)
SGA[SgaNormal](e, m)
```

Parameters:

`e` = any element of the symmetric group algebra
`m` = any positive integer denoting the degree of a symmetric group

Synopsis:

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

Examples:

```
> with(SGA):
> SgaNormal(a*A[3,1,2] - b^2*A[2,1,3,4]);
```

```
                          2
                    - b  A[2, 1] + a A[3, 1, 2]
```

```
> SgaNormal(a*A[3,1,2] - b^2*A[2,1,3,4], 3);
```

```
                       2
                 - b  A[2, 1, 3] + a A[3, 1, 2]
```

See Also:

FUNCTION: `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`. Each permutation acts by permuting the indices of the variables.
- By default the algebra acts on the variables $x_1, x_2, x_3, ...$
- 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 $y_1, y_2, y_3, ...$
- 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(A[2,1], x1+x3);
                                    x2 + x3

> SgaOnPol(A[2,1], x1+x3, 'y');
                                    x1 + x3

> SgaOnPol(A[2,1] - x3*A[3,1,2], x1+x3);

                          - x3 (x2 + x3) + x2 + x3
```

SEE ALSO:

FUNCTION: `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 **X** Schubert basis.
- The expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.
- One may collect the result by adding a third argument: this is done by `SgaOnX(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can use `SgaOnX(e_1, exp, 'noexpand', 'collect')`.
- The action of the element `SgTranspo(i,n)` on the Schubert polynomial `X[perm]` is : let `nu = MultPerm(perm, SgTranspo(i,n))`, then

  `X[perm]` $\longrightarrow$ `X[perm]` if `length(nu)` $>$ `length(perm)`
  `X[perm]` $\longrightarrow$ `X[perm]` + `(x_i+1 - x_i) X[nu]` otherwise.
- The result is expressed on the **X** Schubert basis and is not collected.
- Whenever there is a conflict between the function name `SgaOnX` and another name used in the same session, use the long form `SGA['SgaOnX']`.

EXAMPLES:

```
> with(SGA):
> SgaOnX(q^4*A[1,3,2] - q^3*A[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:

FUNCTION: `SgaOnXfix` - action of an element of the symmetric group algebra on the ring of polynomials as a free module over **Sym**

CALLING SEQUENCE:
`SgaOnXfix(e_1, exp)`
`SGA[SgaOnXfix](e_1, exp)`

PARAMETERS:
`e_1` = any element of the symmetric group algebra

`exp` = any expression

SYNOPSIS:

- The `SgaOnXfix` function realizes the action of an element of the symmetric group algebra, say `e_1`, on an expression `exp` expressed on the **X** Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.
- The expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.
- One may collect the result by adding a third argument: this is done by `SgaOnXfix(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: `SgaOnXfix(e_1, exp, 'noexpand', 'collect')`.
- The action of the element `SgTranspo(i,n)` on the Schubert polynomial `X[perm]` is : let `nu = MultPerm(perm, SgTranspo(i,n))`, then

  `X[perm]` $\longrightarrow$ `X[perm]` if `length(nu)` $>$ `length(perm)`
  `X[perm]` $\longrightarrow$ `X[perm]` + `(x_i+1 - x_i) X[nu]` otherwise.
- The result is expressed on the **X** Schubert basis corresponding to a given symmetric group and is not collected.
- Whenever there is a conflict between the function name `SgaOnXfix` and another name used in the same session, use the long form `SGA['SgaOnXfix']`.

EXAMPLES:

```
> with(SGA):
> _FMn;
```

$$4$$

```
> SgaOnXfix(q*A[3,2,1], s[1,1]*X[2,4,1,3], collect);
```

$$(s[2,\,1] + s[1,\,1,\,1])\,q^{3}\,X[1,\,3,\,4,\,2] - q^{3}\,X[2,\,3,\,4,\,1]\,s[1,\,1]$$
$$+ (s[2,\,2] + s[2,\,1,\,1] + s[1,\,1,\,1,\,1])\,q^{3}\,X[2,\,1,\,3,\,4]$$
$$- (s[2,\,2,\,1] + s[2,\,1,\,1,\,1] + s[1,\,1,\,1,\,1,\,1])\,q^{3}\,X[1,\,2,\,3,\,4]$$
$$- (s[2,\,1] + s[1,\,1,\,1])\,q^{3}\,X[2,\,1,\,4,\,3] + q^{3}\,X[4,\,1,\,2,\,3]\,s[1,\,1]$$

SEE ALSO:

FUNCTION: `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 `ListPerm(n)` is a linear basis of the symmetric group algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say `'y'`, one specifies that coefficients are in the y_i's.
- When this second parameter is `'num'` then $x_1, x_2, x_3, ...$ are specialized to $1, 2, 3, ...$
- This basis is defined by the recursion: for a simple transposition `sk` and a permutation `perm`, such that `length(perm sk) > length(perm)`, then `SgaYang(perm sk) = SgaYang(perm) &!* (1 + (x_j - x_i) sk)` 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) A[1, 3, 2] - (- x3 + x1) (x3 - x2) A[3, 1, 2]

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

> SgaYang([3,1,2], 'y');

         (y3 - y2) A[1, 3, 2] - (- y3 + y1) (y3 - y2) A[3, 1, 2]

       + A[1, 2, 3] + (y3 - y1) A[2, 1, 3]
```

SEE ALSO:

FUNCTION: `Table2Sga` - convert a table into an element of the symmetric group algebra

CALLING SEQUENCE:

```
Table2Sga(t)
SGA[Table2Sga](t)
```

PARAMETERS:

`t` = any table denoting an element of the symmetric group algebra

SYNOPSIS:

- The `Table2Sga` function converts a table into an element of the symmetric group algebra. The table should be indexed by permutations and each entry corresponding to a permutation `perm` is interpreted as the coefficient of `perm` in the expression of the element `e` in the basis of permutations.
- Whenever there is a conflict between the function name `Table2Sga` and another name used in the same session, use the long form `SGA['Table2Sga']`.

EXAMPLES:

```
> with(SGA):
> t:=table([[2,3,1]=x1+x2, [1,3,2]=x1^2, [1,2,3]=1]);

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

> Table2Sga(t);

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

SEE ALSO: `Sga2Table`

5.6 NCA to compute in the nilCoxeter algebra

HELP FOR: A package for 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.
- A general element is expressed as a linear combination of `A[perm]`, where `perm` is a permutation, with arbitrary coefficients.
- 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:

Nca2Table	NcaAdd	NcaMinus	NcaMult
NcaNormal	NcaOnPol	NcaOnX	NcaOnXfix
NcaOnXX	NcaOnY	NcaOnYY	NcaYang
Rd2Nca	Table2Nca		

- For help with a particular function do either ?`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*A[3,1,2]:
> b:=t*A[2,1,3] - A[1,3,2]:
> NcaMult(a, b);
```

$$q\ t\ A[2,\ 1,\ 3] - q\ A[1,\ 3,\ 2] - \frac{u\ A[3,\ 2,\ 1]}{t}$$

FUNCTION: Nca2Table - convert an element of the nilCoxeter algebra into a table

CALLING SEQUENCE:

```
Nca2Table(e)
NCA[Nca2Table](e)
```

PARAMETERS:

e = any element of the nilCoxeter algebra

SYNOPSIS:

- The Nca2Table function converts an element of the nilCoxeter algebra into a table. The table is indexed by permutations and each entry corresponding to a permutation perm is the coefficient of perm in e.
- This function can be useful if one wants to act either on coefficients or on permutations. For instance, one can use the map function to act on coefficients.
- Whenever there is a conflict between the function name Nca2Table and another name used in the same session, use the long form NCA['Nca2Table'].

EXAMPLES:

```
> with(NCA):
> t := Nca2Table((x1+x2)*A[2,3,1] + x1^2*A[1,3,2] + A[1,2,3]);

                        t := table(sparse,[
                               [2, 3, 1] = x1 + x2
                                                 2
                               [1, 3, 2] = x1
                               [1, 2, 3] = 1
                             ])
> t := map(SP['ToX'], t);

                       t := table(sparse,[
                              [2, 3, 1] = X[1, 3, 2]
                              [1, 3, 2] = X[3, 1, 2]
                              [1, 2, 3] = X[1]
                            ])

> Table2Nca(t);

    X[1, 3, 2] A[2, 3, 1] + X[3, 1, 2] A[1, 3, 2] + X[1] A[1, 2, 3]
```

SEE ALSO: Table2Nca SP[ToX]

FUNCTION: 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*A[3,1,2]:
> b:=t - A[1,3,2]:
> NcaAdd(a, b);
```

$$(q + t)\ A[1, 2, 3] + \frac{u\ A[3, 1, 2]}{t} - A[1, 3, 2]$$

```
> a &@+ b;
```

$$(q + t)\ A[1, 2, 3] + \frac{u\ A[3, 1, 2]}{t} - A[1, 3, 2]$$

SEE ALSO:

FUNCTION: `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*A[3,1,2]:
> b:=t - A[1,3,2]:
> NcaMinus(a, b);
```

$$(q - t)\ A[1, 2, 3] + \frac{u\ A[3, 1, 2]}{t} + A[1, 3, 2]$$

```
> a &@- a;
```

$$0$$

SEE ALSO:

FUNCTION: `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*A[3,1,2]:
> b:=t*A[2,1,3]-A[1,3,2]:
> NcaMult(a, b);
```

$$q\ t\ A[2, 1, 3] - q\ A[1, 3, 2] - \frac{s\ A[3, 2, 1]}{t}$$

```
> a &@* a;
```

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

See Also:

Function: `NcaNormal` - normalize an element of the nilCoxeter algebra

Calling Sequence:

```
NcaNormal(e)
NcaNormal(e, m)
NCA[NcaNormal](e)
NCA[NcaNormal](e, m)
```

Parameters:

`e` = any element of the nilCoxeter algebra
`m` = any positive integer denoting the degree of a symmetric group

Synopsis:

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

Examples:

```
> with(NCA):
> NcaNormal(a*A[3,1,2] - b^2*A[2,1,3,4]);
```

```
                              2
                          - b  A[2, 1] + a A[3, 1, 2]
```

```
> NcaNormal(a*A[3,1,2] - b^2*A[2,1,3,4], 3);
```

```
                             2
                         - b  A[2, 1, 3] + a A[3, 1, 2]
```

See Also:

Function: `NcaOnPol` - action of an element of the algebra on a polynomial

Calling Sequence:

```
NcaOnPol(e_1, exp)
```

```
NcaOnPol(e_1, exp, v)
NCA[NcaOnPol](e_1, exp)
NCA[NcaOnPol](e_1, exp, v)
```

PARAMETERS:
`e_1` = any element of the nilCoxeter algebra
`exp` = any expression
`v` = any (extra) string

SYNOPSIS:

- The `NcaOnPol` function realizes the action of an element of the nilCoxeter algebra, say `e_1`, on an expression `exp`.
- Simple divided differences ∂_i are defined to be the operators:

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

- By default the algebra acts on the variables $x_1, x_2, x_3, ...$
- When called with a third argument `v`, being, say `'y'`, the `NcaOnPol` function acts on the variables $y_1, y_2, y_3, ...$
- 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*A[3,1,2], x1^2*x2*y1^2*y2^3);

                              2      2   3
                             q  x1 y1  y2

> NcaOnPol(q^2*A[3,1,2], x1^2*x2*y1^2*y2^3, 'y');

                          2              2   2
                    - q  (y2 + y3) y1  x1  x2
```

SEE ALSO:

FUNCTION: `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.
- The expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.
- One may collect the result by adding a third argument: this is done by `NcaOnX(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can enter `NcaOnX(e_1, exp, 'noexpand', 'collect')`.
- A simple divided difference ∂_i acts on a Schubert polynomial `X[perm]` by sending it to 0 if `perm[i]` < `perm[i+1]`, or to `X[new_perm]` where `new_perm` 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*A[1,3,2] - q^3*A[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`

FUNCTION: `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).
- The expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.
- One may collect the result by adding a third argument: this is done by `NcaOnXX(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can use `NcaOnXX(e_1, exp, 'noexpand', 'collect')`.
- A simple divided difference ∂_i acts on a Schubert polynomial `XX[perm]` by sending it to 0 if `perm[i]` $<$ `perm[i+1]`, or to `XX[new_perm]` where `new_perm` 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*A[1,3,2] - q^3*A[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`

FUNCTION: `NcaOnXfix` - action of an element of the nilCoxeter algebra on the ring of polynomials as a free module over **Sym**

CALLING SEQUENCE:

```
NcaOnXfix(e_1, exp)
NCA[NcaOnXfix](e_1, exp)
```

PARAMETERS:

`e_1` = any element of the nilCoxeter algebra
`exp` = any expression

SYNOPSIS:

- The `NcaOnXfix` function realizes the action of an element of the nilCoxeter, say `e_1`, on an expression exp expressed on the **X** Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.
- The expression `exp` is expanded and the result is not collected.

- One may add 'noexpand' just after the argument exp to choose not to expand the expression exp before treating it.
- One may collect the result by adding a third argument: this is done by NcaOnXfix(e_1, exp, 'collect'). Moreover, one can use both noexpand and collect options: for instance, one can use NcaOnXfix(e_1, exp, 'noexpand', 'collect').
- A simple divided difference ∂_i acts on a Schubert polynomial X[perm] by sending it to 0 if perm[i] < perm[i+1], or to X[new_perm] where new_perm 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 corresponding to a given symmetric group and is not collected.
- Whenever there is a conflict between the function name NcaOnXfix and another name used in the same session, use the long form NCA['NcaOnXfix'].

Examples:

```
> with(NCA):
> _FMn;
```

$$4$$

```
> NcaOnXfix(q*A[1,3,2], s[1,1]*X[2,4,1,3]);
```

$$s[1,\ 1]\ q\ X[2,\ 1,\ 4,\ 3]$$

See Also: FM[FM_n]

Function: 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.
- The expression exp is expanded and the result is not collected.
- One may add 'noexpand' just after the argument exp to choose not to expand the expression exp before treating it.

- One may collect the result by adding a third argument: this is done by `NcaOnY(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can use `NcaOnY(e_1, exp, 'noexpand', 'collect')`.
- A simple divided difference ∂_i acts on a Schubert polynomial `Y[code]` by sending it to 0 if `code[i]` $\leq$ `code[i+1]`, or to `Y[new_code]`s where `new_code` is defined by `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 $\mathbf{Y}$ 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*A[1,3,2] - q^3*A[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`

FUNCTION: `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).
- The expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.
- One may collect the result by adding a third argument: this is done by `NcaOnYY(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can use `NcaOnYY(e_1, exp, 'noexpand', 'collect')`.

- A simple divided difference ∂_i acts on a Schubert polynomial `YY[code]` by sending it to 0 if `code[i]` $\leq$ `code[i+1]`, or to `YY[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*A[1,3,2] - q^3*A[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: `NcaOnXX`

Function: `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 `ListPerm(n)` is a linear basis of the nilCoxeter algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say `'y'`, one specifies that coefficients are in the y_i's.
- When this second parameter is `'num'` then $x_1, x_2, x_3, ...$ are specialized to 1, 2, 3, ...
- This basis is defined by the recursion : for a simple transposition `sk` and a permutation perm, such that `length(perm sk)` $>$ `length(perm)`, then `NcaYang(perm sk)` = `NcaYang(perm) &@* (1 + (x_j - x_i)` ∂_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 - x1) A[2, 1, 3] + (- x3 + x1) (x2 - x3) A[3, 1, 2]

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

SEE ALSO:

FUNCTION: `Rd2Nca` - calculate a product of generators

CALLING SEQUENCE:

```
Rd2Nca(rd)
Rd2Nca(rd, n)
NCA[Rd2Nca](rd)
NCA[Rd2Nca](rd, n)
```

PARAMETERS:

`rd` = any list denoting a decomposition
`n` = any positive integer denoting the degree of a symmetric group

SYNOPSIS:

- The `Rd2Nca` function calculates the product of the generators given by the decomposition `rd`. The product is taken in the nilCoxeter algebra.
- If `rd` is reduced then `Rd2Nca(rd)`=`SG['Rd2Perm'](rd)`, and 0 otherwise.
- When the second argument is not present, `n` is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2Nca` and another name used in the same session, use the long form `NCA['Rd2Nca']`.

EXAMPLES:

```
> with(NCA):
> Rd2Nca([1,2,1], 5);

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

> Rd2Nca([1,2,1]);

                                 A[3, 2, 1]

> Rd2Nca([1,2,1,2]);

                                      0
```

See Also: SG[Rd2Perm]

Function: Table2Nca - convert a table into an element of the nilCoxeter algebra

Calling Sequence:

```
Table2Nca(t)
NCA[Table2Nca](t)
```

Parameters:

t = any table denoting an element of the nilCoxeter algebra

Synopsis:

- The Table2Nca function converts a table into an element of the nilCoxeter algebra. The table should be indexed by permutations and each entry corresponding to a permutation perm is interpreted as the coefficient of perm in the expression of the element e in the basis of permutations.
- Whenever there is a conflict between the function name Table2Nca and another name used in the same session, use the long form NCA['Table2Nca'].

Examples:

```
> with(NCA):
> t:=table([[2,3,1]=x1+x2, [1,3,2]=x1^2, [1,2,3]=1]);

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

> Table2Nca(t);

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

See Also: Nca2Table

5.7 IDCA to compute in the idCoxeter algebra

HELP FOR: A package for 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: $\pi_i^2 = -\pi_i$.
- A general element is expressed as a linear combination of `A[perm]`, where `perm` is a permutation, with arbitrary coefficients.
- 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:

`Idca2Demazure`	`Idca2Table`	`IdcaAdd`	`IdcaMinus`
`IdcaMult`	`IdcaNormal`	`IdcaOnPol`	`IdcaOnX`
`IdcaOnXfix`	`IdcaYang`	`Rd2Idca`	`Table2Idca`

- For help with a particular function do either ?`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*A[3,1,2]:
> b:=t*A[2,1,3] - A[1,3,2]:
> IdcaMult(a, b);
```

$$-q\,A[1,3,2] + q\,t\,A[2,1,3] - \frac{u\,A[3,2,1]}{t} - u\,A[3,1,2]$$

FUNCTION: `Idca2Demazure` - Demazure element of the idCoxeter algebra

CALLING SEQUENCE:

```
Idca2Demazure(e)
IDCA[Idca2Demazure](e)
```

PARAMETERS:

`e` = any element of the idCoxeter algebra

SYNOPSIS:

- The `Idca2Demazure` function converts an element of the idCoxeter algebra into a Demazure operator by replacing each `A[perm]` by the corresponding Demazure element of the idCoxeter algebra.
- Given a permutation `perm`, with reduced decomposition $s_i s_j s_k$... where s_i=`SgTranspo(i)`, then `Idca2Demazure(A[perm])` is equal to the product $(1+\pi_i)$ `&$*` $(1+\pi_j)$ `&$*` $(1+\pi_k)$ `&$*` ...
- `Idca2Demazure(A[perm])` is the sum of all permutations below than `perm` (for the Bruhat order).
- Whenever there is a conflict between the function name `Idca2Demazure` and another name used in the same session, use the long form `IDCA['Idca2Demazure']`.

EXAMPLES:

```
> with(IDCA):
> Idca2Demazure(q*A[2,3,1] + A[1,2,3]);

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

SEE ALSO:

FUNCTION: `Idca2Table` - convert an element of the idCoxeter algebra into a table

CALLING SEQUENCE:

```
Idca2Table(e)
IDCA[Idca2Table](e)
```

PARAMETERS:

`e` = any element of the idCoxeter algebra

SYNOPSIS:

- The `Idca2Table` function converts an element of the idCoxeter algebra into a table. The table is indexed by permutations and each entry corresponding to a permutation `perm` is the coefficient of `perm` in `e`.
- This function can be useful if one wants to act either on coefficients or on permutations. For instance, one can use the `map` function to act on coefficients.
- Whenever there is a conflict between the function name `Idca2Table` and another name used in the same session, use the long form `IDCA['Idca2Table']`.

EXAMPLES:

```
> with(IDCA):
> t := Idca2Table((x1+x2)*A[2,3,1] + x1^2*A[1,3,2] + A[1,2,3]);

                        t := table(sparse,[
                               [2, 3, 1] = x1 + x2
                                                2
                               [1, 3, 2] = x1
                               [1, 2, 3] = 1
                             ])
> t := map(SP['ToX'], t);

                       t := table(sparse,[
                              [2, 3, 1] = X[1, 3, 2]
                              [1, 3, 2] = X[3, 1, 2]
                              [1, 2, 3] = X[1]
                            ])

> Table2Idca(t);

     X[1, 3, 2] A[2, 3, 1] + X[3, 1, 2] A[1, 3, 2] + X[1] A[1, 2, 3]
```

SEE ALSO: Table2Idca SP[ToX]

FUNCTION: 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*A[3,1,2]:
> b:=t - A[1,3,2]:
> IdcaAdd(a, b);
```

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

```
> a &$+ b;
```

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

See Also:

Function: `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*A[3,1,2]:
> b:=t - A[1,3,2]:
> IdcaMinus(a, b);
```

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

```
> a &$- a;
```

```
                                    0
```

See Also:

Function: 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*A[3,1,2]:
> b:=t*A[2,1,3] - A[1,3,2]:
> IdcaMult(a, b);
```

$$q\,t\,A[2,\,1,\,3] - q\,A[1,\,3,\,2] - u\,A[3,\,1,\,2] - \frac{u\,A[3,\,2,\,1]}{t}$$

```
> a &$* b;
```

$$q\,t\,A[2,\,1,\,3] - q\,A[1,\,3,\,2] - u\,A[3,\,1,\,2] - \frac{u\,A[3,\,2,\,1]}{t}$$

See Also:

Function: IdcaNormal - normalize an element of the idCoxeter algebra

Calling Sequence:

```
IdcaNormal(e)
IdcaNormal(e, m)
IDCA[IdcaNormal](e)
IDCA[IdcaNormal](e, m)
```

Parameters:

e = any element of the idCoxeter algebra

`m` = any positive integer denoting the degree of a symmetric group

SYNOPSIS:

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

EXAMPLES:

```
> with(IDCA):
> IdcaNormal(a*A[3,1,2] - b^2*A[2,1,3,4]);

                              2
                        - b  A[2, 1] + a A[3, 1, 2]

> IdcaNormal(a*A[3,1,2] - b^2*A[2,1,3,4], 3);

                              2
                        - b  A[2, 1, 3] + a A[3, 1, 2]
```

SEE ALSO:

FUNCTION: `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 `SgTranspo(i,n)` is:

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

where `exp'` is the image of `exp` under the exchange of x_i and x_{i+1}.

- By default the algebra acts on the variables $x_1, x_2, x_3, ...$
- When called with a third argument `v`, being, say `'y'`, the `IdcaOnPol` function acts on the variables $y_1, y_2, y_3, ...$
- 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*A[3,1,2], x1^2*x2*y1^2*y2^3);

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

> IdcaOnPol(q^2*A[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:

Function: `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 expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.
- One may collect the result by adding a third argument: this is done by `IdcaOnX(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can use `IdcaOnX(e_1, exp, 'noexpand', 'collect')`.

- The action of the element `SgTranspo(i,n)` on the Schubert polynomial `X[perm]` is : let `nu = MultPerm(perm, SgTranspo(i,n))`, then:

  `X[perm]` $\longrightarrow$ 0 if `length(nu)` $>$ `length(perm)`
  `X[perm]` $\longrightarrow$ `X[nu] x_i+1` otherwise.

- 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*A[1,3,2] - q^3*A[2,1], z*X[3,2,1] - X[2,1,3]);
```

$$- X[3,2,1]\, z\, q^{4} + X[3,1,4,2]\, z\, q^{4} + X[3,2,1]\, z\, q^{3} - X[2,4,1,3]\, z\, q^{3} - X[2,1]\, q^{3} + X[1,3,2]\, q^{3}$$

See Also:

Function: `IdcaOnXfix` - action of an element of the idCoxeter algebra on the ring of polynomials as a free module over **Sym**

Calling Sequence:

```
IdcaOnXfix(e_1, exp)
IDCA[IdcaOnXfix](e_1, exp)
```

Parameters:

`e_1` = any element of the symmetric group algebra
`exp` = any expression

Synopsis:

- The `IdcaOnXfix` function realizes the action of an element of the idCoxeter algebra, say `e_1`, on an expression exp expressed on the **X** Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.

- The expression `exp` is expanded and the result is not collected.

- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.

- One may collect the result by adding a third argument: this is done by `IdcaOnXfix(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, `IdcaOnXfix(e_1, exp, 'noexpand', 'collect')`.

- The action of the element `SgTranspo(i,n)` on the Schubert polynomial `X[perm]` is : let `nu = MultPerm(perm, SgTranspo(i,n))`, then

  `X[perm]` $\longrightarrow$ 0 if `length(nu)` > `length(perm)`
  `X[perm]` $\longrightarrow$ `X[nu] x_i+1` otherwise.

- The result is expressed on the **X** Schubert basis corresponding to a given symmetric group and is not collected.

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

Examples:

```
> with(IDCA):
> _FMn;
```

$$4$$

```
> IdcaOnXfix(q*A[1,3,2], s[1,1]*X[2,4,1,3], collect);

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

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

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

See Also:

Function: `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 `ListPerm(n)` is a linear basis of the idCoxeter algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say `'y'`, one specifies that coefficients are in the y_i's.
- When this second parameter is `'num'` then $x_1, x_2, x_3, ...$ are specialized to $q^1, q^2, q^3, ...$
- This basis is defined by the recursion: for a simple transposition `sk` and a permutation `perm`, such that `length(perm sk)` > `length(perm)`, then `IdcaYang(perm sk)` = `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]);
```

$$\frac{(x1 - x3)\, A[2, 1, 3]}{x1} + \frac{(x2 - x3)\, A[1, 3, 2]}{x2} + A[1, 2, 3] + \frac{(x1 - x3)\,(x2 - x3)\, A[3, 1, 2]}{x1\, x2}$$

SEE ALSO:

FUNCTION: `Rd2Idca` - calculate a product of generators

CALLING SEQUENCE:

```
Rd2Idca(rd)
Rd2Idca(rd, n)
IDCA[Rd2Idca](rd)
IDCA[Rd2Idca](rd, n)
```

PARAMETERS:

`rd` = any list denoting a decomposition
`n` = any positive integer denoting the degree of a symmetric group

SYNOPSIS:

- The `Rd2Idca` function calculates the product of the generators given by the decomposition `rd`. The product is taken in the idCoxeter algebra.
- If `rd` is reduced then `Rd2Idca(rd)=SG['Rd2Perm'](rd)`.
- When the second argument is not present, `n` is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2Idca` and another name used in the same session, use the long form `IDCA['Rd2Idca']`.

EXAMPLES:

```
> with(IDCA):
> Rd2Idca([1,2,1], 5);
```

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

```
> Rd2Idca([1,2,1]);
```

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

```
> Rd2Idca([1,2,1,2]);
```

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

See Also:

Function: `Table2Idca` - convert a table into an element of the idCoxeter algebra

Calling Sequence:

```
Table2Idca(t)
IDCA[Table2Idca](t)
```

Parameters:

`t` = any table denoting an element of the idCoxeter algebra

Synopsis:

- The `Table2Idca` function converts a table into an element of the idCoxeter algebra. The table should be indexed by permutations and each entry corresponding to a permutation `perm` is interpreted as the coefficient of `perm` in the expression of the element `e` in the basis of permutations.
- Whenever there is a conflict between the function name `Table2Idca` and another name used in the same session, use the long form `IDCA['Table2Idca']`.

Examples:

```
> with(IDCA):
> t:=table([[2,3,1]=x1+x2, [1,3,2]=x1^2, [1,2,3]=1]);
```

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

```
> Table2Idca(t);
```

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

See Also: `Idca2Table`

5.8 Working in the generic Hecke algebra with HEKA

HELP FOR: A package for the 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:

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

- 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 `A[perm]`, where `perm` is a permutation, 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:

`Heka2Table`	`HekaAdd`	`HekaCharTable`	`HekaJucis`
`HekaMinus`	`HekaMult`	`HekaNormal`	`HekaOnPol`
`HekaOnX`	`HekaOnXfix`	`HekaYang`	`Rd2Heka`
`Table2Heka`			

- For help with a particular function do either ?`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*A[3,1,2]:
> b:=t*A[2,1,3] - A[1,3,2]:
> HekaMult(a, b);

  q t A[2, 1, 3] + (- q1 q2 u - q) A[1, 3, 2] + (q1 u + q2 u) A[3, 1, 2]

           u A[3, 2, 1]
         - ------------
                t
```

FUNCTION: `Heka2Table` - convert an element of the Hecke algebra into a table

CALLING SEQUENCE:

```
Heka2Table(e)
HEKA[Heka2Table](e)
```

PARAMETERS:

`e` = any element of the Hecke algebra

SYNOPSIS:

- The `Heka2Table` function converts an element of the Hecke algebra into a table. The table is indexed by permutations and each entry corresponding to a permutation `perm` is the coefficient of `perm` in `e`.
- This function can be useful if one wants to act either on coefficients or on permutations. For instance, one can use the `map` function to act on coefficients.
- Whenever there is a conflict between the function name `Heka2Table` and another name used in the same session, use the long form `HEKA['Heka2Table']`.

EXAMPLES:

```
> with(HEKA):
> t := Heka2Table((x1+x2)*A[2,3,1] + x1^2*A[1,3,2] + A[1,2,3]);

                        t := table(sparse,[
                               [2, 3, 1] = x1 + x2
                                                2
                               [1, 3, 2] = x1
                               [1, 2, 3] = 1
                             ])
> t := map(SP['ToX'], t);

                       t := table(sparse,[
                              [2, 3, 1] = X[1, 3, 2]
                              [1, 3, 2] = X[3, 1, 2]
                              [1, 2, 3] = X[1]
                            ])

> Table2Heka(t);

     X[1, 3, 2] A[2, 3, 1] + X[3, 1, 2] A[1, 3, 2] + X[1] A[1, 2, 3]
```

SEE ALSO: `Table2Heka` `SP[ToX]`

FUNCTION: `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*A[3,1,2]:
> b:=t - A[1,3,2]:
> HekaAdd(a, b);
```

$$(q + t)\, A[1, 2, 3] - A[1, 3, 2] + \frac{u\, A[3, 1, 2]}{t}$$

```
> a &?+ b;
```

$$(q + t)\, A[1, 2, 3] - A[1, 3, 2] + \frac{u\, A[3, 1, 2]}{t}$$

SEE ALSO:

FUNCTION: `HekaCharTable` - compute the table of characters

CALLING SEQUENCE:
`HekaCharTable(n)`
`HekaCharTable(n, 'index')`
`HekaCharTable(n, part_1, part_2)`
`HEKA[HekaCharTable](n)`
`HEKA[HekaCharTable](n, 'index')`
`HEKA[HekaCharTable](n, part_1, part_2)`

PARAMETERS:
`n` = any positive integer
`part_1, part_2` = any lists denoting partitions

SYNOPSIS:

- The `HekaCharTable` gives the table of values of irreducible characters on special elements of the Hecke algebra. To every partition (of `n`) `part_2 = [a, b, c, ...]` one associates a permutation `perm` which is a direct product of cycles. The code of `perm` is `[a-1, 0$(a-1), b-1, 0$(b-1), c-1, 0$(c-1), ...]`. Then, `HekaCharTable(n, part_1, part_2)` gives the value of the irreducible character indexed by `part_1` on the element `part_2`.
- `HekaCharTable(n)` gives the matrix of `HekaCharTable(n, part_1, part_2)` for all partitions of `n` (`HekaCharTable(n, 'index')`). Partitions indexing rows and columns are also given by `PART[ListPart](n)`.
- 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 `HekaCharTable` and another name used in the same session, use the long form `HEKA['HekaCharTable']`.

EXAMPLES:

```
> with(HEKA):
> HekaCharTable(3);
```

$$\begin{bmatrix} 2 & & \\ q1 & q1 & 1 \\ & & \\ q2\ q1 & q1+q2 & 2 \\ & & \\ 2 & & \\ q2 & q2 & 1 \end{bmatrix}$$

```
> HekaCharTable(3, 'index');

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

> HekaCharTable(3, [2,1], [3]);

                                q2 q1
```

SEE ALSO: `TYP[IsPart]` `PART[ListPart]`

FUNCTION: `HekaJucis` - Jucis-Murphy element of the Hecke algebra

CALLING SEQUENCE:

`HekaJucis(i)`
`HEKA[HekaJucis](i)`

PARAMETERS:

`i` = any positive integer

SYNOPSIS:

- The `HekaJucis` function computes the `i`-th Jucis-Murphy element inside the Hecke algebra of the symmetric group of degree `n` (`i`≤`n`), where `n` is taken to be `i` by default or the second argument if available.
- The algebra generated by the Jucis-Murphy elements is a maximal commutative sub-algebra of the Hecke algebra.
- More explicitly, denoting `T(i,j)` the transposition exchanging `i` and `j`, we have: `HekaJucis(1) = 0`, `HekaJucis(2) = (q1+q2)*T(1,2) - q1*q2`, `HekaJucis(3) = (q1+q2)T(1,3) - (q1 + q2)*q1*q2*T(2,3) + q1^2*q2^2` and `HekaJucis(4) = (q1+q2)T(1,4) - (q1+q2)*q1*q2 * T(2,4) + (q1+q2)*q1^2*q2^2 T(3,4) - q1^3*q2^3`.
- Whenever there is a conflict between the function name `HekaJucis` and another name used in the same session, use the long form `HEKA['HekaJucis']`.

EXAMPLES:

```
> with(HEKA):
> HekaJucis(2);

                    (q1 + q2) A[2, 1] - q1 q2 A[1, 2]

> HekaJucis(4, 6);

    (q1 + q2) A[4, 2, 3, 1, 5, 6] - (q1 + q2) q1 q2 A[1, 4, 3, 2, 5, 6]

                2   2                           3   3
  + (q1 + q2) q1  q2  A[1, 2, 4, 3, 5, 6] - q1  q2  A[1, 2, 3, 4, 5, 6]
```

SEE ALSO: `SG[SgTranspo]`

FUNCTION: `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*A[3,1,2]:
> b:=t - A[1,3,2]:
> HekaMinus(a, b);
```

$$(q - t)\,A[1, 2, 3] + A[1, 3, 2] + \frac{u\,A[3, 1, 2]}{t}$$

```
> a &?- a;
```

$$0$$

SEE ALSO:

FUNCTION: `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*A[3,1,2]:
> b:=t*A[2,1,3] - A[1,3,2]:
> HekaMult(a, b);
```

$$-\frac{u\,A[3, 2, 1]}{t} + (-q - q1\,q2\,u)\,A[1, 3, 2] + q\,t\,A[2, 1, 3] + (q1\,u + q2\,u)\,A[3, 1, 2]$$

```
> a &?* b;
```

```
      u A[3, 2, 1]
- ------------ + (- q - q1 q2 u) A[1, 3, 2] + q t A[2, 1, 3]
        t

    + (q1 u + q2 u) A[3, 1, 2]
```

See Also:

Function: `HekaNormal` - normalize an element of the Hecke algebra

Calling Sequence:

`HekaNormal(e)`
`HekaNormal(e, m)`
`HEKA[HekaNormal](e)`
`HEKA[HekaNormal](e, m)`

Parameters:

`e` = any element of the Hecke algebra
`m` = any positive integer denoting the degree of a symmetric group

Synopsis:

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

Examples:

```
> with(HEKA):
> HekaNormal(a*A[3,1,2] - b^2*A[2,1,3,4]);

                          2
                    - b  A[2, 1] + a A[3, 1, 2]

> HekaNormal(a*A[3,1,2] - b^2*A[2,1,3,4], 3);

                          2
                    - b  A[2, 1, 3] + a A[3, 1, 2]
```

See Also:

Function: 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 `SgTranspo(i,n)` is :

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

  where `exp'` is the image of `exp` under `SgTranspo(i,n)`.
- By default the algebra acts on the variables $x_1, x_2, x_3, ...$
- When called with a third argument `v`, being, say '`y`', the `HekaOnPol` function acts on the variables $y_1, y_2, y_3, ...$
- 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(A[2,1], 1);
                                    q1

> HekaOnPol(A[2,1], x1);

                          x2 q1 + x1 q2 + q1 x1

> HekaOnPol(a^2*A[3,1,2], x1^2*x2*y1*y2^3);

     2    2   2        2                                        2
    a  (x3  q1  + x3 q1  x1 + x2 q2 x3 q1 + x3 x1 q2 q1 + x3 q1  x2

                         2           2        2           2   2          3
+ 2 x2 x1 q2 q1 + x1 q2  x2 + x2 q1  x1 + x2  q1 q2 + x2  q1 ) x1 y1 y2
```

```
> HekaOnPol(a^2*A[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:

Function: `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 expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression exp before treating it.
- One may collect the result by adding a third argument: this is done by `HekaOnX(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can use `HekaOnX(e_1, exp, 'noexpand', 'collect')`.
- The action of the element `SgTranspo(i,n)` on the Schubert polynomial `X[perm]` is : let `nu = MultPerm(perm, SgTranspo(i,n))`, then

  `X[perm]` $\longrightarrow$ `q1 X[perm]` if `length(nu)` $>$ `length(perm)`
  `X[perm]` $\longrightarrow$ `q1 X[perm] + (q2 x_i + q1 x_i+1) X[nu]` otherwise.
- 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*A[1,3,2] - a^3*A[2,1], z*X[3,2,1] - X[2,1,3]);

                      3                        3                     4
   - q2 X[3, 2, 1] z a  - q1 X[2, 4, 1, 3] z a  + q2 X[3, 2, 1] z a
```

```
                                4                3
      + q1 X[3, 1, 4, 2] z a  + a  + q2 X[2, 1] a

                              3    4
        + q1 X[1, 3, 2] a  - a  q1 X[2, 1]
```

See Also:

Function: HekaOnXfix - action of an element of the Hecke algebra of the symmetric group on the ring of polynomials as a free module over **Sym**

Calling Sequence:

```
HekaOnXfix(e_1, exp)
HEKA[HekaOnXfix](e_1, exp)
```

Parameters:

`e_1` = any element of the Hecke algebra of the symmetric group
`exp` = any expression

Synopsis:

- The `HekaOnXfix` function realizes the action of an element of the Hecke algebra of the symmetric group, say `e_1`, on an expression `exp` expressed on the **X** Schubert basis with coefficients that are symmetric polynomials in the basis of Schur functions.
- The expression `exp` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `exp` to choose not to expand the expression `exp` before treating it.
- One may collect the result by adding a third argument: this is done by `HekaOnXfix(e_1, exp, 'collect')`. Moreover, one can use both `noexpand` and `collect` options: for instance, one can use `HekaOnXfix(e_1, exp, 'noexpand', 'collect')`.
- The action of the element `SgTranspo(i,n)` on the Schubert polynomial `X[perm]` is : let `nu = MultPerm(perm, SgTranspo(i,n))`, then

  `X[perm]` $\longrightarrow$ `q1 X[perm]` if `length(nu)` $>$ `length(perm)`
  `X[perm]` $\longrightarrow$ `q1 X[perm] + (q2 x_i + q1 x_i+1) X[nu]` otherwise.
- The result is expressed on the **X** Schubert basis corresponding to a given symmetric group and is not collected.
- Whenever there is a conflict between the function name `HekaOnXfix` and another name used in the same session, use the long form `HEKA['HekaOnXfix']`.

Examples:

```
> with(HEKA):
> _FMn;
```

```
> HekaOnXfix(q*A[1,3,2], s[1,1]*X[2,4,1,3], collect);

  q2 X[2, 4, 1, 3] s[1, 1] q + (s[2, 1] + s[1, 1, 1]) q1 q X[2, 1, 4, 3]

     + q2 X[2, 3, 4, 1] s[1, 1] q - q1 X[4, 1, 2, 3] s[1, 1] q

     - (s[2, 2] + s[2, 1, 1] + s[1, 1, 1, 1]) q1 q X[2, 1, 3, 4]
```

See Also:

Function: `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 `ListPerm(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.
- When called with a second parameter, say `'y'`, one specifies that coefficients are in the y_i's.
- This basis is defined by the recursion: for a simple transposition `sk` and a permutation `perm`, such that `length(perm sk) > length(perm)`, then `HekaYang(perm sk) = HekaYang(perm) &?* (1 + (x_j/x_i - 1)/(q_1 + q_2)τ_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([2,3,1]);

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

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

See Also:

Function: `Rd2Heka` - calculate a product of generators

Calling Sequence:

```
Rd2Heka(rd)
Rd2Heka(rd, n)
HEKA[Rd2Heka](rd)
HEKA[Rd2Heka](rd, n)
```

Parameters:

`rd` = any list denoting a decomposition
`n` = any positive integer denoting the degree of a symmetric group

Synopsis:

- The `Rd2Heka` function calculates the product of the generators given by the decomposition `rd`. The product is taken in the Hecke algebra.
- If `rd` is reduced then `Rd2Heka(rd)=SG['Rd2Perm'](rd)`, and otherwise it involves parameters `q1` and `q2`.
- When the second argument is not present, `n` is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2Heka` and another name used in the same session, use the long form `HEKA['Rd2Heka']`.

Examples:

```
> with(HEKA):
> Rd2Heka([1,2,1], 5);

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

> Rd2Heka([1,2,1]);

                                 A[3, 2, 1]

> Rd2Heka([1,2,1,2]);

                  - q1 q2 A[3, 1, 2] + (q1 + q2) A[3, 2, 1]
```

See Also: `SG[Rd2Perm]`

Function: `Table2Heka` - convert a table into an element of the Hecke algebra

Calling Sequence:

```
Table2Heka(t)
HEKA[Table2Heka](t)
```

PARAMETERS:

`t` = any table denoting an element of the Hecke algebra

SYNOPSIS:

- The `Table2Heka` function converts a table into an element of the Hecke algebra. The table should be indexed by permutations and each entry corresponding to a permutation `perm` is interpreted as the coefficient of `perm` in the expression of the element `e` in the basis of permutations.
- Whenever there is a conflict between the function name `Table2Heka` and another name used in the same session, use the long form `HEKA['Table2Heka']`.

EXAMPLES:

```
> with(HEKA):
> t:=table([[2,3,1]=x1+x2, [1,3,2]=x1^2, [1,2,3]=1]);
```

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

```
> Table2Heka(t);
```

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

SEE ALSO: `Heka2Table`

5.9 CG: a package about classical Lie groups

HELP FOR: A package on Classical Lie Groups.

CALLING SEQUENCE:
```
CG[<function>](args)
<function>(args)
```

SYNOPSIS:

- This package provides functions to work with classical Lie groups.
- Two extra bases of the ring of symmetric functions are provided. They are stored in the global variable `CGBases := {'O', 'Sp'}` and correspond to the orthogonal and the symplectic Schur function bases.
- The available functions are:

CgMultiplicity	CgPositiveRoots	CgRho	CgSchurSeries
O2s	ODimRep	OJtMat	Sp2s
SpDimRep	SpJtMat	ToO	ToSp

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

EXAMPLES:

```
> with(CG):
> ToO(s[2,1]*O[3]);

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

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

FUNCTION: `CgMultiplicity` - calculate weight multiplicities

CALLING SEQUENCE:
```
CgMultiplicity(typ, lambda, mu)
CG[CgMultiplicity](typ, lambda, mu)
```

PARAMETERS:
`typ` = one of the classical group `A`, `B`, `C` or `D`
`lambda, mu` = any lists denoting partitions

SYNOPSIS:

- The `CgMultiplicity` function calculates the multiplicity `K(lambda, mu)` of the weight `mu` in the irreducible module (of the simple Lie algebra of type `typ`) with highest weight `lambda`, using the Freudenthal formula.
- For type `A`, `K(lambda, mu)` is the corresponding Kostka number.
- Whenever there is a conflict between the function name `CgMultiplicity` and another name used in the same session,use the long form `CG['CgMultiplicity']`.

EXAMPLES:

```
> with(CG):
> CgMultiplicity(A, [9,1], [5,3,2]);
```

$$2$$

SEE ALSO: `ODimRep SpDimRep`

FUNCTION: `CgPositiveRoots` - positive roots of a classical group

CALLING SEQUENCE:

```
CgPositiveRoots(typ, rank)
CG[CgPositiveRoots](typ, rank)
```

PARAMETERS:

`typ` = one of the classical group `A`, `B`, `C` or `D`
`rank` = the rank of the group

SYNOPSIS:

- The `CgPositiveRoots` function computes the positive roots of any classical group.
- The result is a list of lists.
- Whenever there is a conflict between the function name `CgPositiveRoots` and another name used in the same session, use the long form `CG['CgPositiveRoots']`.

EXAMPLES:

```
> with(CG):
> CgPositiveRoots(D, 4);
```

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

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

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

See Also: CgMultiplicity

Function: CgRho - half the sum of positive roots

Calling Sequence:
```
CgRho(typ, rank)
CG[CgRho](typ, rank)
```

Parameters:
typ = one of the classical group A, B, C or D
rank = the rank of the group

Synopsis:

- The CgRho function computes half the sum of positive roots.
- The result is a list.
- Whenever there is a conflict between the function name CgRho and another name used in the same session, use the long form CG['CgRho'].

Examples:

```
> with(CG):
> CgRho(B, 4);
                              [1/2, 3/2, 5/2, 7/2]
```

See Also: CgMultiplicity CgPositiveRoots

Function: CgSchurSeries - truncated Schur function series

Calling Sequence:
```
CgSchurSeries(typ, rank)
CG[CgSchurSeries](typ, rank)
```

Parameters:
typ = the type of the Schur function series
rank = the rank of truncature

Synopsis:

- The CgSchurSeries function computes the Schur function series of a given type truncated to the given rank. The available types are A, B, C and D, according to the notations of King and Wybourne.
- The result is a linear combination of Schur functions.
- Whenever there is a conflict between the function name CgSchurSeries and another name used in the same session, use the long form CG['CgSchurSeries'].

Examples:

```
> with(CG):
> CgSchurSeries(A, 10);

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

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

See Also: `PART[Part2Frob]`

Function: `O2s` - from the orthogonal Schur basis to the Schur basis

Calling Sequence:

```
O2s(sf)
CG[O2s](sf)
```

Parameters:

`sf` = any element in the orthogonal Schur basis

Synopsis:

- The `O2s` function converts any element expressed in the orthogonal Schur basis into an element expressed in the basis of Schur functions.
- The result is a linear combination of Schur functions.
- Whenever there is a conflict between the function name `O2s` and another name used in the same session, use the long form `CG['O2s']`.

Examples:

```
> with(CG):
> O2s(q*O[2,1] - O[1,1,1]);

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

See Also: `OJtMat`

FUNCTION: ODimRep - calculate the dimension of an irreducible representation of the orthogonal group

CALLING SEQUENCE:
```
ODimRep(rk, lambda)
CG[ODimRep](rk, lambda)
```

PARAMETERS:
`rk` = the rank of `O`
`lambda` = any list denoting a partition

SYNOPSIS:

- The `ODimRep` function calculates the dimension of the irreducible representation of the orthogonal group, with highest weight represented by a partition `lambda`.
- Whenever there is a conflict between the function name `ODimRep` and another name used in the same session, use the long form `CG['ODimRep']`.

EXAMPLES:

```
> with(CG):
> ODimRep(6, [4,2,2]);
                                  720
```

SEE ALSO: SpDimRep

FUNCTION: OJtMat - Jacobi-Trudi-type matrix

CALLING SEQUENCE:
```
OJtMat(part)
CG[OJtMat](part)
```

PARAMETERS:
`part` = any list denoting a partition

SYNOPSIS:

- The `OJtMat` function computes the Jacobi-Trudi-type matrix which corresponds to the partition `part`. Its determinant gives the expression of the orthogonal Schur function `O[part]` in the `h`-basis.
- Whenever there is a conflict between the function name `OJtMat` and another name used in the same session, use the long form `CG['OJtMat']`.

EXAMPLES:

```
> with(CG):
> OJtMat([7,4,3]);
```

$$\begin{bmatrix} h7 - h5 & h8 - h4 & h9 - h7 + h5 - h3 \\ h3 - h1 & h4 - 1 & h5 - h3 + h1 \\ h1 & h2 & h3 - h1 \end{bmatrix}$$

See Also: O2s

Function: Sp2s - from the symplectic Schur basis to the Schur basis

Calling Sequence:

Sp2s(sf)
CG[Sp2s](sf)

Parameters:

sf = any element in the symplectic Schur basis

Synopsis:

- The Sp2s function converts any element expressed in the symplectic Schur basis into an element expressed in the basis of Schur functions.
- The result is a linear combination of Schur functions.
- Whenever there is a conflict between the function name Sp2s and another name used in the same session, use the long form CG['Sp2s'].

Examples:

```
> with(CG):
> Sp2s(Sp[6,3,2,1] - Sp[1,1,1]);
```

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

See Also:

Function: SpDimRep - calculate the dimension of an irreducible representation of the symplectic group

Calling Sequence:
SpDimRep(lambda)
CG[SpDimRep](lambda)

Parameters:
lambda = any list denoting a partition

Synopsis:

- The SpDimRep function calculates the dimension of the irreducible representation of the symplectic group, with highest weight represented by a partition lambda.
- Whenever there is a conflict between the function name SpDimRep and another name used in the same session, use the long form CG['SpDimRep'].

Examples:

```
> with(CG):
> SpDimRep([4,2,2]);
                                1078
```

See Also: ODimRep

Function: SpJtMat - Jacobi-Trudi-type matrix

Calling Sequence:
SpJtMat(part)
CG[SpJtMat](part)

Parameters:
part = any list denoting a partition

Synopsis:

- The SpJtMat function computes the Jacobi-Trudi-type matrix which corresponds to the partition part. Its determinant gives the expansion of the symplectic Schur function Sp[part] in the h-basis.
- Whenever there is a conflict between the function name SpJtMat and another name used in the same session, use the long form CG['SpJtMat'].

Examples:

```
> with(CG):
> SpJtMat([7,4,3]);
                          [ h7  h8 + h6  h9 + h5 ]
                          [                      ]
                          [ h3  h4 + h2  h5 + h1 ]
                          [                      ]
                          [ h1   h2 + 1     h3   ]
```

See Also: Sp2s

Function: ToO - express any symmetric function in the basis of orthogonal Schur functions

Calling Sequence:

```
ToO(sf)
ToO(sf, b)
CG[ToO](sf)
CG[ToO](sf, b)
```

Parameters:

`sf` = any symmetric function
`b` = any name of a known basis

Synopsis:

- The `ToO` function expresses any symmetric function in the basis of orthogonal Schur functions, i.e. as a linear combination of orthogonal Schur functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function `sf` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `sf` is solely expressed in terms of the known basis `b`.
- One may add `'noexpand'` just after the argument `sf` to choose not to expand the symmetric function `sf` before treating it.
- One may collect the result by adding a third argument: this is done by `ToO(sf, b, 'collect')`. For instance, `ToO(sf, 'O', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `ToO` and another name used in the same session, use the long form `CG['ToO']`.

Examples:

```
> with(CG):
> ToO(h4+2*e2*s[3,1]);

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

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

See Also: ToSp

FUNCTION: ToSp - express any symmetric function in the basis of symplectic Schur functions

CALLING SEQUENCE:

```
ToSp(sf)
ToSp(sf, b)
CG[ToSp](sf)
CG[ToSp](sf, b)
```

PARAMETERS:

`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `ToSp` function expresses any symmetric function in the basis of symplectic Schur functions, i.e. as a linear combination of symplectic Schur functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function `sf` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `sf` is solely expressed in terms of the known basis `b`.
- One may add `'noexpand'` just after the argument `sf` to choose not to expand the symmetric function `sf` before treating it.
- One may collect the result by adding a third argument as in `ToSp(sf, b, 'collect')`. For instance, `ToSp(sf, 'Sp', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `ToSp` and another name used in the same session, use the long form `CG['ToSp']`.

EXAMPLES:

```
> with(CG):
> ToSp(e3*s[3,2]);

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

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

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

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

SEE ALSO: ToO

5.10 BN and the hyperoctahedral groups

HELP FOR: A package for the hyperoctahedral group.

CALLING SEQUENCE:

```
BN[<function>](args)
<function>(args)
```

SYNOPSIS:

- This package provides functions to work with the hyperoctahedral group, in particular with its Bruhat order. It gives functions to handle the elements of this group, represented by codes. It also provides functions to make conversions between a code and a signed permutation (`Bar`) or a palindromic permutation (`Palin`). This last coding comes from the embedding of B_n into $\mathfrak{S}_{2n}$.

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

- The available functions are:

BnBar2Code	BnBar2Mat	BnCode2Bar	BnCode2Inv
BnCode2Interval	BnCode2Length	BnCode2ListRd	BnCode2Palin
BnCode2Rd	BnGenCode	BnListCode	BnMat2Bar
BnMultBar	BnMultCode	BnPalin2Code	BnRd2Code
BnTranspoPalin			

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

EXAMPLES:

```
> with(BN):
> BnCode2Interval([1,1,3]);

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

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

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

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

FUNCTION: BnBar2Code - code of a signed permutation

CALLING SEQUENCE:

```
BnBar2Code(bar)
BN[BnBar2Code](bar)
```

PARAMETERS:

`bar` = any list denoting a signed permutation

SYNOPSIS:

- The `BnBar2Code` function calculates the code of a signed permutation `bar`.
- Whenever there is a conflict between the function name `BnBar2Code` and another name used in the same session, use the long form `BN['BnBar2Code']`.

EXAMPLES:

```
> with(BN):
> BnBar2Code([-3, 2, -1]);
```

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

SEE ALSO: `BnCode2Bar` `TYP[IsBnCode]`

FUNCTION: BnBar2Mat - signed permutation matrix from a signed permutation

CALLING SEQUENCE:

```
BnBar2Mat(bar)
BN[BnBar2Mat](bar)
```

PARAMETERS:

`bar` = any list denoting a signed permutation

SYNOPSIS:

- The `BnBar2Mat` function calculates the matrix corresponding to the signed permutation `bar`. This matrix has entries either 1 or -1 in position `(i, abs(bar[i]))` and 0 otherwise. The sign of the entry in the `i`-th row, is the sign of `bar[i]`.
- Whenever there is a conflict between the function name `BnBar2Mat` and another name used in the same session, use the long form `BN['BnBar2Mat']`.

EXAMPLES:

```
> with(BN):
> BnBar2Mat([-3, 2, -1]);
```

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

See Also: `BnMat2Bar`

Function: `BnCode2Bar` - signed permutation corresponding to a code

Calling Sequence:

```
BnCode2Bar(code)
BN[BnCode2Bar](code)
```

Parameters:

`code` = any list denoting a code

Synopsis:

- The `BnCode2Bar` function calculates the signed permutation that corresponds to a `code`.
- Whenever there is a conflict between the function name `BnCode2Bar` and another name used in the same session, use the long form `BN['BnCode2Bar']`.

Examples:

```
> with(BN):
> BnCode2Bar([1, 1, 3]);
```

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

See Also: `BnBar2Code TYP[IsBnBar]`

Function: `BnCode2Interval` - list of codes of all group elements below a given one

Calling Sequence:

```
BnCode2Interval(code)
BN[BnCode2Interval](code)
```

Parameters:

`code` = any list denoting a code

SYNOPSIS:

- The `BnCode2Interval` function computes the list of all codes of group elements that are smaller than the given `code`, according to the Bruhat order.
- Whenever there is a conflict between the function name `BnCode2Interval` and another name used in the same session, use the long form `BN['BnCode2Interval']`.

EXAMPLES:

```
> with(BN):
> BnCode2Interval([1, 0, 3]);

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

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

SEE ALSO:

FUNCTION: `BnCode2Inv` - code of the inverse of a group element

CALLING SEQUENCE:

```
BnCode2Inv(code)
BN[BnCode2Inv](code)
```

PARAMETERS:

`code` = any list denoting a code

SYNOPSIS:

- The `BnCode2Inv` function returns the code of the inverse of the group element given by its `code`.
- Whenever there is a conflict between the function name `BnCode2Inv` and another name used in the same session, use the long form `BN['BnCode2Inv']`.

EXAMPLES:

```
> with(BN):
> BnCode2Inv([1, 0, 3, 3, 1, 5]);

                              [0, 1, 3, 4, 1, 4]
> BnCode2Inv([0, 1, 3, 4, 1, 4]);

                              [1, 0, 3, 3, 1, 5]
```

See Also:

Function: `BnCode2Length` - length of a code

Calling Sequence:

```
BnCode2Length(code)
BN[BnCode2Length](code)
```

Parameters:

`code` = any list denoting a code

Synopsis:

- The `BnCode2Length` function calculates the length of an element of the group given by its `code`. It is equal to the minimal length of an expression of the group element as a product of simple reflections. It is also the sum of the components of the `code`.
- Whenever there is a conflict between the function name `BnCode2Length` and another name used in the same session, use the long form `BN['BnCode2Length']`.

Examples:

```
> with(BN):
> BnCode2Length([0,1,0,5,4]);
```

```
                                10
```

See Also: `BnCode2Rd`

Function: `BnCode2ListRd` - list of all reduced decompositions of a code

Calling Sequence:

```
BnCode2ListRd(code)
BN[BnCode2ListRd](code)
```

Parameters:

`code` = any list denoting a code

Synopsis:

- The `BnCode2ListRd` function computes the list of all reduced decompositions of a given `code`. The special generator is denoted 0.
- When called with the second argument `'nb'`, it computes the number of all reduced decompositions of a given `code`. It does not generate all reduced decompositions but only counts their number.
- Whenever there is a conflict between the function name `BnCode2ListRd` and another name used in the same session, use the long form `BN['BnCode2ListRd']`.

Examples:

```
> with(BN):
> BnCode2ListRd([1, 1, 5]);

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

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

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

> BnCode2ListRd([1, 1, 5], 'nb');

                                    9
```

See Also: BnCode2Rd BnRd2Code

Function: BnCode2Palin - palindromic permutation corresponding to a code

Calling Sequence:

```
BnCode2Palin(code)
BN[BnCode2Palin](code)
```

Parameters:

code = any list denoting a code

Synopsis:

- The BnCode2Palin function calculates the palindromic permutation w corresponding to a code. This permutation belongs to the symmetric group of degree $2n$, where n is the degree of the hyperoctahedral group B_n, and satisfies the symmetries:

$$w[i] + w[2*n-i+1] = 2*n+1, \quad i=1..n$$

- Whenever there is a conflict between the function name BnCode2Palin and another name used in the same session, use the long form BN['BnCode2Palin'].

Examples:

```
> with(BN):
> BnCode2Palin([1, 1, 3]);

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

See Also: `BnPalin2Code` `TYP[IsBnPalin]`

Function: `BnCode2Rd` - canonical reduced decomposition of a code

Calling Sequence:

```
BnCode2Rd(code)
BN[BnCode2Rd](code)
```

Parameters:

`code` = any list denoting a code

Synopsis:

- The `BnCode2Rd` function computes the canonical reduced decomposition of a given `code` (i.e. the lexicographic first one). The special generator is denoted 0.
- Whenever there is a conflict between the function name `BnCode2Rd` and another name used in the same session, use the long form `BN['BnCode2Rd']`.

Examples:

```
> with(BN):
> BnCode2Rd([1, 1, 5]);
```

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

See Also: `BnCode2ListRd` `BnRd2Code`

Function: `BnGenCode` - generate one code

Calling Sequence:

```
BnGenCode(n, typecode)
BnGenCode(code, typecode)
BN[BnGenCode](n, typecode)
BN[BnGenCode](code, typecode)
```

Parameters:

`n` = any positive integer
`code` = any list denoting a code
`typecode` = `'first'`, `'previous'`, `'next'`, `'last'`

Synopsis:

- The `BnGenCode` function generates one element of a hyperoctahedral group in lexicographic order, according to the parameter `typecode`.
- `BnGenCode(n, 'first')` returns the first code of B_n.
- `BnGenCode(n, 'last')` returns the last code of B_n.

- `BnGenCode(code, 'previous')` gives the previous code according to the lexicographic order. It returns the `NULL` value when `code` is already the first code `[0, 0, 0, ...]`.
- `BnGenCode(code, 'next')` gives the next code according to the lexicographic order. It returns the `NULL` value when `code` is already the last code `[1, 3, 5, ...]`.
- Whenever there is a conflict between the function name `BnGenCode` and another name used in the same session, use the long form `BN['BnGenCode']`.

Examples:

```
> with(BN):
> BnGenCode([1, 2, 5, 7], 'next');

                              [1, 3, 0, 0]

> BnGenCode([1, 3, 0, 0], 'previous');

                              [1, 2, 5, 7]
```

See Also:

Function: `BnListCode` - generate all codes

Calling Sequence:

```
BnListCode(n)
BN[BnListCode](n)
```

Parameters:

`n` = any positive integer

Synopsis:

- The `BnListCode` function generates all the $n!\,2^n$ codes of the hyperoctahedral group B_n.
- Whenever there is a conflict between the function name `BnListCode` and another name used in the same session, use the long form `BN['BnListCode']`.

Examples:

```
> with(BN):
> BnListCode(2);

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

See Also: TYP[IsBnCode]

Function: BnMat2Bar - calculate the signed permutation from a given permutation matrix

Calling Sequence:

```
BnMat2Bar(mat)
BN[BnMat2Bar](mat)
```

Parameters:

`mat` = any signed permutation matrix

Synopsis:

- The `BnMat2Bar` function calculates the signed permutation corresponding to a given permutation matrix. This matrix has entries either 1 or -1 in position `(i, abs(bar[i]))` and 0 otherwise. The sign of the entry in the `i`-th row, is the sign of `bar[i]`.
- Whenever there is a conflict between the function name `BnMat2Bar` and another name used in the same session, use the long form `BN['BnMat2Bar']`.

Examples:

```
> with(BN):
> BnBar2Mat([-3, 2, -1]);
```

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

```
> BnMat2Bar( BnBar2Mat([-3, 2, -1]) );

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

See Also: BnBar2Mat

Function: BnMultBar - multiply signed permutations

Calling Sequence:

```
BnMultBar(bar_1, bar_2)
BN[BnMultBar](bar_1, bar_2)
```

Parameters:

`bar_1, bar_2` = any lists denoting signed permutations

Synopsis:

- The `BnMultBar` function calculates the product of a sequence (of arbitrary length) of signed permutations.

- `BnMultBar(bar_1, bar_2)` computes `bar_1(bar_2)`.
- Signed permutations need not to belong to the same hyperoctahedral group.
- Whenever there is a conflict between the function name `BnMultBar` and another name used in the same session, use the long form `BN['BnMultBar']`.

EXAMPLES:

```
> with(BN):
> BnMultBar([-1, 3, -2], [-2, 3, -1], [1, 3, 2, -4]);

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

SEE ALSO:

FUNCTION: `BnMultCode` - multiply codes

CALLING SEQUENCE:

```
BnMultCode(code_1, code_2)
BN[BnMultCode](code_1, code_2)
```

PARAMETERS:

`code_1, code_2` = any lists denoting codes

SYNOPSIS:

- The `BnMultCode` function computes the product of a sequence (of arbitrary length) of codes.
- `BnMultCode(code_1, code_2)` computes `code_1(code_2)`.
- Codes need not to belong to the same group.
- Whenever there is a conflict between the function name `BnMultCode` and another name used in the same session, use the long form `BN['BnMultCode']`.

EXAMPLES:

```
> with(BN):
> BnMultCode([1, 3, 1], [1, 2, 1], [0, 0, 1, 7]);

                              [0, 3, 3, 7]
```

See Also:

Function: `BnPalin2Code` - code corresponding to a palindromic permutation

Calling Sequence:

```
BnPalin2Code(palin)
BN[BnPalin2Code](palin)
```

Parameters:

`palin` = any list denoting a palindromic permutation

Synopsis:

- The `BnPalin2Code` function calculates the code corresponding to a palindromic permutation. The code belongs to B_n if the permutation belongs to the symmetric group of degree $2n$.
- Whenever there is a conflict between the function name `BnPalin2Code` and another name used in the same session, use the long form `BN['BnPalin2Code']`.

Examples:

```
> with(BN):
> BnPalin2Code([4, 2, 6, 1, 5, 3]);

                              [1, 1, 3]
```

See Also: `BnCode2Palin` `TYP[IsBnCode]`

Function: `BnRd2Code` - calculate a code from a decomposition into simple reflections

Calling Sequence:

```
BnRd2Code(rd)
BnRd2Code(rd, n)
BN[BnRd2Code](rd)
BN[BnRd2Code](rd, n)
```

Parameters:

`rd` = any list denoting a decomposition
`n` = any positive integer

Synopsis:

- The `BnRd2Code` function computes the code corresponding to the list `rd`, considered as a product of simple reflections. The special generator is denoted 0.
- Let `rd=[i,j,k,...]`, then `BnRd2Code(rd)` = `SG[MultPerm](s_i, s_j, s_k, ...)` where `s_i` stands for `BnTranspoPalin(i)`.
- When the second argument is not present, `n` is taken to be the smallest possible one.

- The decomposition needs not to be reduced.
- Whenever there is a conflict between the function name `BnRd2Code` and another name used in the same session, use the long form `BN['BnRd2Code']`.

Examples:

```
> with(BN):
> BnRd2Code([0, 1, 2, 1, 0, 1, 2]);
```

```
                              [1, 1, 5]
```

See Also: `BnCode2Rd`

Function: `BnTranspoPalin` - image of a simple reflection in $\mathfrak{S}_{2n}$

Calling Sequence:

```
BnTranspoPalin(i, n)
BN[BnTranspoPalin](i, n)
```

Parameters:

`n` = any positive integer
`i` = any non negative integer

Synopsis:

- The `BnTranspoPalin` function returns a generator of B_n. The result is an element of B_n embedded into $\mathfrak{S}_{2n}$, as a palindromic permutation.
- `BnTranspoPalin(0, n) = [1, ..., n+1, n, ..., 2*n]`.
- `BnTranspoPalin(i, n) = [1, ..., n-i+1, n-i, ..., n+i+1, n+i, ..., 2*n]`.
- Whenever there is a conflict between the function name `BnTranspoPalin` and another name used in the same session, use the long form `BN['BnTranspoPalin']`.

Examples:

```
> with(BN):
> BnTranspoPalin(1, 3);
```

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

See Also:

5.11 Algebras related to hyperoctahedral groups: package BNA

HELP FOR: A package for algebras related to hyperoctahedral groups.

CALLING SEQUENCE:
`BNA[<function>](args)`
`<function>(args)`

SYNOPSIS:

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

- A general element is expressed as a linear combination of `B[code]`, where code is a code of B_n, with arbitrary coefficients.

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

- The available functions are:

BnGaMult	BnGaOnPol	BnGaYang	BnIdcaMult
BnIdcaOnPol	BnIdcaYang	BnNcaMult	BnNcaOnPol
BnNcaYang	Bna2Table	BnaAdd	BnaCode2Bar
BnaCode2Palin	BnaMinus	BnaNormal	BnaPalin2Code
Rd2BnGa	Rd2BnIdca	Rd2BnNca	Table2Bna

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

- Instead of `BnaAdd`, `BnaMinus`, `BnGaMult`, `BnNcaMult`, `BnIdcaMult`, one can use in short

  `&!!+` `&!!-` `&!!*` `&!@*` `&!$*`

EXAMPLES:

```
> with(BNA):
> BnaCode2Palin(q*B[0,1,0,2] - q^2*B[1,1,0,3]);

                                      2
        q A[2, 4, 1, 3, 6, 8, 5, 7] - q  A[2, 5, 3, 1, 8, 6, 4, 7]
```

FUNCTION: **BnGaMult** - multiplication of two elements of the B_n-group algebra

CALLING SEQUENCE:

```
BnGaMult(e_1, e_2)
BNA[BnGaMult](e_1, e_2)
```

PARAMETERS:

`e_1, e_2` = any elements of the B_n-group algebra

SYNOPSIS:

- The **BnGaMult** function multiplies two elements of the B_n-group algebra.
- One can use instead the operator `&!!*`.
- Whenever there is a conflict between the function name **BnGaMult** and another name used in the same session, use the long form **BNA**['**BnGaMult**'].

EXAMPLES:

```
> with(BNA):
> a:=q + u/t*B[0,1,2]:
> b:=t - B[1,2,1]:
> BnGaMult(a, b);
```

$$-q\,B[1,2,1] - \frac{u\,B[0,2,5]}{t} + q\,t\,B[0,0,0] + u\,B[0,1,2]$$

```
> a &!!* b;
```

$$-q\,B[1,2,1] - \frac{u\,B[0,2,5]}{t} + q\,t\,B[0,0,0] + u\,B[0,1,2]$$

SEE ALSO: **BnIdcaMult** **BnNcaMult**

FUNCTION: **BnGaOnPol** - action of an element of the B_n-algebra

CALLING SEQUENCE:

```
BnGaOnPol(e_1, exp)
BnGaOnPol(e_1, exp, v)
BNA[BnGaOnPol](e_1, exp)
BNA[BnGaOnPol](e_1, exp, v)
```

PARAMETERS:

`e_1` = any element of the B_n-algebra
`exp` = any expression
`v` = any (extra) string

SYNOPSIS:

- The `BnGaOnPol` function realizes the action of an element of the B_n-algebra, say `e_1`, on an expression `exp`. The signed permutation action is to permute variable indices and to change the sign as indicated by the signed permutation.
- By default the algebra acts on the variables $x_1, x_2, x_3, ...$
- When called with a third argument `v`, being, say `'y'`, the `BnGaOnPol` function acts on the variables $y_1, y_2, y_3, ...$
- Whenever there is a conflict between the function name `BnGaOnPol` and another name used in the same session, use the long form `BNA['BnGaOnPol']`.

EXAMPLES:

```
> with(BNA):
> BnGaOnPol(B[1,0,3], x1^3*x2^2*x3);

                                    3   2
                                - x3  x1  x2

> BnGaOnPol(B[1,0,3], x1^3*x2^2*x3, 'y');

                                    3   2
                                  x1  x2  x3
```

SEE ALSO: `BnIdcaOnPol` `BnNcaOnPol`

FUNCTION: `BnGaYang` - Yang-Baxter element of the B_n-group algebra

CALLING SEQUENCE:

```
BnGaYang(code)
BNA[BnGaYang](code)
```

PARAMETERS:

`code` = any list denoting a code

SYNOPSIS:

- The `BnGaYang` function calculates a Yang-Baxter element of the B_n-group algebra.
- The set of `BnGaYang`(code), for all codes in B_n, is a linear basis of the B_n-group algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say `'y'`, one specifies that coefficients are in the y_i's.
- When this second parameter is `'num'` then $x_1, x_2, x_3, ...$ are specialized to 1, 2, 3, ...

- This basis is defined by the recursion: for a simple reflection sk, k>0 and a code, such that length(code sk) > length(code), then one has BnGaYang(code sk) = BnGaYang(code) &!!* $(1 + (sign(\texttt{perm[k]})x_j - sign(\texttt{perm[k+1]})x_i)$ sk) where i=abs(perm[k]), j=abs(perm[k+1]) and perm is the corresponding signed permutation. When k=0, one has: BnGaYang(code s0) = BnGaYang(code) &!!* $(1 - 2 * (sign(\texttt{perm[1]})x_i)$ s0) in which i=abs(perm[1]).
- Whenever there is a conflict between the function name BnGaYang and another name used in the same session, use the long form BNA['BnGaYang'].

Examples:

```
> with(BNA):
> BnGaYang([0,2]);

    (x2 - x1) B[0, 1] + B[0, 0] - 2 (x2 - x1) x2 B[0, 2] - 2 x2 B[1, 0]

> BnGaYang([0,2], 'num');

                  B[0, 1] + B[0, 0] - 4 B[0, 2] - 4 B[1, 0]
```

See Also: BnIdcaYang BnNcaYang

Function: BnIdcaMult - multiplication of two elements of the B_n-idCoxeter algebra

Calling Sequence:

```
BnIdcaMult(e_1, e_2)
BNA[BnIdcaMult](e_1, e_2)
```

Parameters:

e_1, e_2 = any elements of the B_n-idCoxeter algebra

Synopsis:

- The BnIdcaMult function multiplies two elements of the B_n-idCoxeter algebra.
- One can use instead the operator &!$*.
- Whenever there is a conflict between the function name BnIdcaMult and another name used in the same session, use the long form BNA['BnIdcaMult'].

Examples:

```
> with(BNA):
> a:=t - B[1,2,1]:
> b:=q + u/t*B[0,1,2]:
> BnIdcaMult(a, b);
```

```
                                                          u B[1, 3, 2]
      t q B[0, 0, 0] + u B[0, 1, 2] - q B[1, 2, 1] + ------------
                                                             t

> a &!$* b;

                                                          u B[1, 3, 2]
      t q B[0, 0, 0] + u B[0, 1, 2] - q B[1, 2, 1] + ------------
                                                             t
```

See Also: `BnGaMult` `BnNcaMult`

Function: `BnIdcaOnPol` - action of an element of the B_n-idCoxeter algebra

Calling Sequence:

```
BnIdcaOnPol(e_1, exp)
BnIdcaOnPol(e_1, exp, v)
BNA[BnIdcaOnPol](e_1, exp)
BNA[BnIdcaOnPol](e_1, exp, v)
```

Parameters:

`e_1` = any element of the B_n-idCoxeter algebra
`exp` = any expression
`v` = any (extra) string

Synopsis:

- The `BnIdcaOnPol` function realizes the action of an element of the B_n-idCoxeter algebra, say `e_1`, on an expression `exp`.
- The action of the `i`-th elementary generator is:

$$exp \longrightarrow \frac{exp - exp'}{x_i - x_{i+1}} x_{i+1} \quad , \qquad i \neq 0 \quad ,$$

$$exp \longrightarrow \frac{x_1 exp - \frac{1}{x_1} exp''}{1/x_1 - x1} \quad , \qquad i = 0 \quad ,$$

  where `exp'` is the image of `exp` under the exchange of x_i and x_{i+1} and `exp''` is the image of `exp` under the exchange of x_1 and $1/x_1$.
- By default the algebra acts on the variables $x_1, x_2, x_3, ...$
- When called with a third argument `v`, being, say `'y'`, the `BnIdcaOnPol` function acts on the variables $y_1, y_2, y_3, ...$
- Whenever there is a conflict between the function name `BnIdcaOnPol` and another name used in the same session, use the long form `BNA['BnIdcaOnPol']`.

Examples:

```
> with(BNA):
> BnIdcaOnPol(B[1,0,1], x1^3*x2^2*y3^2);

                        6     4     2                   2
                     (x1  + x1  + x1  + 1) (x2 + x3) y3  x3
                   - ---------------------------------------
                                        3
                                      x1

> BnIdcaOnPol(B[1,0,1], x1^3*x2^2*y3^2, `y`);

                                       2   3
                              (y3 + y2) x2  x1  y3
```

See Also: BnGaOnPol BnNcaOnPol

Function: BnIdcaYang - Yang-Baxter element of the B_n-idCoxeter algebra

Calling Sequence:

```
BnIdcaYang(code)
BNA[BnIdcaYang](code)
```

Parameters:

code = any list denoting a code

Synopsis:

- The BnIdcaYang function calculates a Yang-Baxter element of the B_n-idCoxeter algebra.
- The set BnIdcaYang(code), for all codes in B_n is a linear basis of the B_n-idCoxeter algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say 'y', one specifies that coefficients are in the yi's.
- When this second parameter is 'num' then $x_1, x_2, x_3, ...$ are specialized to $q^1, q^2, q^3, ...$

  This basis is defined by the recursion: for a simple reflection sk, k>0 and a code, such that length(code sk) > length(code), then one has BnIdcaYang(code sk) = BnIdcaYang(code) &!$* $(1+(1-(x_{j_1}x_{j_2})/(x_{i_1}x_{i_2}))\pi_k)$ where i_1=palin[n-k], i_2=palin[n+k], j_1=palin[n-k+1], j_2=palin[n+k-1] and palin is the BnCode2Palin(code). If k=0, then the expression becomes BnIdcaYang(code sk) = BnIdcaYang(code) &!$* $(1+(1-(x_j/x_i))\pi_0)$ where i=palin[n] and j=palin[n+1]. π_0, π_1, π_2, ... denote the generators of the B_n-idCoxeter algebra.
- Whenever there is a conflict between the function name BnIdcaYang and another name used in the same session, use the long form BNA['BnIdcaYang'].

EXAMPLES:

```
> with(BNA):
> BnIdcaYang([0,2]);

     (- x1 x3 + x2 x4) B[0, 1]   (- x1 + x4) (- x1 x3 + x2 x4) B[0, 2]
   - ------------------------- + ------------------------------------
               x1 x3                                 2
                                                    x1  x3

           (- x1 + x4) B[1, 0]
         - ------------------- + B[0, 0]
                    x1

> BnIdcaYang([0,2], 'num');

          2                     3          2                   3
   (1 - q ) B[0, 1] + (- 1 + q ) (- 1 + q ) B[0, 2] + (1 - q ) B[1, 0]

   + B[0, 0]
```

SEE ALSO: **BnGaYang BnNcaYang**

FUNCTION: **BnNcaMult** - multiplication of two elements of the B_n-nilCoxeter algebra

CALLING SEQUENCE:

BnNcaMult(e_1, e_2)
BNA[BnNcaMult](e_1, e_2)

PARAMETERS:

e_1, e_2 = any elements of the B_n-nilCoxeter algebra

SYNOPSIS:

- The **BnNcaMult** function multiplies two elements of the B_n-nilCoxeter algebra.
- One can use instead the operator **&!@***.
- Whenever there is a conflict between the function name **BnNcaMult** and another name used in the same session, use the long form **BNA['BnNcaMult']**.

EXAMPLES:

```
> with(BNA):
> a:=t - B[1,2,1]:
> b:=q + u/t*B[0,1,2]:
> BnNcaMult(a, b);

                t q B[0, 0, 0] + u B[0, 1, 2] - q B[1, 2, 1]
```

```
> a &!@* b;
```

```
                    t q B[0, 0, 0] + u B[0, 1, 2] - q B[1, 2, 1]
```

SEE ALSO: BnGaMult BnIdcaMult

FUNCTION: BnNcaOnPol - action of an element of the B_n-nilCoxeter algebra

CALLING SEQUENCE:

```
BnNcaOnPol(e_1, exp)
BnNcaOnPol(e_1, exp, v)
BNA[BnNcaOnPol](e_1, exp)
BNA[BnNcaOnPol](e_1, exp, v)
```

PARAMETERS:

`e_1` = any element of the B_n-nilCoxeter algebra
`exp` = any expression
`v` = any (extra) string

SYNOPSIS:

- The `BnNcaOnPol` function realizes the action of an element of the B_n-nilCoxeter algebra, say `e_1`, on an expression `exp`.
- Simple divided differences ∂_i are the operators:

$$f \longrightarrow \frac{f(\ldots, x_i, x_{i+1}, \ldots) - f(\ldots, x_{i+1}, x_i, \ldots)}{x_i - x_{i+1}} \quad , \qquad i \neq 0 \quad ,$$

$$f \longrightarrow \frac{f(\ldots, x_i, \ldots) - f(\ldots, -x_i, \ldots)}{2x_i} \quad , \qquad i = 0 \quad .$$

- By default the algebra acts on the variables $x_1, x_2, x_3, ...$
- When called with a third argument `v`, being, say '`y`', the `BnNcaOnPol` function acts on the variables $y_1, y_2, y_3, ...$
- Whenever there is a conflict between the function name `BnNcaOnPol` and another name used in the same session, use the long form `BNA['BnNcaOnPol']`.

EXAMPLES:

```
> with(BNA):
> BnNcaOnPol(B[1,0,1], x1^3*x2^2*y3^2);
```

```
                                    2    2
                          (x3 + x2) x1   y3
```

```
> BnNcaOnPol(B[1,0,1], x1^3*x2^2*y3^2, `y`);

                                    0
```

See Also: BnGaOnPol BnIdcaOnPol

Function: BnNcaYang - Yang-Baxter element of the B_n-nilCoxeter algebra

Calling Sequence:

```
BnNcaYang(code)
BNA[BnNcaYang](code)
```

Parameters:

code = any list denoting a code

Synopsis:

- The BnNcaYang function calculates a Yang-Baxter element of the B_n-nilCoxeter algebra.
- The set BnNcaYang(code), for all codes in B_n is a linear basis of the B_n-nilCoxeter algebra, as a free module with coefficients in the x_i's.
- When called with a second parameter, say 'y', one specifies that coefficients are in the y_i's.
- When this second parameter is 'num' then $x_1, x_2, x_3, ...$ are specialized to 1, 2, 3, ...
- This basis is defined by the recursion : for a simple reflection sk, k>0 and a code, such that length(code sk) > length(code), then one has BnNcaYang(code sk) = BnNcaYang(code) &!@* $(1+(sign($perm[k]$)x_j - sign($perm[k+1]$)x_i)\partial_k)$ in which one has i=abs(perm[k]), j=abs(perm[k+1]) and perm is the corresponding signed permutation. When k=0, one has: BnNcaYang(code s0) = BnNcaYang(code) &!@* $(1-2*(sign($perm[1]$)x_i)\partial_0)$ in which i stands for abs(perm[1]). ∂_0, ∂_1, ∂_2, ... denote the generators of the B_n-nilCoxeter algebra.
- Whenever there is a conflict between the function name BnNcaYang and another name used in the same session, use the long form BNA['BnNcaYang'].

Examples:

```
> with(BNA):
> BnNcaYang([0,2]);

  - 2 (x2 - x1) x2 B[0, 2] + (x2 - x1) B[0, 1] - 2 x2 B[1, 0] + B[0, 0]

> BnNcaYang([0,2], 'num');

                - 4 B[0, 2] + B[0, 1] - 4 B[1, 0] + B[0, 0]
```

See Also: BnGaYang BnIdcaYang

Function: Bna2Table - convert an element of any B_n-algebra into a table

Calling Sequence:

```
Bna2Table(e)
BNA[Bna2Table](e)
```

Parameters:

e = any element of any B_n-algebra

Synopsis:

- The Bna2Table function converts an element of any B_n-algebra into a table. The table is indexed by codes of B_n and each entry corresponding to a code cod is the coefficient of B[cod] in e.
- This function can be useful if one wants to act either on coefficients or on codes. For instance, one can use the map function to act on coefficients.
- Whenever there is a conflict between the function name Bna2Table and another name used in the same session, use the long form BNA['Bna2Table'].

Examples:

```
> with(BNA):
> t := Bna2Table((x1+x2)*B[0,2,1] + x1^2*B[0,2,2] + B[0,0,0]);

                        t := table([
                               [0, 2, 1] = x1 + x2
                                                2
                               [0, 2, 2] = x1
                               [0, 0, 0] = 1
                             ])

> t := map(ToX, t);

                       t := table([
                              [0, 2, 1] = X[1, 3, 2]
                              [0, 2, 2] = X[3, 1, 2]
                              [0, 0, 0] = X[1]
                            ])

> Table2Bna(t);

     X[1, 3, 2] B[0, 2, 1] + X[3, 1, 2] B[0, 2, 2] + X[1] B[0, 0, 0]
```

See Also: Table2Bna SP[ToX]

FUNCTION: `BnaAdd` - sum of two elements of any B_n-algebra

CALLING SEQUENCE:

```
BnaAdd(e_1, e_2)
BNA[BnaAdd](e_1, e_2)
```

PARAMETERS:

`e_1, e_2` = any elements of any B_n-algebra

SYNOPSIS:

- The `BnaAdd` function adds two elements of either the B_n-group algebra, B_n-nilCoxeter algebra or the B_n-idCoxeter algebra.
- One can use instead the operator `&!!+`.
- Whenever there is a conflict between the function name `BnaAdd` and another name used in the same session, use the long form `BNA['BnaAdd']`.

EXAMPLES:

```
> with(BNA):
> a:=q + u/t*B[0,1,2]:
> b:=t - B[1,2,1]:
> BnaAdd(a, b);
```

$$- B[1, 2, 1] + \frac{u\, B[0, 1, 2]}{t} + (q + t)\, B[0, 0, 0]$$

```
> a &!!+ b;
```

$$- B[1, 2, 1] + \frac{u\, B[0, 1, 2]}{t} + (q + t)\, B[0, 0, 0]$$

SEE ALSO:

FUNCTION: `BnaCode2Bar` - convert codes into signed permutations

CALLING SEQUENCE:

```
BnaCode2Bar(e)
BNA[BnaCode2Bar](e)
```

PARAMETERS:

`e` = any element of any B_n-algebra

SYNOPSIS:

- The BnaCode2Bar function converts all codes that appear in the element e into signed permutations. The element e can belong to any B_n-algebra, but the result is no longer considered as an element of a B_n-algebra, the bases in this package being indexed by codes.
- Whenever there is a conflict between the function name BnaCode2Bar and another name used in the same session, use the long form BNA['BnaCode2Bar'].

EXAMPLES:

```
> with(BNA):
> BnaCode2Bar(q*B[0,1,0,2] - q^2*B[1,1,0,3]);
```

$$q\ B[2,\ 4,\ 1,\ 3] - q^2\ B[4,\ 2,\ -1,\ 3]$$

SEE ALSO:

FUNCTION: BnaCode2Palin - convert codes into palindromic permutations

CALLING SEQUENCE:

BnaCode2Palin(e)
BNA[BnaCode2Palin](e)

PARAMETERS:

e = any element of any B_n-algebra

SYNOPSIS:

- The BnaCode2Palin function converts all codes that appear in the element e into palindromic permutations. The element e can belong to any B_n-algebra but the result is no longer considered as an element of a B_n-algebra, the bases in this package being indexed by codes. The image belongs to the group algebra of $\mathfrak{S}_{2n}$.
- Whenever there is a conflict between the function name BnaCode2Palin and another name used in the same session, use the long form BNA['BnaCode2Palin'].

EXAMPLES:

```
> with(BNA):
> BnaCode2Palin(q*B[0,1,0,2] - q^2*B[1,1,0,3]);
```

$$q\ A[2,\ 4,\ 1,\ 3,\ 6,\ 8,\ 5,\ 7] - q^2\ A[2,\ 5,\ 3,\ 1,\ 8,\ 6,\ 4,\ 7]$$

See Also: BnaPalin2Code

Function: BnaMinus - difference of two elements of any B_n-algebra

Calling Sequence:

```
BnaMinus(e_1, e_2)
BNA[BnaMinus](e_1, e_2)
```

Parameters:

e_1, e_2 = any elements of any B_n-algebra

Synopsis:

- The BnaMinus function computes the difference between two elements of either the B_n-group algebra, B_n-nilCoxeter algebra or the B_n-idCoxeter algebra.
- One can use instead the operator &!!-.
- Whenever there is a conflict between the function name BnaMinus and another name used in the same session, use the long form BNA['BnaMinus'].

Examples:

```
> with(BNA):
> a:=q + u/t*B[0,1,2]:
> b:=t - B[1,2,1]:
> BnaMinus(a, b);
```

$$B[1, 2, 1] + \frac{u\ B[0, 1, 2]}{t} + (q - t)\ B[0, 0, 0]$$

```
> a &!!- a;
```

$$0$$

See Also:

Function: BnaNormal - normalize an element of a B_n-algebra

Calling Sequence:

```
BnaNormal(e)
BnaNormal(e, m)
BNA[BnaNormal](e)
BNA[BnaNormal](e, m)
```

Parameters:

e = any element of a B_n-algebra
m = any positive integer denoting the degree of a B_n-group

SYNOPSIS:

- The `BnaNormal` function reduces each code inside an expression to the minimum B_n-group containing it (B_n is contained in B_{n+1} by addition of one 0 to the right).
- If a parameter `m` has been specified, the minimum is choosen to be `m`.
- Whenever there is a conflict between the function name `BnaNormal` and another name used in the same session, use the long form `BNA['BnaNormal']`.

EXAMPLES:

```
> with(BNA):
> BnaNormal(a*B[0,1,2] - b^2*B[1,0,4,0,0]);

                      2
                - b  B[1, 0, 4] + a B[0, 1, 2]

> BnaNormal(a*B[0,1,2] - b^2*B[1,0,4,0,0], 4);

                    2
              - b  B[1, 0, 4, 0] + a B[0, 1, 2, 0]
```

SEE ALSO:

FUNCTION: `BnaPalin2Code` - convert palindromic permutations into codes

CALLING SEQUENCE:

```
BnaPalin2Code(e, n)
BNA[BnaPalin2Code](e, n)
```

PARAMETERS:

`e` = any element of any $\mathfrak{S}_n$-algebra
`n` = any positive integer

SYNOPSIS:

- The `BnaPalin2Code` function converts all palindromic permutations that appear in the element `e` into codes of B_n. The element `e` should belong to the group algebra of $\mathfrak{S}_{2n}$.
- Whenever there is a conflict between the function name `BnaPalin2Code` and another name used in the same session, use the long form `BNA['BnaPalin2Code']`.

EXAMPLES:

```
> with(BNA):
> BnaPalin2Code(q*A[2,4,1,3,6,8,5,7] - q^2*A[2,5,3,1,8,6,4,7], 4);

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

See Also: BnaCode2Palin

Function: Rd2BnGa - calculate a product of generators

Calling Sequence:

```
Rd2BnGa(rd)
Rd2BnGa(rd, n)
BNA[Rd2BnGa](rd)
BNA[Rd2BnGa](rd, n)
```

Parameters:

`rd` = any list denoting a decomposition
`n` = any positive integer

Synopsis:

- The `Rd2BnGa` function calculates the product of the generators given by the decomposition `rd` which needs not to be reduced (0 denotes the special generator). The product is taken in the B_n-group algebra.
- `Rd2BnGa(rd)=BN['BnRd2Code'](rd)`.
- When the second argument is not present, `n` is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2BnGa` and another name used in the same session, use the long form `BNA['Rd2BnGa']`.

Examples:

```
> with(BNA):
> Rd2BnGa([0,1,0,1], 5);
```

```
                              B[1, 3, 0, 0, 0]
```

```
> Rd2BnGa([0,1,0,1]);
```

```
                                  B[1, 3]
```

See Also: BN[BnRd2Code]

FUNCTION: Rd2BnIdca - calculate a product of generators

CALLING SEQUENCE:
```
Rd2BnIdca(rd)
Rd2BnIdca(rd, n)
BNA[Rd2BnIdca](rd)
BNA[Rd2BnIdca](rd, n)
```

PARAMETERS:
`rd` = any list denoting a decomposition
`n` = any positive integer

SYNOPSIS:

- The `Rd2BnIdca` function calculates the product of the generators given by the decomposition `rd` (0 denotes the special generator). The product is taken in the B_n-idCoxeter algebra.
- If `rd` is reduced then `Rd2BnIdca(rd)=BN['BnRd2Code'](rd)`, and otherwise there is a possible extra sign.
- When the second argument is not present, `n` is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2BnIdca` and another name used in the same session, use the long form `BNA['Rd2BnIdca']`.

EXAMPLES:

```
> with(BNA):
> Rd2BnIdca([0,1,0,1], 5);

                              B[1, 3, 0, 0, 0]

> Rd2BnIdca([0,1,0,1]);

                                   B[1, 3]

> Rd2BnIdca([1,0,1,0,1]);

                                 - B[1, 3]
> Rd2BnIdca([0,0]);

                                    B[1]
```

SEE ALSO: BN[BnRd2Code]

FUNCTION: Rd2BnNca - calculate a product of generators

CALLING SEQUENCE:
```
Rd2BnNca(rd)
Rd2BnNca(rd, n)
```

```
BNA[Rd2BnNca](rd)
BNA[Rd2BnNca](rd, n)
```

PARAMETERS:
`rd` = any list denoting a decomposition
`n` = any positive integer

SYNOPSIS:

- The `Rd2BnNca` function calculates the product of the generators given by the decomposition `rd` (0 denotes the special generator). The product is taken in the B_n-nilCoxeter algebra.
- If `rd` is reduced then `Rd2BnNca(rd)=BN['BnRd2Code'](rd)`, and 0 otherwise.
- When the second argument is not present, `n` is the smallest possible degree.
- Whenever there is a conflict between the function name `Rd2BnNca` and another name used in the same session, use the long form `BNA['Rd2BnNca']`.

EXAMPLES:

```
> with(BNA):
> Rd2BnNca([0,1,0,1], 5);
```

```
                              B[1, 3, 0, 0, 0]
```

```
> Rd2BnNca([0,1,0,1]);
```

```
                                   B[1, 3]
```

```
> Rd2BnNca([1,0,1,0,1]);
```

```
                                     0
```

SEE ALSO: `BN[BnRd2Code]`

FUNCTION: `Table2Bna` - convert a table into an element of any B_n-algebra

CALLING SEQUENCE:

```
Table2Bna(t)
BNA[Table2Bna](t)
```

PARAMETERS:
`t` = any table denoting an element of any B_n-algebra

SYNOPSIS:

- The `Table2Bna` function converts a table into an element of any B_n-algebra. The table should be indexed by codes of B_n and each entry corresponding to a code `cod` is interpreted as the coefficient of `cod` in the expression of the element `e`.
- Whenever there is a conflict between the function name `Table2Bna` and another name used in the same session, use the long form `BNA['Table2Bna']`.

EXAMPLES:

```
> with(BNA):
> t:=table([[0,2,1]=x1+x2, [0,2,2]=x1^2, [0,0,0]=1]);

                          t := table([
                                 [0, 2, 1] = x1 + x2
                                                2
                                 [0, 2, 2] = x1
                                 [0, 0, 0] = 1
                               ])

> Table2Bna(t);

                                                    2
            (x1 + x2) B[0, 2, 1] + B[0, 0, 0] + x1  B[0, 2, 2]
```

SEE ALSO: `Bna2Table`

5.12 SYMF to handle symmetric functions

HELP FOR: A package for 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', 's', 'p', 'm', 'c'`}.
- Whenever there is a conflict between a function name in `SYMF` and another name used in the same session, use the long form `SYMF`[<function>].
- The available functions are:

`Char2Sf`	`Sf2Char`	`Sf2Table`	`Sf2TableVar`
`SfAddBasis`	`SfCCProd`	`SfDiff`	`SfDualBasis`
`SfEval`	`SfInternal`	`SfJtMat`	`SfMat`
`SfOmega`	`SfPlethysm`	`SfScalar`	`SfTheta`
`SfZee`	`Table2Sf`	`Toc`	`Toe`
`Toh`	`Tom`	`Top`	`Tos`

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

EXAMPLES:

```
> with(SYMF):
> Tos(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]
```

FUNCTION: `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]);
```

$$\frac{1}{30}\,p_5\,p_3\,p_1^2$$

SEE ALSO: `SfInternal` `SfPlethysm` `Sf2Char` `SfZee`

FUNCTION: `Sf2Char` - convert a symmetric function into a virtual character

CALLING SEQUENCE:

```
Sf2Char(sf)
SYMF[Sf2Char](sf)
```

PARAMETERS:
`sf` = any symmetric function

SYNOPSIS:

- The `Sf2Char` function converts a symmetric function into a virtual character. This function is the inverse of `Char2Sf`.
- The `Sf2Char` function realizes the isomorphism between the linear space of homogeneous symmetric functions of degree `n` and class functions on the symmetric group of degree `n` (Frobenius isomorphism).
- The result is a linear combination of `cl[part]` where `part` is a partition.
- Whenever there is a conflict between the function name `Sf2Char` and another name used in the same session, use the long form `SYMF['Sf2Char']`.

EXAMPLES:

```
> with(SYMF):
> Sf2Char(s[3,1,1]);

              - 2 cl[2, 2, 1] + 6 cl[1, 1, 1, 1, 1] + cl[5]

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

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

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

SEE ALSO: `Char2Sf` `SfInternal` `SfPlethysm` `SfZee`

FUNCTION: `Sf2Table` - convert a symmetric function into a table

CALLING SEQUENCE:
`Sf2Table(sf, b)`
`SYMF[Sf2Table](sf, b)`

PARAMETERS:
`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Sf2Table` function converts a symmetric function into a table. The table is indexed by partitions (possibly empty) and each entry that corresponds to a partition `part` is interpreted as the coefficient of `b[part]` in the expression of the symmetric function in the basis `b`.
- Note that for instance if `sf=s[3,1,1]` and `b='s'` then the corresponding partition is `[3,1,1]` and if `sf=h3*h1^2` then the corresponding partition is `[3,1,1]` too.

- This function can be useful if one wants to act either on coefficients or on partitions. For instance, one can use the `map` function to act on coefficients.
- Whenever there is a conflict between the function name `Sf2Table` and another name used in the same session, use the long form `SYMF['Sf2Table']`.

EXAMPLES:

```
> with(SYMF):
> t := Sf2Table(q^2*s[3,1,1] + q*s[4,2] + q + s[4,2], 's');

                          t := table([
                                  [4, 2] = q + 1
                                               2
                                  [3, 1, 1] = q
                                  [] = q
                              ])

> t := map(coefficient -> coefficient - 1, t);

                        t := table([
                                [4, 2] = q
                                             2
                                [3, 1, 1] = q  - 1
                                [] = q - 1
                            ])

> Table2Sf(t, 's');

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

> Sf2Table(q*h1^3*h2 - q^2*h4*h3^2*h1, 'h');

                             table([
                                                  2
                                 [4, 3, 3, 1] = - q
                                 [2, 1, 1, 1] = q
                             ])
```

SEE ALSO: `Sf2TableVar` `Table2Sf`

FUNCTION: `Sf2TableVar` - variable set of a symmetric function

CALLING SEQUENCE:

```
Sf2TableVar(sf)
SYMF[Sf2TableVar](sf)
```

PARAMETERS:

`sf` = any symmetric function

SYNOPSIS:

- The `Sf2TableVar` function returns a table whose entries describe the set of all symmetric functions that appear in a given expression `sf`. The indices of the table are taken into the global variable `SYMFBases`.
- For all bases of type `'indexed'` (`SYMFBasesIndexed`), the entry consists of the set of all partitions that index the corresponding symmetric function in `sf`.
- For all bases of type `'string'` (`SYMFBasesString`), the entry consists of the set of all integers that index the corresponding symmetric function in `sf`.
- An entry equal to an empty set means that the symmetric function does not appear in `sf`.
- One can add a second argument to select some symmetric functions among those in `SYMFBases`. If the second argument is a set of names, then the restriction to this set of indices of the table is returned. In case the second argument is the name of a basis, then the corresponding entry in the table is returned instead of the table.
- Whenever there is a conflict between the function name `Sf2TableVar` and another name used in the same session, use the long form `SYMF['Sf2TableVar']`.

EXAMPLES:

```
> with(SYMF):
> Sf2TableVar(s[5]*s[4]*h3 + q*m[3]*p4*p6);

                              table([
                                  p = {4, 6}
                                  s = {[5], [4]}
                                  m = {[3]}
                                  c = {}
                                  e = {}
                                  h = {3}
                              ])

> Sf2TableVar(s[5]*s[4]*h3 + q*m[3]*p4*p6, {'p', 'm'});

                                table([
                                    p = {4, 6}
                                    m = {[3]}
                                ])

> Sf2TableVar(s[5]*s[4]*h3 + q*m[3]*p4*p6, 's');

                                  {[5], [4]}
```

SEE ALSO: `Sf2Table`

FUNCTION: SfAddBasis - addition of a new basis

CALLING SEQUENCE:

```
SfAddBasis(b)
SfAddBasis(b, scal)
SfAddBasis(b, scal, leadcoeff)
SYMF[SfAddBasis](b)
SYMF[SfAddBasis](b, scal)
SYMF[SfAddBasis](b, scal, leadcoeff)
```

PARAMETERS:

`b` = any name

`scal, leadcoeff` = any procedures accepting a partition as input

SYNOPSIS:

- The function `part` $\longrightarrow$ `scal(part)` choosen by the user defines a scalar product <,> on symmetric functions for which power sums are orthogonal, and such that <`p[part]`, `p[part]`> = `scal(part)`. The usual scalar product `SfScalar` corresponds to `scal(part)` = `SfZee(part)`.
- There exists a unique basis `b[]` of the linear space of symmetric functions that is orthogonal and satisfies: `b[part]` = `leadcoeff(part)* m[part]` + combination of `m[mu]`, `mu`<`part` (with respect to the total order on partitions listed with the `ListPart` function).
- By default, the leading coefficient is set equal to 1, i.e. is the function which returns 1 whatever is its argument: `proc(part) 1 end`.
- The new basis `b` is added to the list of bases and will be recognized in any expression; a new procedure `Tob` is available, with the same syntax as for instance `Tom`, `Tos`, and is devoted to express any symmetric function in the `b`-basis.
- It has also the effect to define a `Isb` function that tests whether an expression is an element of the `b`-basis.
- The result is the current set of known bases.
- An error is produced if `'b'` is the name of a previously defined basis.
- Whenever there is a conflict between the function name `SfAddBasis` and another name used in the same session, use the long form `SYMF['SfAddBasis']`.

EXAMPLES:

```
> with(SYMF):
> NewZee:=proc(part) SfZee(part, 'q', 't') end:
> SfAddBasis(Mac, NewZee);        # Macdonald polynomials, up to constants

                          {p, s, e, h, m, c, Mac}

> map(factor, Tom(Mac[2]*s[2,1], 'collect'));
```

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

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

         (9 t + 11 t q - 9 q - 11) m[2, 1, 1, 1]
       + ---------------------------------------
                       t q - 1

            (- 1 + t) (q + 1) m[1, 1, 1, 1, 1]
       + 20 ----------------------------------
                        t q - 1
```

```
> map(factor, ToMac(Mac[2]*s[1], 'collect'));
```

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

```
> map(factor, SfMat(3, 'Mac', 'm'));
```

```
                [                 2                    2 ]
                [       (q + 1) (q  + q + 1) (- 1 + t)  ]
                [ 1  0  ------------------------------- ]
                [                2                      ]
                [             (q  t - 1) (t q - 1)      ]
                [                                       ]
                [        (2 t q + t + q + 2) (- 1 + t)  ]
                [ 0  1   ---------------------------    ]
                [                     2                 ]
                [                  q t  - 1             ]
                [                                       ]
                [ 0  0               1                  ]
```

See Also: `SfDualBasis SfMat SfScalar SfZee`

Function: `SfCCProd` - compute the conjugacy class product of symmetric functions

Calling Sequence:

```
SfCCProd(sf_1, sf_2, b_1, b_2)
SYMF[SfCCProd](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 SfCCProd function computes the conjugacy class product of symmetric functions sf_1 and sf_2. This product is defined, on the Schur basis, to be $s_\lambda s_\mu = 0 (\lambda \neq \mu)$, $s_\lambda s_\lambda = 1/f(\lambda) s_\lambda$ in which $f(\lambda)$ stands for the number of standard tableaux of shape λ.
- The result is expressed in the basis of Schur functions.
- In the case that sf_1 and sf_2 are solely expressed in the basis b_1 and b_2, one may specify these bases.
- Whenever there is a conflict between the function name SfCCProd and another name used in the same session, use the long form SYMF['SfCCProd'].

Examples:

```
> with(SYMF):
> SfCCProd(s^2*s[2,1,1] + s[3,1], q*s[2,1,1]);
```

$$1/3\ s^2\ q\ s[2,\ 1,\ 1]$$

See Also:

Function: SfDiff - differential operator

Calling Sequence:

```
SfDiff(sf_1, sf_2)
SYMF[SfDiff](sf_1, sf_2)
```

Parameters:

sf_1, sf_2 = any symmetric functions

Synopsis:

- The SfDiff function computes $D_{\mathtt{sf_1}}(\mathtt{sf_2})$ in which $D_{\mathtt{sf_1}}$ stands for the differential operator corresponding to the symmetric function sf_1. Both symmetric functions sf_1 and sf_2 can be expressed in terms of any known basis since both expressions are converted in terms of Schur functions.
- The result is a linear combination of Schur functions, and not collected.
- Whenever there is a conflict between the function name SfDiff and another name used in the same session, use the long form SYMF['SfDiff'].

Examples:

```
> with(SYMF):
> SfDiff(s[3,1] + q*s[3], 2*e2*p2);

                              2 s[] + 2 q s[1]

> Tos(2*e2*p2);

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

> SfDiff(s[4,3,2,1,1], s[4,3,2,1,1]);

                                     s[]
```

See Also: `Tos`

Function: `SfDualBasis` - addition of a new basis, dual to an existing one

Calling Sequence:

```
SfDualBasis(b, old)
SfDualBasis(b, old, scal)
SYMF[SfDualBasis](b, old)
SYMF[SfDualBasis](b, old, scal)
```

Parameters:

`b` = any name
`old` = any name of a previously defined basis
`scal` = any procedure accepting a partition as input

Synopsis:

- The function `part` $\longrightarrow$ `scal(part)` choosen by the user defines a scalar product <,> on symmetric functions for which power sums are orthogonal, and such that <`p[part]`, `p[part]`> = `scal(part)`. The usual scalar product `SfScalar` corresponds to `scal(part)=SfZee(part)` and is choosen by default.

- Given any existing linear basis `old[part]` of the space of symmetric functions, there exists a unique dual basis `b[part]` relative to <,>, i.e. such that <`b[part]`, `old[part2]`> = 1 for `part=part2`, and 0 otherwise. The `old` basis can be of the type multiplicative (like `'h','e','p'`) or indexed (as `'s', 'm'`, or any other new defined basis). For example, the monomial symmetric functions `m[part]` are dual (with respect to the standard scalar product) to the `h`-basis (products of complete symmetric functions).

- The new basis `b` is added to the list of bases and will be recognized in any expression; a new procedure `Tob` is available, with the same syntax as for instance `Tom`, `Tos`, and is devoted to express any symmetric function in the `b`-basis.

- It has also the effect to define a `Isb` function that tests whether an expression is an element of the `b`-basis.

- The result is the current set of known bases.

- An error is produced if 'b' is the name of a previously defined basis, or old is unknown.
- Whenever there is a conflict between the function name SfDualBasis and another name used in the same session, use the long form SYMF['SfDualBasis'].

EXAMPLES:

```
> with(SYMF):
> NewZee:=proc(part) SfZee(part, 0, t) end:  # Hall-Littlewood polynomials
> SfDualBasis(MS, 's', NewZee);        # basis of modified Schur functions

                          {p, s, e, h, m, c, MS}

> Tos(MS[3], 'collect');

                                 2               3    2
          (1 - t) s[3] + (- t + t ) s[2, 1] + (- t  + t ) s[1, 1, 1]

> Tos(SfTheta(s[3], t, 0), 'collect');

                                 2               3    2
          (1 - t) s[3] + (- t + t ) s[2, 1] + (- t  + t ) s[1, 1, 1]

> ToMS(MS[2]*s[2], 'collect');

                MS[4]          (2 + t) MS[3, 1]     (t + 3) MS[2, 2]
         ------------------ + ------------------ + ------------------
                 2                    2                    2
         (- 1 + t)  (t + 1)   (- 1 + t)  (t + 1)   (- 1 + t)  (t + 1)

               (4 + 3 t) MS[2, 1, 1]     MS[1, 1, 1, 1]
             + --------------------- + 6 --------------
                          2                      2
                 (- 1 + t)  (t + 1)        (- 1 + t)

> SfMat(4, 'MS', 'm');

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

SEE ALSO: SfAddBasis SfMat SfScalar SfZee

FUNCTION: `SfEval` - realize the action of symmetric functions on polynomials

CALLING SEQUENCE:

```
SfEval(sf, expr)
SYMF[SfEval](sf, expr)
```

PARAMETERS:

`sf` = any symmetric function
`expr` = any expression depending on some variables

SYNOPSIS:

- Symmetric functions can be considered as operators on polynomials (λ-ring structure of the ring of polynomials). Let `sf` be a power-sum `pk`, then `SfEval(pk, c_u u + c_v v + ...)` is equal to `c_u u^k + c_v v^k + ...` where `u`, `v`, ... are monomials and `c_u`, `c_v`, ... are scalars.
- For a product of power-sums `sf=pi pj` ..., `SfEval(sf, expr)` is set to be equal to the product `SfEval(pi, expr) * SfEval(pj, expr) *` ... The definition is extended by linearity to any symmetric function `sf`.
- Whenever there is a conflict between the function name `SfEval` and another name used in the same session, use the long form `SYMF['SfEval']`.

EXAMPLES:

```
> with(SYMF):
> SfEval(p2, 3*x*y + 2*z);
```

$$3\,x^2\,y^2 + 2\,z^2$$

```
> SfEval(p3*p4 + q*p2, 2*x + 3*y);
```

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

SEE ALSO: `SfPlethysm`

FUNCTION: `SfInternal` - compute the inner tensor product of symmetric functions

CALLING SEQUENCE:

```
SfInternal(sf_1, sf_2, b_1, b_2)
SYMF[SfInternal](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 SfInternal 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, SfInternal computes the symmetric function associated with the product (Kronecker product) of these two virtual characters.
- For the basis of products of power-sums,

  SfInternal(...p2^j p1^i, ...p2^j' p1^i') = 0 or = SfZee([...,2^j,1^i]) ...p2^j p1^i whether (...,j,i) ≠ (...,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 SfInternal and another name used in the same session, use the long form SYMF['SfInternal'].

EXAMPLES:

```
> with(SYMF):
> SfInternal(p3*p3*p2*p1*p1, p3*p3*p2*p1*p1);

                                    2      2
                                72 p3  p2 p1

> SfZee([3,3,2,1,1]);

                                   72

> SfInternal(s[3,2,1], p3*h3, 's');

                                  2            3
                            - 2/3 p3  - 1/3 p3 p1
```

SEE ALSO: Char2Sf SfPlethysm Sf2Char SfZee

FUNCTION: SfJtMat - compute the Jacobi-Trudi matrix

CALLING SEQUENCE:

```
SfJtMat(part)
SYMF[SfJtMat](part)
```

PARAMETERS:

part = any list denoting a (skew) partition

SYNOPSIS:

- The `SfJtMat` function computes the Jacobi-Trudi matrix corresponding to the (skew) partition `part`. The determinant of this matrix is exactly the Schur function indexed by the partition, expressed in the `h`-basis.
- Whenever one uses a second argument `b`, the entries of the matrix are the `b`i's instead of the `h`i's.
- Whenever there is a conflict between the function name `SfJtMat` and another name used in the same session, use the long form `SYMF['SfJtMat']`.

EXAMPLES:

```
> with(SYMF):
> SfJtMat([3,2,1,1]);
```

$$\begin{bmatrix} h3 & h4 & h5 & h6 \\ h1 & h2 & h3 & h4 \\ 0 & 1 & h1 & h2 \\ 0 & 0 & 1 & h1 \end{bmatrix}$$

```
> SfJtMat([[2,1], [1]]);
```

$$\begin{bmatrix} h1 & h3 \\ 0 & h1 \end{bmatrix}$$

```
> SfJtMat([[2,2,1], [1,1,1]], 'b');
```

$$\begin{bmatrix} b1 & b2 & b3 \\ 1 & b1 & b2 \\ 0 & 0 & 1 \end{bmatrix}$$

SEE ALSO:

FUNCTION: `SfMat` - transition matrix between symmetric functions bases

CALLING SEQUENCE:

```
SfMat(n, b_1, b_2)
SYMF[SfMat](n, b_1, b_2)
```

PARAMETERS:

`n` = any positive integer
`b_1, b_2` = any names denoting known bases

SYNOPSIS:

- The SfMat function returns the transition matrix from the b_1 basis to the b_2 basis of the homogeneous component of weight n of the space of symmetric functions.
- Parameters b_1 and b_2 are taken from the SYMFBases global variable.
- The result is a matrix indexed by partitions of n, each row giving the expansion of an element of the b_1 basis in the b_2 basis.
- Whenever there is a conflict between the function name SfMat and another name used in the same session, use the long form SYMF['SfMat'].

EXAMPLES:

```
> with(SYMF):
> SfMat(4, 's', 'm');
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 2 & 3 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

SEE ALSO: TYP[Sf]

FUNCTION: SfOmega - apply the Ω-automorphism

CALLING SEQUENCE:

```
SfOmega(sf)
SYMF[SfOmega](sf)
```

PARAMETERS:

sf = any symmetric function

SYNOPSIS:

- The SfOmega function applies the Ω-automorphism to the symmetric function sf. This involution is defined as: h.i $\longrightarrow$ e.i, e.i $\longrightarrow$ h.i and p.i $\longrightarrow$ $(-1)^{(i-1)}$p.i. In terms of Schur functions, the involution conjugates the indexing partition.
- Whenever there is a conflict between the function name SfOmega and another name used in the same session, use the long form SYMF['SfOmega'].

EXAMPLES:

```
> with(SYMF):
> SfOmega(h3);
```

$$e_3$$

```
> SfOmega(p4);
```

$$-p_4$$

SEE ALSO: PART[Part2Conjugate] SfTheta

FUNCTION: SfPlethysm - plethysm of symmetric functions

CALLING SEQUENCE:

```
SfPlethysm(sf_1, sf_2)
SYMF[SfPlethysm](sf_1, sf_2)
```

PARAMETERS:

sf_1, sf_2 = any symmetric functions

SYNOPSIS:

- The SfPlethysm 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 and that one wants the result to be in the b_3 basis by using either SfPlethysm(sf_1, sf_2, b_1), or SfPlethysm(sf_1, sf_2, b_1, b_2) or SfPlethysm(sf_1, sf_2, b_1, b_2, b_3). The names b_1, b_2 and b_3 must be known bases.
- 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,
  1. (sf_1 + sf_2)[sf_3] = sf_1[sf_3] + sf_2[sf_3]
  2. (sf_1 * sf_2)[sf_3] = sf_1[sf_3] * sf_2[sf_3]
  3. sf_1[p.i] = p.i[sf_1]
  4. p.i[p.j] = p.(i*j)
- The default is to compute plethysms on the p-basis and return the result expressed on the p-basis.

- Special algorithms have been included to compute plethysms between elementary and complete symmetric functions in the basis of Schur functions. For that purpose, use all five arguments that is `SfPlethysm(sf_1, sf_2, b_1, b_2, 's')` where `b_1` and `b_2` are either `'e'` or `'h'`. These special plethysms use Newton's formula and Muir's formula. Note that these special algorithms are available when `sf_2` is a single elementary, complete or Schur symmetric function with some coefficient. Otherwise, the system uses the p-basis.
- Whenever there is a conflict between the function name `SfPlethysm` and another name used in the same session, use the long form `SYMF['SfPlethysm']`.

EXAMPLES:

```
> with(SYMF):
> SfPlethysm(p1*s[2], h1);
```

$$1/2\ p2\ p1 + 1/2\ p1^3$$

```
> SfPlethysm(e2*e1, h3, 'e', 'h', 's');
```

$$s[8, 1] + s[7, 2] + s[7, 1, 1] + 2\ s[6, 3] + s[6, 2, 1] + s[5, 4] + 2\ s[5, 3, 1] + s[4, 3, 2] + s[3, 3, 3]$$

SEE ALSO: `Char2Sf`

FUNCTION: `SfScalar` - scalar product of symmetric functions

CALLING SEQUENCE:

```
SfScalar(sf_1, sf_2)
SfScalar(sf_1, sf_2, sfp)
SYMF[SfScalar](sf_1, sf_2)
SYMF[SfScalar](sf_1, sf_2, sfp)
```

PARAMETERS:

`sf_1, sf_2` = any symmetric functions
`sfp` = any procedure accepting a partition as input

SYNOPSIS:

- The `SfScalar` 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=...p3^i p2^j p1^k`, then `SfScalar(sf, sf) = SfZee([...3^i, 2^j, 1^k])`.
- One can choose another scalar product, for which the p-basis will still remain orthogonal by replacing `SfZee` 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:

  1. `SfScalar(sf_1, sf_2, b_1, b_2)`

  2. `SfScalar(sf_1, sf_2, b_1, b_2, sfp)`

- When specifying the bases, it uses the property that the Schur basis is orthonormal, and the monomial basis is the dual basis of the basis of products of complete symmetric functions, for the usual scalar product described previously.

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

Examples:

```
> with(SYMF):
> SfScalar(p3^2*p1, p3^2*p1, 'p', 'p');

                                  18

> SfZee([3,3,1]);

                                  18

> SfScalar(s[6,3,3,3] + q*s[14,3,3], s[6,3,3,3] - q*s[14,3,1], 's', 's');

                                   1
```

See Also: `SfZee`

Function: `SfTheta` - apply the Θ-automorphism

Calling Sequence:

```
SfTheta(sf, a)
SfTheta(sf, q, t)
SYMF[SfTheta](sf, a)
SYMF[SfTheta](sf, q, t)
```

Parameters:

`sf` = any symmetric function
`a, q, t` = any names or expressions

Synopsis:

- The `SfTheta` function realizes a certain multiplicative automorphism of the ring of symmetric functions. It is defined on power-sum functions as follows: `SfTheta(sf, a)` gives the image of `sf` under the transformation `p.i` $\longrightarrow$ `a*p.i`.

- SfTheta(sf, q, t) is the image of sf under the transformation: p.i ⟶ (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 SfTheta and another name used in the same session, use the long form SYMF['SfTheta'].

EXAMPLES:

```
> with(SYMF):
> SfTheta(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

> SfTheta(p3, q, t);

                                      3
                                (1 - q ) p3
                              - -----------
                                         3
                                  - 1 + t

> SfTheta(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: SfOmega

FUNCTION: SfZee - compute the scalar product of a p-function with itself

CALLING SEQUENCE:

```
SfZee(part)
SfZee(part, q)
SfZee(part, q, t)
SYMF[SfZee](part)
SYMF[SfZee](part, q)
SYMF[SfZee](part, q, t)
```

PARAMETERS:

part = any list denoting a partition
q, t = any expressions

SYNOPSIS:

- Let `part=[...3^i 2^j 1^k]`, and let `sf` denote the product `...p3^i*p2^j*p1^k`, then we have `SfZee(part)` = `SfScalar(sf, sf)` = `... 3^i i! 2^j j! 1^k k!`.
- The argument must be a partition, that is a weakly decreasing list of positive integers.
- When used with a second parameter `q`, it corresponds to a usual deformation of the standard `SfScalar(sf, sf)` scalar product. Thus: `SfZee(part, q)` = `SfZee(part)*q^(nops(part))`.
- When used with three paramaters, it gives another deformation of the usual scalar product: `SfZee(part, q, t)` = `SfZee(part)` multiplied by all `(1-q^part[i])/(1-t^part[i])` for `i` from 1 to `nops(part)`.
- Whenever there is a conflict between the function name `SfZee` and another name used in the same session, use the long form `SYMF['SfZee']`.

EXAMPLES:

```
> with(SYMF):
> SfZee([4,3,1,1]);
```

$$24$$

```
> SfZee([4,3,1,1], 'q');
```

$$24\, q^4$$

```
> SfZee([4,3,1,1], 'q', 't');
```

$$24\,\frac{(1-q^4)\,(1-q^3)\,(1-q^2)}{(1-t^4)\,(1-t^3)\,(1-t^2)}$$

SEE ALSO:

FUNCTION: `Table2Sf` - convert a table into a symmetric function

CALLING SEQUENCE:

```
Table2Sf(t, b)
SYMF[Table2Sf](t, b)
```

PARAMETERS:

`t` = any table representing a symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Table2Sf` function converts a table into a symmetric function. The table should be indexed by partitions and each entry corresponding to a partition (possibly empty) `part` is interpreted as the coefficient of `b[part]` in the expression of the symmetric function in the basis `b`.
- Note that for instance if `b='s'` then `[3,1,1]` stands for `s[3,1,1]` and if `b='h'` then `[3,1,1]` denotes the element `h3*h1^2`.
- Whenever there is a conflict between the function name `Table2Sf` and another name used in the same session, use the long form `SYMF['Table2Sf']`.

EXAMPLES:

```
> with(SYMF):
> t:=table([[]=q, [3,1,1]=q^2, [4,2]=q+1]);

                          t := table([
                                                  2
                                  [3, 1, 1] = q
                                  [4, 2] = q + 1
                                  [] = q
                              ])

> Table2Sf(t, 's');

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

> Table2Sf(t, 'e');

                        2      2
                       q  e3 e1  + (q + 1) e4 e2 + q
```

SEE ALSO: `Sf2Table`

FUNCTION: `Toc` - express any symmetric function in the basis of cycle indices

CALLING SEQUENCE:

```
Toc(sf)
Toc(sf, b)
SYMF[Toc](sf)
SYMF[Toc](sf, b)
```

PARAMETERS:

`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Toc` function expresses any symmetric function in the basis of cycle indices, i.e. as a linear combination of cycle indices `c[part]` of conjugacy classes corresponding to the partition `part=[..., 3^i, 2^j, 1^k]`:

  ```
  c[part] = convert(part,`+`)!  / SfZee(part) * ...  p3^i p2^j p1^k  .
  ```

  In other words, the `c`-basis is proportional to the `p`-basis.

- The input is any expression in terms of the basic symmetric functions.
- The symmetric function `sf` is expanded and the result is not collected.
- One may 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 `Toc(sf, b, 'collect')`. For instance, `Toc(sf, 'c', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `Toc` and another name used in the same session, use the long form `SYMF['Toc']`.

EXAMPLES:

```
> with(SYMF):
> Toc((1+q)^3*e3*e2):                          # expands the input
> Toc((1+q)^3*e3*e2,noexpand):                 # does not expand (1+q)^3
> Toc((1+q)^3*e3*e2,collect):                  # collects the result
> Toc((1+q)^3*e3*e2,noexpand,'e'):             # the most efficient
> Toc((1+q)^3*e3*e2,'e',collect):              # specifies a basis
> Toc((1+q)^3*e3*e2,noexpand,'e',collect):
> Toc(c[2,1]*h4 - 2*q*p3);

     1/840 c[4, 2, 1] + 1/420 c[3, 2, 1, 1] + 1/280 c[2, 2, 2, 1]

     + 1/140 c[2, 2, 1, 1, 1] + 1/168 c[2, 1, 1, 1, 1, 1] - c[3] q
```

SEE ALSO: `Toe Toh Tom Top Tos`

FUNCTION: `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.
- 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)^5*h3*h4):                          # expands the input
> Toe((1+q)^5*h3*h4,noexpand):                 # does not expand (1+q)^5
> Toe((1+q)^5*h3*h4,collect):                  # collects the result
> Toe((1+q)^5*h3*h4,noexpand,'h'):             # the most efficient
> Toe((1+q)^5*h3*h4,'h',collect):              # specifies a basis
> Toe((1+q)^5*h3*h4,noexpand,'h',collect):
> Toe(s[2,1]*h4 - 2*q*p3);

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

              5        4                                    3
          + e1  e2 - e1  e3 - 6 q e3 + 6 q e2 e1 - 2 q e1
```

SEE ALSO: Toc Toh Tom Top Tos

FUNCTION: 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`.
- 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)^5*e3*e4):                          # expands the input
> Toh((1+q)^5*e3*e4,noexpand):                 # does not expand (1+q)^5
> Toh((1+q)^5*e3*e4,collect):                  # collects the result
> Toh((1+q)^5*e3*e4,noexpand,'e'):             # the most efficient
> Toh((1+q)^5*e3*e4,'e',collect):              # specifies a basis
> Toh((1+q)^5*e3*e4,noexpand,'e',collect):
> Toh(s[2,1]*e4 - 2*q*p3);

                            2          2        3        2          2   3
- h4 h2 h1 + h4 h3 + 5 h3 h1  h2 - 2 h3  h1 + h2  h1 - h2  h3 - 3 h2  h1

                5        4                                  3
            + h1  h2 - h1  h3 - 6 q h3 + 6 q h2 h1 - 2 q h1
```

SEE ALSO: `Toc Toe Tom Top Tos`

FUNCTION: `Tom` - express any symmetric function in the basis of basis of monomial symmetric functions

CALLING SEQUENCE:
`Tom(sf)`
`Tom(sf, b)`
`SYMF[Tom](sf)`

SYMF[Tom](sf, b)

PARAMETERS:
sf = any symmetric function
b = any name of a known basis

SYNOPSIS:

- The Tom function expresses any symmetric function in the basis of monomial symmetric functions.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may specify by a second argument, say b, that sf is solely expressed in terms of the known basis b.
- 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 Tom(sf, b, 'collect'). For instance, Tom(sf, 'm', 'collect') may be used to collect the argument sf.
- Whenever there is a conflict between the function name Tom and another name used in the same session, use the long form SYMF['Tom'].

EXAMPLES:

```
> with(SYMF):
> Tom((1+q)^5*e3*e4):                          # expands the input
> Tom((1+q)^5*e3*e4,noexpand):                 # does not expand (1+q)^5
> Tom((1+q)^5*e3*e4,collect):                  # collects the result
> Tom((1+q)^5*e3*e4,noexpand,'e'):             # the most efficient
> Tom((1+q)^5*e3*e4,'e',collect):              # specifies a basis
> Tom((1+q)^5*e3*e4,noexpand,'e',collect):
> Tom(s[2,1]*h4 - 2*q*p3);

    8 m[3, 2, 2] + m[6, 1] + 2 m[5, 2] + 4 m[5, 1, 1] + 2 m[4, 3]

  + 6 m[4, 2, 1] + 11 m[4, 1, 1, 1] + 6 m[3, 3, 1] + 14 m[3, 2, 1, 1]

  + 24 m[3, 1, 1, 1, 1] + 17 m[2, 2, 2, 1] + 28 m[2, 2, 1, 1, 1]

  + 45 m[2, 1, 1, 1, 1, 1] + 70 m[1, 1, 1, 1, 1, 1, 1] - 2 q m[3]
```

SEE ALSO: Toc Toe Toh Top Tos

FUNCTION: 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`.
- 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)^5*e3*e4):                          # expands the input
> Top((1+q)^5*e3*e4,noexpand):                 # does not expand (1+q)^5
> Top((1+q)^5*e3*e4,collect):                  # collects the result
> Top((1+q)^5*e3*e4,noexpand,'p'):             # the most efficient
> Top((1+q)^5*e3*e4,'p',collect):              # specifies a basis
> Top((1+q)^5*e3*e4,noexpand,'p',collect):
> Top(s[2,1]*h3 - 2*q*e3);

          3         2            4                          6           3
 1/18 p3 p1  - 1/9 p3  + 1/6 p2 p1  - 1/6 p2 p1 p3 + 1/18 p1  - 1/3 q p1

                       + q p2 p1 - 2/3 q p3
```

SEE ALSO: `Toc Toe Toh Tom Tos`

FUNCTION: `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`.
- 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)^5*e3*e4):                          # expands the input
> Tos((1+q)^5*e3*e4,noexpand):                 # does not expand (1+q)^5
> Tos((1+q)^5*e3*e4,collect):                  # collects the result
> Tos((1+q)^5*e3*e4,noexpand,'e'):             # the most efficient
> Tos((1+q)^5*e3*e4,'e',collect):              # specifies a basis
> Tos((1+q)^5*e3*e4,noexpand,'e',collect):
> Tos(s[2,1]*h4 - 2*q*p3);

            s[4, 2, 1] + s[5, 1, 1] + s[5, 2] + s[6, 1]

                - 2 (s[3] - s[2, 1] + s[1, 1, 1]) q
```

SEE ALSO: `Toc Toe Toh Tom Top`

5.13 SFA to handle symmetric functions on different alphabets

HELP FOR: A package for symmetric functions as operators on alphabets.

CALLING SEQUENCE:
`SFA[<function>](args)`
`<function>(args)`

SYNOPSIS:

- This package provides functions to work with symmetric functions, considered as operators acting on the ring of polynomial with real coefficients. It more precisely implements the structure of λ-ring.
- To get more information on symmetric functions on alphabets do either `?TYP,SfA` or `?TYP[SfA]`.
- All defined bases are stored in the global variable `SFABases` that is by default `['e', 'h', 's', 'p', 'm']`.
- Whenever there is a conflict between a function name in `SFA` and another name used in the same session, use the long form `SFA`[<function>].
- The available functions are:

Pol2SfA	Sf2SfA	SfA2Sf	SfA2TableVar
SfACollect	SfAExpand	SfAOmega	SfAVars
ToeA	TohA	TomA	TopA
TosA			

- For help with a particular function do either `?SFA`[<function>] or `?SFA`,<function> where <function> is one from the above list.

EXAMPLES:

```
> with(SFA):
> r := TosA( e[3,2](2*A1) );

   r := s[1, 1, 1, 1, 1](2 A1) + s[2, 2, 1](2 A1) + s[2, 1, 1, 1](2 A1)

> SfAExpand(r);

       20 s[2, 2, 1](A1) + 22 s[2, 1, 1, 1](A1) + 12 s[1, 1, 1, 1, 1](A1)

            + 2 s[4, 1](A1) + 8 s[3, 2](A1) + 12 s[3, 1, 1](A1)

> SfAExpand( (p[3]/3 + p[2,1]/2 + p[1,1,1]/6)(p[2](6*A1^2) + k) );
```

```
          3        2              2
1/6 k  + 1/2 k  + 1/3 k + 3 k  p[2, 2](A1) + 3 k p[2, 2](A1)

   + 3 k p[4, 4](A1) + 18 k p[2, 2, 2, 2](A1) + 2 p[6, 6](A1)

   + 18 p[4, 4, 2, 2](A1) + 36 p[2, 2, 2, 2, 2, 2](A1)
```

FUNCTION: Pol2SfA - from the basis of monomials to SFA

CALLING SEQUENCE:

```
Pol2SfA(pol, vlist)
SFA[Pol2SfA](pol, vlist)
```

PARAMETERS:

`pol` = any polynomial symmetric in `vlist`
`vlist` = a list of variables

SYNOPSIS:

- The `Pol2SfA` function transforms any polynomial symmetric in the list of variables `vlist` into a linear combination of monomial symmetric functions of `SFA`.
- Whenever there is a conflict between the function name `Pol2SfA` and another name used in the same session, use the long form `SFA['Pol2SfA']`.

EXAMPLES:

```
> with(SFA):
> SfAVars( {{x}, u, v} );

                                {v, u, {x}}

> r := SfAExpand( m[3,2](x1+u+v) - q*s[2,2](x1+u+v) );

       2  3     2  3     3  2     3  2    2  3    3  2      2  2      2
 r := x1  u  + x1  v  + x1  v  + x1  u  + u  v  + u  v  - q v  u  - q v  x1

          2   2          2        2        2  2
     - q v  x1  - q v x1 u  - q v x1  u - q x1  u

> Pol2SfA(r, [x1,u,v]);

   m[3, 2](x1 + u + v) - q m[2, 1, 1](x1 + u + v) - q m[2, 2](x1 + u + v)
```

SEE ALSO: CLG[x2m_n] SfAExpand SfAVars

FUNCTION: `Sf2SfA` - from `SYMF` objects to `SFA` objects

CALLING SEQUENCE:

```
Sf2SfA(sf)
Sf2SfA(sf, A)
SFA[Sf2SfA](sf)
SFA[Sf2SfA](sf, A)
```

PARAMETERS:

`sf` = any valid expression in `SYMF`
`A` = any alphabet

SYNOPSIS:

- The `Sf2SfA` function transforms any symmetric function of the `SYMF` package into an object of `SFA` for a specific alphabet (by default `A1`).
- Valid bases for symmetric functions are `'e'`, `'h'`, `'p'`, `'s'`, `'m'`. Added bases and the `c` basis are not considered.
- Allowed alphabet expressions are linear combinations of formal alphabets `A1`, `A2`, `A3`, ..., for instance `3/2*A1 - A3 + 3/4` is valid.
- The inverse function is `SfA2Sf`.
- Whenever there is a conflict between the function name `Sf2SfA` and another name used in the same session, use the long form `SFA['Sf2SfA']`.

EXAMPLES:

```
> with(SFA):
> Sf2SfA(s[3,1]*p2 - e2*e1);

                      s[3, 1](A1) p[2](A1) - e[2, 1](A1)

> Sf2SfA(s[3,1]*p2 - e2*e1, A2);

                      s[3, 1](A2) p[2](A2) - e[2, 1](A2)

> Sf2SfA(s[2,2], 2*A3);

                                s[2, 2](2 A3)
```

SEE ALSO: `SfA2Sf`

FUNCTION: `SfA2Sf` - from `SFA` objects to `SYMF` objects

CALLING SEQUENCE:

```
SfA2Sf(sfa)
SfA2Sf(sfa, alist)
```

```
SFA[SfA2Sf](sfa)
SFA[SfA2Sf](sfa, alist)
```

PARAMETERS:
`sf` = any valid expression in `SFA`
`alist` = a list of alphabets

SYNOPSIS:

- The `SfA2Sf` function transforms any symmetric function of the `SFA` package into an object of `SYMF` for a given list of specific alphabets (by default `A1`).
- Valid bases for symmetric functions are `'e'`, `'h'`, `'p'`, `'s'`, `'m'`. Added bases and the `c` basis are not considered.
- Allowed alphabet expressions are linear combinations of formal alphabets `A1`, `A2`, `A3`, ..., for instance `3/2*A1 - A3 + 3/4` is valid.
- The inverse function is `Sf2SfA`.
- Whenever there is a conflict between the function name `SfA2Sf` and another name used in the same session, use the long form `SFA['SfA2Sf']`.

EXAMPLES:

```
> with(SFA):
> sfa := 7/5*s[2,2](2*A3) + s[3,1](A2)*p[2](A2) - q*e[2,1](A2):
> SfA2Sf(sfa);

                      7/5 s[2, 2] + s[3, 1] p2 - e2 e1 q

> SfA2Sf(sfa, [ A2, A3 ]);

                    7/5 s[2, 2](2 A3) + s[3, 1] p2 - e2 e1 q

> SfA2Sf(sfa, [ A2, 2*A3 ]);

                      7/5 s[2, 2] + s[3, 1] p2 - e2 e1 q
```

SEE ALSO: `Sf2SfA`

FUNCTION: `SfA2TableVar` - symmetric functions appearing in an expression

CALLING SEQUENCE:

```
SfA2TableVar(sfa)
SfA2TableVar(sfa, base)
SFA[SfA2TableVar](sfa)
SFA[SfA2TableVar](sfa, base)
```

PARAMETERS:

`sfa` = any valid expression in `SFA`

`base` = a basis belonging to `SFABases`

SYNOPSIS:

- The `SfA2TableVar` function extracts symmetric functions appearing in `sfa`, and returns a table indexed by each basis `b`, each entry being itself a table indexed by each alphabet belonging to a `b`-function. The final entry is the set of indexing vectors of the `b`-function.
- When a base is specified as second argument, `SfA2TableVar` only returns the second table corresponding to the `b`-functions appearing in `sfa`.
- Whenever there is a conflict between the function name `SfA2TableVar` and another name used in the same session, use the long form `SFA['SfA2TableVar']`.

EXAMPLES:

```
> with(SFA):
> t := SfA2TableVar( q*s[3,2](A1)^2*m[3,1](p[2](A1))-s[2](A1)*h[3,2](k) );

               t := table([
                        s = table([
                                A1 = {[2], [3, 2]}
                            ])
                        p = table([
                                A1 = {[2]}
                            ])
                        e =
                        h = table([
                                k = {[3, 2]}
                            ])
                        m = table([
                                p[2](A1) = {[3, 2]}
                            ])
                    ])

> t[s][A1];

                                {[2], [3, 2]}

> SfA2TableVar( q*s[3,2](A1)^2*m[3,1](p[2](A1)) - s[2](A1)*h[3,2](k), s );

                            table([
                                A1 = {[2], [3, 2]}
                            ])
```

SEE ALSO: `SYMF[Sf2TableVar]`

FUNCTION: SfACollect - collect products that are on algebraic bases

CALLING SEQUENCE:

```
SfACollect(sfa)
SfACollect(sfa, base)
SfACollect(sfa, blist)
SFA[SfACollect](sfa)
SFA[SfACollect](sfa, base)
SFA[SfACollect](sfa, blist)
```

PARAMETERS:

`sfa` = any valid expression in `SFA`
`base` = an algebraic basis (`e, h, p`) or a linear basis
`blist` = a list of bases

SYNOPSIS:

- The `SfACollect` function transforms products of power sums, elementary and complete functions into a valid `SFA` expression. The `SfACollect` function also normalizes indexing vectors of each 'symmetric' function.
- Valid bases are `'e'`, `'h'`, `'p'`, `'s'`, `'m'`. They all are considered by default.
- One may specify to collect products (and / or normalize indexing vectors) expressed only on specified bases by adding a second argument which is either a single basis or a list of bases to be taken into account.
- Whenever there is a conflict between the function name `SfACollect` and another name used in the same session, use the long form `SFA['SfACollect']`.

EXAMPLES:

```
> with(SFA):
> SfACollect( p[3](A1)^4*e[2](A2)*e[1](A2) - q*h[2](A3)^6 );

          p[3, 3, 3, 3](A1) e[2, 1](A2) - h[2, 2, 2, 2, 2, 2](A3) q

> SfACollect( e[3](A1)*h[1,4](A1)^2 - s[2](A1)*s[1,3](A1), ['s', 'h'] );

 h[4, 4, 1, 1](A1) e[3](A1) + s[4, 2](A1) + s[3, 2, 1](A1) + s[2, 2, 2](A1)
```

SEE ALSO: `SfAExpand`

FUNCTION: SfAExpand - expand a SFA expression

CALLING SEQUENCE:

```
SfAExpand(sfa)
SFA[SfAExpand](sfa)
```

PARAMETERS:
`sfa` = any valid expression in `SFA`

SYNOPSIS:

- The `SfAExpand` function expands, whenever possible, any expression in `SFA`. It is a direct implementation of the λ-ring structure, i.e. symmetric functions are considered as operators over the ring of polynomials.
- Expansion is made considering the shape of the alphabet of each symmetric function appearing in `sfa`, and also products of symmetric functions. Expansion is inductively propagated, thus enabling chains of plethysms.
- Alphabets are either formal alphabets (`A1, A2, ...`), or real constants (any float object or any formal variable $\neq$`Ai`), or variables (when explicitely so declared), or symmetric functions of `SFA`, or any linear combination of alphabets. For instance, `3/2*A1 + k*(A2 - s[3](-A1))` is a valid alphabet.
- One may declare variables by setting e.g. `SfAVars({ {x}, {y}, z1, z2 })`. Here, `z1`, `z2`, all `x`i's, and all `y`i's will be held as variables in alphabets.
- Rules used for e.g. power sums symmetric functions are:
  - `p[i](-A) = -p[i](A)`
  - `p[i](A B) = p[i](A) p[i](B)`
  - `p[i](A+B) = p[i](A)+p[i](B)`
  - `p[i](k) = k`, `k` real constant ($\neq$ `Ai`)
  - `p[i](x) = x^i`, `x` explicitely declared variable
  - `p[i](Ai) = p[i](Ai)`, `Ai` formal alphabet
  - `p[i](p[j](A)) = p[ij](A)` (plethysm, from previous equations)

  where `i` and `j` are positive integers. Equivalent rules for the other bases are also implemented.
- Whenever there is a conflict between the function name `SfAExpand` and another name used in the same session, use the long form `SFA['SfAExpand']`.

EXAMPLES:

```
> with(SFA):
> SfAExpand( p[4,2](7+A1) );

                  49 + 7 p[2](A1) + 7 p[4](A1) + p[4, 2](A1)

> SfAExpand( s[2](A1 - 3/2*A2 + k) );

           2
      1/2 k  + 1/2 k - 3/2 k s[1](A2) + k s[1](A1) + 3/8 s[2](A2)

       + 15/8 s[1, 1](A2) - 3/2 s[1](A1) s[1](A2) + s[2](A1)

> SfAVars({ {x}, {y}, z1 }):
```

```
> SfAExpand( p[3]( q*m[2,1]( x1-k+A2 ) - z1^2+C ) );

    3                3         6         6                                2
q x1  p[6](A2) - q x1  k - q x1  k + q x1  p[3](A2) - q k p[6](A2) + q k

                                                          6
  + q k - q k p[3](A2) + q p[6, 3](A2) - q p[9](A2) - z1  + C
```

See Also: SfAVars SYMF[SfEval] SYMF[SfPlethysm] SfACollect

Function: SfAOmega - apply the Ω-automorphism

Calling Sequence:

```
SfAOmega(sfa)
SfAOmega(sfa, alist)
SFA[SfAOmega](sfa)
SFA[SfAOmega](sfa, alist)
```

Parameters:

`sfa` = any valid expression in `SFA`
`alist` = a list of alphabets

Synopsis:

- The Ω-automorphism is an involution defined on symmetric functions by:
  - `h[I](A)` ↔ `e[I](A)`
  - `s[I](A)` ↔ `s[I~](A)`
  - `p[I](A)` ↔ `-1^(length(I)+|I|) p[I](A)`,

  where `I~` is the conjugate partition of `I`, and `|I|` is the weight of `I`.
- One can apply `SfAOmega` solely on symmetric functions over the alphabets given in the second parameter `alist`.
- Whenever there is a conflict between the function name `SfAOmega` and another name used in the same session, use the long form `SFA['SfAOmega']`.

Examples:

```
> with(SFA):
> SfAOmega( p[3,1](A1) - q*s[3](A2) );

                        p[3, 1](A1) - q s[1, 1, 1](A2)

> SfAOmega( p[3,1](A1) - q*s[3](A2), [ A1 ]);

                           p[3, 1](A1) - q s[3](A2)
```

See Also: SYMF[SfOmega] PART[Part2Conjugate]

Function: SfAVars - specify valid variables in alphabets

Calling Sequence:

```
SfAVars(vset)
SFA[SfAVars](vset)
```

Parameters:

vset = a set of variables

Synopsis:

- The SfAVars function specifies which variables are considered for the SfAExpand function.
  - vset gives the set of valid variables for SFA, for instance:
  - { x1, x2, x3 },
  - { {x}, {y} }, all the xi's and yi's,
  - { {x}, {y}, t }, idem + the variable t.
- With no argument, SfAVars returns the current set of valid variables.
- Note that each call to SfAVars destroy the remember table of SfAExpand, if vset is specified.
- Whenever there is a conflict between the function name SfAVars and another name used in the same session, use the long form SFA['SfAVars'].

Examples:

```
> with(SFA):
> SfAVars({ {x}, t });
```

```
                                  {t, {x}}
```

```
> SfAExpand( s[3](x1+x2-t) );
```

```
                 2                 2     3        2     2        3
           - t x2  - t x1 x2 - t x1  + x2  + x1 x2  + x1  x2 + x1
```

```
> SfAVars({ {y}, t }):
> SfAExpand( s[3](x1+x2-t) );
```

```
            2                                  2                    3
   - 1/2 t x2  - 1/2 t x2 - t x1 x2 - 1/2 t x1  - 1/2 t x1 + 1/6 x2

              2                     2                 2            3
      + 1/2 x2  + 1/3 x2 + 1/2 x1 x2  + x1 x2 + 1/2 x1  x2 + 1/6 x1

              2
      + 1/2 x1  + 1/3 x1
```

```
> SfAVars();
```

```
{t, {y}}
```

See Also: SfAExpand

Function: ToeA - develop any symmetric function on the e-basis

Calling Sequence:

```
ToeA(sfa)
ToeA(sfa, alist)
SFA[ToeA](sfa)
SFA[ToeA](sfa, alist)
```

Parameters:

`sfa` = any valid expression in SFA
`alist` = a list of alphabets

Synopsis:

- The ToeA function analyses sfa and develops any symmetric function in SFA on the linear basis of product of elementary symmetric functions.
- One can develop only symmetric functions of which alphabets are given in the optional argument.
- Whenever there is a conflict between the function name ToeA and another name used in the same session, use the long form SFA['ToeA'].

Examples:

```
> with(SFA):
> ToeA( s[3,1](2*A1) );

        e[2, 1, 1](2 A1) - e[2, 2](2 A1) - e[3, 1](2 A1) + e[4](2 A1)

> ToeA( s[3,1](2*A1)*m[2,1](A2^2), [ A2^2, A3 ] );

                                    2                          2
            s[3, 1](2 A1) e[2, 1](A2 ) - 3 s[3, 1](2 A1) e[3](A2 )

> ToeA( s[2]( p[4,3](2*A1) ) + q*m[2,1](A2^2),  [ p[4,3](2*A1) ] );

                                                                  2
        e[1, 1](p[4, 3](2 A1)) - e[2](p[4, 3](2 A1)) + q m[2, 1](A2 )
```

See Also: TohA TopA TosA TomA SYMF[Toe] SfAExpand

Function: TohA - develop any symmetric function on the h-basis

Calling Sequence:

```
TohA(sfa)
TohA(sfa, alist)
SFA[TohA](sfa)
SFA[TohA](sfa, alist)
```

Parameters:

sfa = any valid expression in SFA
alist = a list of alphabets

Synopsis:

- The TohA function analyses sfa and develops any symmetric function in SFA on the linear basis of product of complete symmetric functions.
- One can develop only symmetric functions of which alphabets are given in the optional argument.
- Whenever there is a conflict between the function name TohA and another name used in the same session, use the long form SFA['TohA'].

Examples:

```
> with(SFA):
> TohA( s[3,1](2*A1) );
```

$$h[3,\ 1](2\ A1) - h[4](2\ A1)$$

```
> TohA( s[3,1](2*A1)*m[1,1](A2^2), [ A2^2, A3 ] );
```

$$- s[3,\ 1](2\ A1)\ h[2](A2^2) + s[3,\ 1](2\ A1)\ h[1,\ 1](A2^2)$$

```
> TohA( s[2]( p[4,3](2*A1) ) + q*m[2,1](A2^2),  [ p[4,3](2*A1) ] );
```

$$h[2](p[4,\ 3](2\ A1)) + q\ m[2,\ 1](A2^2)$$

See Also: ToeA TopA TosA TomA SYMF[Toh] SfAExpand

Function: TomA - develop any symmetric function on the m-basis

Calling Sequence:

```
TomA(sfa)
TomA(sfa, alist)
SFA[TomA](sfa)
```

```
SFA[TomA](sfa, alist)
```

PARAMETERS:
`sfa` = any valid expression in `SFA`
`alist` = a list of alphabets

SYNOPSIS:

- The `TomA` function analyses `sfa` and develops any symmetric function in `SFA` on the linear basis of monomial symmetric functions.
- One can develop only symmetric functions of which alphabets are given in the optional argument.
- Whenever there is a conflict between the function name `TomA` and another name used in the same session, use the long form `SFA['TomA']`.

EXAMPLES:

```
> with(SFA):
> TomA( s[3,1](2*A1) );

      m[3, 1](2 A1) + m[2, 2](2 A1) + 2 m[2, 1, 1](2 A1)

        + 3 m[1, 1, 1, 1](2 A1)

> TomA( s[3,1](2*A1)*m[2](A2^2)^2, [ A2^2, A3 ] );

                                      2                          2
           2 s[3, 1](2 A1) m[2, 2](A2 ) + s[3, 1](2 A1) m[4](A2 )

> TomA( s[2]( p[4,3](2*A1) ) + q*m[2,1](A2^2),  [ p[4,3](2*A1) ] );

                                                                 2
        m[2](p[4, 3](2 A1)) + m[1, 1](p[4, 3](2 A1)) + q m[2, 1](A2 )
```

SEE ALSO: `ToeA` `TohA` `TopA` `TosA` `SYMF[Tom]` `SfAExpand`

FUNCTION: `TopA` - develop any symmetric function on the p-basis

CALLING SEQUENCE:
```
TopA(sfa)
TopA(sfa, alist)
SFA[TopA](sfa)
SFA[TopA](sfa, alist)
```

PARAMETERS:
`sfa` = any valid expression in `SFA`
`alist` = a list of alphabets

SYNOPSIS:

- The `TopA` function analyses `sfa` and develops any symmetric function in `SFA` on the linear basis of product of power sums functions.
- One can develop only symmetric functions of which alphabets are given in the optional argument.
- Whenever there is a conflict between the function name `TopA` and another name used in the same session, use the long form `SFA['TopA']`.

EXAMPLES:

```
> with(SFA):
> TopA( s[3](2*A1) );

          1/3 p[3](2 A1) + 1/2 p[2, 1](2 A1) + 1/6 p[1, 1, 1](2 A1)

> TopA( s[3,1](2*A1)*m[2,1](A2^2), [ A2^2, A3 ] );

                                    2                          2
             s[3, 1](2 A1) p[2, 1](A2 ) - s[3, 1](2 A1) p[3](A2 )

> TopA( s[2]( p[4,3](2*A1) ) + q*m[2,1](A2^2),  [ p[4,3](2*A1) ] );

                                                                    2
    1/2 p[2](p[4, 3](2 A1)) + 1/2 p[1, 1](p[4, 3](2 A1)) + q m[2, 1](A2 )
```

SEE ALSO: `ToeA TohA TosA TomA SYMF[Top] SfAExpand`

FUNCTION: `TosA` - develop any symmetric function on the s-basis

CALLING SEQUENCE:

```
TosA(sfa)
TosA(sfa, alist)
SFA[TosA](sfa)
SFA[TosA](sfa, alist)
```

PARAMETERS:

`sfa` = any valid expression in `SFA`
`alist` = a list of alphabets

SYNOPSIS:

- The `TosA` function analyses `sfa` and develops any symmetric function in `SFA` on the linear basis of Schur functions.
- One can develop only symmetric functions of which alphabets are given in the optional argument.
- Whenever there is a conflict between the function name `TosA` and another name used in the same session, use the long form `SFA['TosA']`.

EXAMPLES:

```
> with(SFA):
> TosA( h[4,3](2*A1) );

          s[4, 3](2 A1) + s[5, 2](2 A1) + s[6, 1](2 A1) + s[7](2 A1)

> TosA( s[3,1](2*A1)*m[2,1](A2^2), [ A2^2, A3 ] );

                                  2                                  2
         s[3, 1](2 A1) s[2, 1](A2 ) - 2 s[3, 1](2 A1) s[1, 1, 1](A2 )

> TosA( s[4]( p[4,3](2*A1) )^2 + q*p[2](A2),  [ p[4,3](2*A1) ] );

     s[8](p[4, 3](2 A1)) + s[7, 1](p[4, 3](2 A1)) + s[6, 2](p[4, 3](2 A1))

          + s[5, 3](p[4, 3](2 A1)) + s[4, 4](p[4, 3](2 A1)) + q p[2](A2)
```

SEE ALSO: `ToeA` `TohA` `TopA` `TomA` `SYMF[Tos]` `SfAExpand`

5.14 FREE, a package to work in the free algebra

HELP FOR: A package to handle words.

CALLING SEQUENCE:
`FREE[<function>](args)`
`<function>(args)`

SYNOPSIS:

- This package provides functions to work with the free algebra, that is the algebra of words. It is also possible to work in the plactic monoid, i.e. the free algebra of words modulo the plactic relations.
- A general element is expressed as a linear combination of `w[word]`, where `word` is a sequence of integers, with arbitrary coefficients.
- Whenever there is a conflict between a function name in `FREE` and another name used in the same session, use the long form `FREE`[<function>].
- The available functions are:

`Free2NilFree`	`Free2PlaxClass`	`Free2Plax`	`Free2Pol`
`Free2StdFree`	`Free2Table`	`FreeAdd`	`FreeConcat`
`FreeMinus`	`FreeNormal`	`FreeShuffle`	`IdcaOnFree`
`NilFree2Free`	`Pol2Free`	`Pol2Row`	`Sf2Free`
`SgaOnFree`	`Table2Free`		

- For help with a particular function do either ?`FREE`[<function>] or ?`FREE`,<function> where <function> is one from the above list.
- Instead of `FreeAdd`, `FreeMinus`, `FreeConcat`, one can use in short

  `&~+` `&~-` `&~*`

EXAMPLES:

```
> with(FREE):
> FreeShuffle(w[2,1] + w[2,1], w[1,1], 'q');

                  4                        2      3
              2 q  w[1, 1, 2, 1] + (2 q  + 2 q ) w[1, 2, 1, 1]

                                2
                 + (2 + 2 q + 2 q ) w[2, 1, 1, 1]
```

FUNCTION: Free2NilFree - inverse plaxification map

CALLING SEQUENCE:
```
Free2NilFree(free)
FREE[Free2NilFree](free)
```

PARAMETERS:
`free` = any element of the free algebra

SYNOPSIS:

- The `Free2NilFree` is the inverse plaxification map which transforms a word into a reduced decomposition. This map is defined recursively by:

  ```
  invplaxify(w[i,j,...])  = w[i, invplaxify(SgaOnFree(s_i, w[j,...]))]
  ```

  where `s_i` is the simple transposition `(i,i+1)`.
- The inverse plaxification map has the property to send a plactic class onto a nil plactic class.
- Whenever there is a conflict between the function name `Free2NilFree` and another name used in the same session, use the long form `FREE['Free2NilFree']`.

EXAMPLES:

```
> with(FREE):
> Free2NilFree(w[3,2,2,1,1,1]);

                          w[3, 2, 3, 1, 2, 3]

> Free2NilFree(w[1,2,1] + w[2,1,1]);

                        w[1, 2, 1] + w[2, 1, 2]
```

SEE ALSO: NilFree2Free

FUNCTION: Free2Plax - quotient of the free algebra by plactic relations

CALLING SEQUENCE:
```
Free2Plax(free)
FREE[Free2Plax](free)
```

PARAMETERS:
`free` = any element of the free algebra

SYNOPSIS:

- The `Free2Plax` function projects the free algebra onto the plactic algebra. It sends any word to the unique tableau in its class (that is to say, to the word obtained by reading the successive rows of the tableau given by Schensted algorithm).
- Whenever there is a conflict between the function name `Free2Plax` and another name used in the same session, use the long form `FREE['Free2Plax']`.

EXAMPLES:

```
> with(FREE):
> Free2Plax(w[1,2,1] + w[2,1,1] + z*w[1,3,2]);

                        2 w[2, 1, 1] + z w[3, 1, 2]
```

SEE ALSO:

FUNCTION: `Free2PlaxClass` - replace each word by its plactic class

CALLING SEQUENCE:

```
Free2PlaxClass(free)
FREE[Free2PlaxClass](free)
```

PARAMETERS:

`free` = any element of the free algebra

SYNOPSIS:

- The `Free2PlaxClass` function replaces each word by the sum of all words which are in the same plactic class (*i.e.* which give the same tableau by Schensted algorithm).
- Whenever there is a conflict between the function name `Free2PlaxClass` and another name used in the same session, use the long form `FREE['Free2PlaxClass']`.

EXAMPLES:

```
> with(FREE):
> Free2PlaxClass(w[2,3,1,3]);

              w[2, 1, 3, 3] + w[2, 3, 1, 3] + w[2, 3, 3, 1]

> Free2PlaxClass(q*w[2,3,1] + w[2,1,3]);

                  (1 + q) w[2, 3, 1] + (1 + q) w[2, 1, 3]
```

SEE ALSO:

FUNCTION: `Free2Pol` - from words to polynomials

CALLING SEQUENCE:

```
Free2Pol(free)
FREE[Free2Pol](free)
```

PARAMETERS:
`free` = any element of the free algebra

SYNOPSIS:

- The `Free2Pol` function projects the free algebra onto the algebra of polynomials. It sends each word `w[n1, n2, ..., nk]` to the monomial `x.n1 * x.n2 * ... * x.nk`.
- Whenever there is a conflict between the function name `Free2Pol` and another name used in the same session, use the long form `FREE['Free2Pol']`.

EXAMPLES:

```
> with(FREE):
> Free2Pol(w[1,2,1] + w[2,1,1] + z*w[1,3,2]);
```

$$2\,x1\;x2^{2} + z\;x1\;x3\;x2$$

SEE ALSO: `Pol2Free` `Pol2Row`

FUNCTION: `Free2StdFree` - standardization of words

CALLING SEQUENCE:
`Free2StdFree(free)`
`FREE[Free2StdFree](free)`

PARAMETERS:
`free` = any element of the free algebra

SYNOPSIS:

- The `Free2StdFree` function standardizes all words that appear in a given element of the free algebra. The standardization process replaces the 1's by `1..h`, the 2's by `h+1..k`, etc, from left to right, in each word of the `free` element. It is compatible with Schensted algorithm.
- Whenever there is a conflict between the function name `Free2StdFree` and another name used in the same session, use the long form `FREE['Free2StdFree']`.

EXAMPLES:

```
> with(FREE):
> Free2StdFree(w[3,4,3,1,2,1,1] + w[1,1,1]);
```

$$w[5,\,7,\,6,\,1,\,4,\,2,\,3] + w[1,\,2,\,3]$$

See Also:

Function: `Free2Table` - convert an element of the free algebra into a table

Calling Sequence:

```
Free2Table(free)
FREE[Free2Table](free)
```

Parameters:

`free` = any element of the free algebra

Synopsis:

- The `Free2Table` function converts an element of the free algebra into a table. The table is indexed by lists and each entry corresponding to a list `li` is the coefficient of the word `w[li]` in the `free` element.
- This function can be useful if one wants to act either on coefficients or on words. For instance, one can use the `map` function to act on coefficients.
- Whenever there is a conflict between the function name `Free2Table` and another name used in the same session, use the long form `FREE['Free2Table']`.

Examples:

```
> with(FREE):
> Free2Table(q^2*w[1,2,1] + w[2,1,1] + q*w[1,2,1]);
```

```
                          table(sparse,[
                                        2
                            [1, 2, 1] = q  + q
                            [2, 1, 1] = 1
                          ])
```

See Also: `Table2Free`

Function: `FreeAdd` - sum of two elements of the free algebra

Calling Sequence:

```
FreeAdd(e_1, e_2)
FREE[FreeAdd](e_1, e_2)
```

Parameters:

`e_1, e_2` = any elements of the free algebra

Synopsis:

- The `FreeAdd` function adds two elements of the free algebra. It corresponds to the usual operation '`+`' but the function normalizes the result, by essentially collecting coefficients.

- One can use instead the operator &~+.
- Whenever there is a conflict between the function name FreeAdd and another name used in the same session, use the long form FREE['FreeAdd'].

EXAMPLES:

```
> with(FREE):
> a:=q + u/t*w[2,1,1]:
> b:=t - w[1,2,2]:
> FreeAdd(a, b);
```

$$(q + t)\ w[] - w[1, 2, 2] + \frac{u\ w[2, 1, 1]}{t}$$

```
> a &~+ b;
```

$$(q + t)\ w[] - w[1, 2, 2] + \frac{u\ w[2, 1, 1]}{t}$$

SEE ALSO:

FUNCTION: FreeConcat - concatenation product in the free algebra

CALLING SEQUENCE:

```
FreeConcat(e_1, e_2)
FREE[FreeConcat](e_1, e_2)
```

PARAMETERS:

e_1, e_2 = any elements of the free algebra

SYNOPSIS:

- The FreeConcat function computes the concatenation product of two elements of the free algebra. The function normalizes the result, by essentially collecting coefficients.
- One can use instead the operator &~*.
- Whenever there is a conflict between the function name FreeConcat and another name used in the same session, use the long form FREE['FreeConcat'].

EXAMPLES:

```
> with(FREE):
> a:=q + u/t*w[2,1,1]:
> b:=t - w[1,2,2]:
> FreeConcat(a, b);
```

```
                                                  u w[2, 1, 1, 1, 2, 2]
      q t w[] + u w[2, 1, 1] - q w[1, 2, 2] - ----------------------
                                                          t
```

```
> a &~* b;
```

```
                                                  u w[2, 1, 1, 1, 2, 2]
      q t w[] + u w[2, 1, 1] - q w[1, 2, 2] - ----------------------
                                                          t
```

See Also: `FreeShuffle`

Function: `FreeMinus` - difference of two elements of the free algebra

Calling Sequence:

```
FreeMinus(e_1, e_2)
FREE[FreeMinus](e_1, e_2)
```

Parameters:

`e_1, e_2` = any elements of the free algebra

Synopsis:

- The `FreeMinus` function computes the difference of two elements of the free algebra. It corresponds to the usual operation '`-`' but the function normalizes the result, by essentially collecting coefficients.
- One can use instead the operator `&~-`.
- Whenever there is a conflict between the function name `FreeMinus` and another name used in the same session, use the long form `FREE['FreeMinus']`.

Examples:

```
> with(FREE):
> a:=q + u/t*w[2,1,1]:
> b:=t - w[1,2,2]:
> FreeMinus(a, b);
```

```
                  u w[2, 1, 1]
                  ------------ + (q - t) w[] + w[1, 2, 2]
                        t
```

```
> a &~- a;
```

```
                                  0
```

See Also:

Function: FreeNormal - normalize an element of the free algebra

Calling Sequence:
FreeNormal(free)
FREE[FreeNormal](free)

Parameters:
free = any element of the free algebra

Synopsis:

- The FreeNormal function normalizes an element by essentially collecting coefficients. In particular, the empty word w[] is added when necessary.
- Whenever there is a conflict between the function name FreeNormal and another name used in the same session, use the long form FREE['FreeNormal'].

Examples:

```
> with(FREE):
> FreeNormal(w[2,1] + q + 1 + q*w[1,2] + w[2,1]);

                    (1 + q) w[] + q w[1, 2] + 2 w[2, 1]
```

See Also:

Function: FreeShuffle - shuffle product in the free algebra

Calling Sequence:
FreeShuffle(e_1, e_2)
FREE[FreeShuffle](e_1, e_2)

Parameters:
e_1, e_2 = any elements of the free algebra

Synopsis:

- The FreeShuffle function computes the shuffle product of two elements of the free algebra. The function normalizes the result, by essentially collecting coefficients.
- The shuffle of two words w[a, u] and w[b, v] is recursively defined by: FreeShuffle(w[a,u], w[b,v]) = FreeConcat(w[a], FreeShuffle(w[u],w[b,v])) + FreeConcat(w[b],FreeShuffle(w[a,u], w[v]) and FreeShuffle(w[], w[c]) = FreeShuffle(w[c], w[]) = c.
- One can add the extra argument 'q' to compute the q-shuffle product. It defined on words by: FreeShuffle(w[a,u], w[b,v], 'q') = FreeConcat(w[a], FreeShuffle(w[u],w[b,v], 'q') + q^(nops([a,u])) * FreeConcat(w[b], FreeShuffle(w[a,u], w[v], 'q').
- Whenever there is a conflict between the function name FreeShuffle and another name used in the same session, use the long form FREE['FreeShuffle'].

EXAMPLES:

```
> with(FREE):
> FreeShuffle(w[1,2] + w[2,1], w[1,2,1]);

       4 w[1, 2, 1, 2, 1] + 4 w[1, 1, 2, 2, 1] + 2 w[1, 1, 2, 1, 2]

     + 2 w[1, 2, 1, 1, 2] + 2 w[2, 1, 1, 2, 1] + 2 w[2, 1, 2, 1, 1]

     + 4 w[1, 2, 2, 1, 1]

> FreeShuffle(w[1,2], w[2,1], 'q');

                2                    3    4
               q  w[2, 1, 2, 1] + (q  + q ) w[2, 1, 1, 2]

                                        2
             + (1 + q) w[1, 2, 2, 1] + q  w[1, 2, 1, 2]
```

SEE ALSO: `FreeConcat`

FUNCTION: `IdcaOnFree` - action of the idCoxeter algebra on words

CALLING SEQUENCE:

`IdcaOnFree(e_1, free)`
`FREE[IdcaOnFree](e_1, free)`

PARAMETERS:

`e_1` = any element of the idCoxeter algebra
`free` = any element of the free algebra

SYNOPSIS:

- The IdcaOnFree function realizes the action of the idCoxeter algebra on the free algebra.
- The elementary action of a simple transposition, say (1,2), on a word `v` is as follows: forget all letters different from 1 and 2, then cross out all successive factors [2,1] of what is left, in any order. The remaining word is of the type `v'=w[1$a,2$b]`. If `a¿b` then the image of `v` is the sum of all words obtained from `v` by replacing its subword `v'` by `w[1$(a-1), 2$(b+1)], ..., w[1$b, 2$a]` successively. If `a=b`, it is sent to 0. If `a<b` it is sent to minus the image of the word where `v'` is exchanged into `w[1$b, 2$a]`.
- For example, `IdcaOnFree(A[2,1], w[3,1,1,1,2,1,2]) = w[3,1,1,2,2,1,2] + w[3,1,2,2, 2,1,2]` because `v' = [*,1,1,1,*,*,2]`.
- Whenever there is a conflict between the function name `IdcaOnFree` and another name used in the same session, use the long form `FREE['IdcaOnFree']`.

Examples:

```
> with(FREE):
> IdcaOnFree(A[2,1], w[3,1,1,1,2,1,2]);

                w[3, 1, 2, 2, 2, 1, 2] + w[3, 1, 1, 2, 2, 1, 2]
```

See Also: SgaOnFree

Function: NilFree2Free - plaxification map

Calling Sequence:

```
NilFree2Free(free)
FREE[NilFree2Free](free)
```

Parameters:

`free` = any element of the free algebra

Synopsis:

- The `NilFree2Free` function transforms each word seen as a reduced decomposition into a word, according to the plaxification map. This map is defined recursively by:

  ```
  plaxify(w[i,j,...])  = w[i, SgaOnFree(s_i, plaxify(w[j,...]))]
  ```

  where `s_i` is the simple transposition `(i,i+1)`.

- The plaxification map has the property to send a nilplactic class (of reduced decompositions) onto a plactic class; it moreover preserves the Q-symbol of Schensted.

- Whenever there is a conflict between the function name `NilFree2Free` and another name used in the same session, use the long form `FREE['NilFree2Free']`.

Examples:

```
> with(FREE):
> NilFree2Free(w[2,1,2,1]);

                              w[2, 1, 3, 1]

> NilFree2Free(w[2,3,1] + w[2,1,3]);

                          w[2, 1, 2] + w[2, 2, 1]

> r := map(z -> w[op(z)], Perm2ListRd([2,4,3,1]));

            r := [w[1, 2, 3, 2], w[3, 1, 2, 3], w[1, 3, 2, 3]]

> r := map(NilFree2Free, r);
```

```
            r := [w[1, 1, 3, 1], w[3, 1, 1, 1], w[1, 3, 1, 1]]

> map(Word2Tab, r);

             [[[3], [1, 1, 1]], [[3], [1, 1, 1]], [[3], [1, 1, 1]]]
```

See Also: `Free2NilFree` `Schensted` `SgaOnFree` `Free2PlaxClass`

Function: `Pol2Free` - from polynomials to words

Calling Sequence:

```
Pol2Free(pol)
FREE[Pol2Free](pol)
```

Parameters:

`pol` = any polynomial

Synopsis:

- The `Pol2Free` function converts a polynomial into an element of the free algebra. The polynomial is first expressed in the basis of ordered products of elementary symmetric functions in different alphabets $\{x_1, \ldots, x_k\}$. Then the elementary symmetric functions are replaced by sums of all strictly decreasing words and their product is taken in the free algebra.
- Whenever there is a conflict between the function name `Pol2Free` and another name used in the same session, use the long form `FREE['Pol2Free']`.

Examples:

```
> with(FREE):
> Pol2Free(x1*x2+x1*x3+x2*x3 + aa*x2^2);

         aa w[1, 2] + aa w[2, 2] + (1 - aa) w[2, 1] + w[3, 1] + w[3, 2]
```

See Also: `Free2Pol` `Pol2Row` `Sf2Free`

Function: `Pol2Row` - from polynomials to increasing words

Calling Sequence:

```
Pol2Row(pol)
FREE[Pol2Row](pol)
```

Parameters:

`pol` = any polynomial

SYNOPSIS:

- The Pol2Row function converts a polynomial into an element of the free algebra. Each monomial x.l1 * x.l2 * ... * x.lk is replaced by the increasing word w[l1, l2, ..., lk] (l1 $\leq$ l2 $\leq$ ... $\leq$ lk).
- Whenever there is a conflict between the function name Pol2Row and another name used in the same session, use the long form FREE['Pol2Row'].

EXAMPLES:

```
> with(FREE):
> Pol2Row(x1*x2^2 + q*x1^5*x7);
```

$$w[1, 2, 2] + q\, w[1, 1, 1, 1, 1, 7]$$

SEE ALSO: Free2Pol Pol2Free

FUNCTION: Sf2Free - from symmetric polynomials to words

CALLING SEQUENCE:

```
Sf2Free(sf, n)
FREE[Sf2Free](sf, n)
```

PARAMETERS:

sf = any symmetric polynomial
n = any nonnegative integer denoting the number of variables

SYNOPSIS:

- The Sf2Free function converts a symmetric polynomial sf in n variables into an element of the free algebra. The correspondance amounts to develop each Schur function as a sum of tableaux (seen as special words) of a given shape. Thus, the symmetric function is first expressed in the basis of Schur functions and then each Schur function is converted.
- Whenever there is a conflict between the function name Sf2Free and another name used in the same session, use the long form FREE['Sf2Free'].

EXAMPLES:

```
> with(FREE):
> Sf2Free(e3, 4);
```

$$w[3, 2, 1] + w[4, 2, 1] + w[4, 3, 1] + w[4, 3, 2]$$

```
> Sf2Free(s[2,1], 3);
```

$$w[2, 1, 1] + w[3, 1, 1] + w[2, 1, 2] + w[3, 1, 2]$$
$$+ w[2, 1, 3] + w[3, 1, 3] + w[3, 2, 2] + w[3, 2, 3]$$

See Also: `Pol2Free`

Function: `SgaOnFree` - action of the symmetric group algebra on words

Calling Sequence:

```
SgaOnFree(e_1, free)
FREE[SgaOnFree](e_1, free)
```

Parameters:

`e_1` = any element of the symmetric group algebra
`free` = any element of the free algebra

Synopsis:

- The SgaOnFree function realizes the action of the symmetric group algebra on the free algebra. This action is compatible with plactic congruences and Schensted algorithm.
- The elementary action of a simple transposition, say `A[2,1]`, on a word `v` is as follows: forget all letters different from 1 and 2, then cross out all successive factors [2,1] of what is left, in any order. The remaining word is of the type `v'` = `w[1$a, 2$b]`. Then the image of `v` is obtained by replacing its subword `v'` by `w[1$b, 2$a]`.
- In particular, the image of a tableau (resp. contretableau) is a tableau (resp. contretableau).
- Whenever there is a conflict between the function name `SgaOnFree` and another name used in the same session, use the long form `FREE['SgaOnFree']`.

Examples:

```
> with(FREE):
> SgaOnFree(A[1,3,2], w[2,2,3,1,1,2,2,3]);
```

```
                          w[2, 3, 3, 1, 1, 2, 3, 3]
```

See Also: `IdcaOnFree`

Function: `Table2Free` - convert a table into an element of the free algebra

Calling Sequence:

```
Table2Free(t)
FREE[Table2Free](t)
```

Parameters:

`t` = any table denoting an element of the free algebra

Synopsis:

- The `Table2Free` function converts a table into an element of the free algebra. The table should be indexed by lists of integers and each entry corresponding to a word `word` is interpreted as the coefficient of `w[word]` in the expression of the corresponding element in the free algebra.

- Whenever there is a conflict between the function name `Table2Free` and another name used in the same session, use the long form `FREE['Table2Free']`.

EXAMPLES:

```
> with(FREE):
> t:=table([[1,2,1]=q^2+q, [2,1,1]=1]);

                          t := table([
                                                2
                                 [1, 2, 1] = q  + q
                                 [2, 1, 1] = 1
                               ])

> Table2Free(t);

                           2
                        (q  + q) w[1, 2, 1] + w[2, 1, 1]
```

SEE ALSO: `Free2Table`

5.15 CLG, a package to handle characters of linear groups

HELP FOR: A package to handle characters of the linear group.

CALLING SEQUENCE:

```
CLG[<function>](args)
<function>(args)
```

SYNOPSIS:

- This package provides functions to manipulate characters of the linear group $GL(n)$. It essentially amounts to consider symmetric functions in a fixed number of variables.
- One chooses the order using the function `CLG_n` (default degree is 5).
- Whenever there is a conflict between a function name in `CLG` and another name used in the same session, use the long form `CLG`[<function>].
- The available functions are:

`CLG_n`	`Gauss`	`GlDimRep`	`Toe_n`
`Toh_n`	`Tom_n`	`Top_n`	`Tos_n`
`Tox_n`	`x2m_n`		

- For help with a particular function do either ?`CLG`[<function>] or ?`CLG`,<function> where <function> is one from the above list.

EXAMPLES:

```
> with(CLG):
> CLG_n(5);
```

$$5$$

```
> Tos_n(s[4,3,2,1,1]^2, collect);
```

$$3\,s_{[6,6,5,3,2]} + 2\,s_{[6,6,4,4,2]} + 3\,s_{[6,6,4,3,3]} + 3\,s_{[6,5,5,4,2]} + 3\,s_{[6,5,5,3,3]} + 4\,s_{[6,5,4,4,3]} + 4\,s_{[7,6,4,3,2]} + 3\,s_{[7,5,5,3,2]} + 2\,s_{[7,6,3,3,3]} + 3\,s_{[7,5,4,4,2]} + 4\,s_{[7,5,4,3,3]} + s_{[8,6,3,3,2]} + s_{[8,6,4,2,2]} + s_{[5,5,5,5,2]} + 2\,s_{[5,5,5,4,3]} + s_{[5,5,4,4,4]} + s_{[6,4,4,4,4]} + 2\,s_{[7,4,4,4,3]}$$

```
+ s[7, 7, 4, 2, 2] + s[7, 7, 3, 3, 2] + 2 s[7, 6, 5, 2, 2]

+ s[8, 5, 5, 2, 2] + 2 s[8, 5, 4, 3, 2] + s[8, 5, 3, 3, 3]

+ s[8, 4, 4, 4, 2] + s[8, 4, 4, 3, 3] + s[6, 6, 6, 2, 2]
```

FUNCTION: CLG_n - fix the order of the linear group

CALLING SEQUENCE:

```
CLG_n(n)
CLG[CLG_n](n)
```

PARAMETERS:

`n` = any positive integer

SYNOPSIS:

- The `CLG_n` function may be used to fix the order of the linear group to `n`. Default order is `_CLGn`=5.
- When called without any argument, it returns the current value of `_CLGn`.
- Whenever there is a conflict between the function name `CLG_n` and another name used in the same session, use the long form `CLG['CLG_n']`.

EXAMPLES:

```
> with(CLG):
> CLG_n(5);
```

5

SEE ALSO:

FUNCTION: Gauss - compute a Gaussian polynomial

CALLING SEQUENCE:

```
Gauss(m, n, 'q')
CLG[Gauss](m, n, 'q')
```

PARAMETERS:

`m` = any non-negative integer
`n` = any integer

SYNOPSIS:

- The Gauss function computes the Gaussian polynomial corresponding to the parameters `m` and `n`. This is a polynomial in the variable `'q'` that is the generalization of the corresponding binomial coefficient $\binom{m}{n}$:

- `Gauss(m,n,q)` = 0 whenever `n`<0 or `n`>`m`.
- `Gauss(m,n,q)` = 1 whenever `n`=0 or `n`=`m`.
- `Gauss(m,n,q)` = `Gauss(m-1,n-1,q)` + q^n * `Gauss(m-1,n,q)`.

- Whenever there is a conflict between the function name `Gauss` and another name used in the same session, use the long form `CLG['Gauss']`.

EXAMPLES:

```
> with(CLG):
> Gauss(4, 2, q);
```

$$q^4 + q^3 + 2q^2 + q + 1$$

```
> q:=2:
> Gauss(4, 2, q);
```

$$35$$

```
> Gauss(4, 2, 'q');
```

$$q^4 + q^3 + 2q^2 + q + 1$$

SEE ALSO:

FUNCTION: `GlDimRep` - compute the dimension of a representation

CALLING SEQUENCE:

```
GlDimRep(part)
CLG[GlDimRep](part)
```

PARAMETERS:

`part` = any list denoting a (skew) partition

SYNOPSIS:

- The `GlDimRep` function computes the dimension of the irreducible representation of index part, for the linear group of order `_CLGn`.
- When part is a skew partition, it computes the dimension of the representation indexed by the skew partition `part[1]/part[2]`.
- Adding the parameter `'q'`, it computes the specialization of `s[part]` or `s[part[1]/part[2]]` in `x_1=1`, `x_2=q`, ..., `x_n=q^n-1`.
- Whenever there is a conflict between the function name `GlDimRep` and another name used in the same session, use the long form `CLG['GlDimRep']`.

Examples:

```
> with(CLG):
> GlDimRep([2,2,1]);
```

$$75$$

```
> GlDimRep([[2,2,1], [1,1]]);
```

$$50$$

```
> GlDimRep([[3,1,1], [2,1]], 'q');
```

$$(q^4 + q^3 + q^2 + q + 1)^2$$

See Also:

Function: `Toe_n` - convert any symmetric function to a e-polynomial

Calling Sequence:

```
Toe_n(sf)
Toe_n(sf, b)
CLG[Toe_n](sf)
CLG[Toe_n](sf, b)
```

Parameters:

`sf` = any symmetric function
`b` = any name of a known basis

Synopsis:

- The `Toe_n` function converts any symmetric function to a e-polynomial. The `e.k`, for `k>_CLGn`, are sent to 0.
- 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: `Toe_n(sf, b, 'collect')`. For instance, `Toe_n(sf, 'e', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `Toe_n` and another name used in the same session, use the long form `CLG['Toe_n']`.

EXAMPLES:

```
> with(CLG):
> _CLGn;
                                5

> Toe_n((1+q)^5*s[3,2,2,1]):                    # expands the input
> Toe_n((1+q)^5*s[3,2,2,1],noexpand):           # does not expand (1+q)^5
> Toe_n((1+q)^5*s[3,2,2,1],collect):            # collects the result
> Toe_n((1+q)^5*s[3,2,2,1],noexpand,'s'):       # the most efficient
> Toe_n((1+q)^5*s[3,2,2,1],'s',collect):        # specifies a basis
> Toe_n((1+q)^5*s[3,2,2,1],noexpand,'s',collect):
> Toe_n(s[2,1]*h4 - 2*q*p3 + aa*e6);

                                  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` `Tom_n` `Top_n` `Tos_n`

FUNCTION: `Toh_n` - convert any symmetric function to a `h`-polynomial

CALLING SEQUENCE:

```
Toh_n(sf)
Toh_n(sf, b)
CLG[Toh_n](sf)
CLG[Toh_n](sf, b)
```

PARAMETERS:

`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Toh_n` function converts any symmetric function to a `h`-polynomial. It uses the relations `e.k` = 0 for `k>_CLGn`.
- 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: `Toh_n(sf, b, 'collect')`. For instance, `Toh_n(sf, 'h', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `Toh_n` and another name used in the same session, use the long form `CLG['Toh_n']`.

EXAMPLES:

```
> with(CLG):
> _CLGn;
```

```
                                5
```

```
> Toh_n((1+q)^5*s[3,2,2,1]):                    # expands the input
> Toh_n((1+q)^5*s[3,2,2,1],noexpand):           # does not expand (1+q)^5
> Toh_n((1+q)^5*s[3,2,2,1],collect):            # collects the result
> Toh_n((1+q)^5*s[3,2,2,1],noexpand,'s'):       # the most efficient
> Toh_n((1+q)^5*s[3,2,2,1],'s',collect):        # specifies a basis
> Toh_n((1+q)^5*s[3,2,2,1],noexpand,'s',collect):
> Toh_n(s[2,1]*h4 - 2*q*p3 + aa*e6);
```

```
                                                        3
            h4 h2 h1 - h4 h3 - 6 q h3 + 6 q h2 h1 - 2 q h1
```

SEE ALSO: TYP[Sf] Toe_n Tom_n Top_n Tos_n

FUNCTION: Tom_n - express any symmetric function in the basis of monomial symmetric functions

CALLING SEQUENCE:

```
Tom_n(sf)
Tom_n(sf, b)
CLG[Tom_n](sf)
CLG[Tom_n](sf, b)
```

PARAMETERS:

`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Tom_n` function expresses any symmetric function in the basis of monomial symmetric functions. It sends to 0 all those which are indexed by partitions with more than `_CLGn` parts.
- 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: `Tom_n(sf, b, 'collect')`. For instance, `Tom_n(sf, 'm', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `Tom_n` and another name used in the same session, use the long form `CLG['Tom_n']`.

EXAMPLES:

```
> with(CLG):
> _CLGn;

                                    5

> Tom_n((1+q)^5*e3*e4):                          # expands the input
> Tom_n((1+q)^5*e3*e4,noexpand):                 # does not expand (1+q)^5
> Tom_n((1+q)^5*e3*e4,collect):                  # collects the result
> Tom_n((1+q)^5*e3*e4,noexpand,'e'):             # the most efficient
> Tom_n((1+q)^5*e3*e4,'e',collect):              # specifies a basis
> Tom_n((1+q)^5*e3*e4,noexpand,'e',collect):
> Tom_n(s[2,1]*h4 - 2*q*p3 + aa*m[1,1,1,1,1,1]);

 m[6, 1] + 2 m[5, 2] + 4 m[5, 1, 1] + 2 m[4, 3] + 6 m[4, 2, 1]

 + 11 m[4, 1, 1, 1] + 6 m[3, 3, 1] + 8 m[3, 2, 2] + 14 m[3, 2, 1, 1]

 + 24 m[3, 1, 1, 1, 1] + 17 m[2, 2, 2, 1] + 28 m[2, 2, 1, 1, 1] - 2 q m[3]
```

SEE ALSO: Toe_n Toh_n Top_n Tos_n

FUNCTION: Top_n - convert any symmetric function to a p-polynomial

CALLING SEQUENCE:

```
Top_n(sf)
Top_n(sf, b)
CLG[Top_n](sf)
CLG[Top_n](sf, b)
```

PARAMETERS:

`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Top_n` function converts any symmetric function to a p-polynomial. It uses the relations `e.k` = 0 for `k>_CLGn`.
- 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: `Top_n(sf, b, 'collect')`. For instance, `Top_n(sf, 'p', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `Top_n` and another name used in the same session, use the long form `CLG['Top_n']`.

EXAMPLES:

```
> with(CLG):
> _CLGn;
```

```
                                5
```

```
> Top_n((1+q)^5*s[3,2,2,1]):                    # expands the input
> Top_n((1+q)^5*s[3,2,2,1],noexpand):           # does not expand (1+q)^5
> Top_n((1+q)^5*s[3,2,2,1],collect):            # collects the result
> Top_n((1+q)^5*s[3,2,2,1],noexpand,'s'):       # the most efficient
> Top_n((1+q)^5*s[3,2,2,1],'s',collect):        # specifies a basis
> Top_n((1+q)^5*s[3,2,2,1],noexpand,'s',collect):
> Top_n(s[2,1]*h4 - 2*q*p3 + aa*e6);
```

```
          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` `Tom_n` `Tos_n`

FUNCTION: `Tos_n` - express any symmetric function in the basis of Schur functions

CALLING SEQUENCE:

```
Tos_n(sf)
Tos_n(sf, b)
CLG[Tos_n](sf)
CLG[Tos_n](sf, b)
```

PARAMETERS:

`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Tos_n` function expresses any symmetric function in the basis of Schur functions, i.e. as a linear combination of Schur functions. It sends to 0 all Schur functions indexed by partitions with more than `_CLGn` parts.
- 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: `Tos_n(sf, b, 'collect')`. For instance, `Tos_n(sf, 's', 'collect')` may be used to collect the argument `sf`.
- Whenever there is a conflict between the function name `Tos_n` and another name used in the same session, use the long form `CLG['Tos_n']`.

EXAMPLES:

```
> with(CLG):
> _CLGn;
```

$$5$$

```
> Tos_n((1+q)^5*e3*e4):                          # expands the input
> Tos_n((1+q)^5*e3*e4,noexpand):                 # does not expand (1+q)^5
> Tos_n((1+q)^5*e3*e4,collect):                  # collects the result
> Tos_n((1+q)^5*e3*e4,noexpand,'e'):             # the most efficient
> Tos_n((1+q)^5*e3*e4,'e',collect):              # specifies a basis
> Tos_n((1+q)^5*e3*e4,noexpand,'e',collect):
> Tos_n(s[2,1]*h4 - 2*q*p3 + aa*s[1,1,1,1,1,1]);

          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` `Tom_n` `Top_n`

FUNCTION: `Tox_n` - express any symmetric function in the basis of monomials

CALLING SEQUENCE:

```
Tox_n(sf)
Tox_n(sf, b)
CLG[Tox_n](sf)
CLG[Tox_n](sf, b)
```

PARAMETERS:

`sf` = any symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `Tox_n` function expresses any symmetric function in the basis of monomials in x_1, x_2, ...
- 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: `Tox_n(sf, b, 'collect')`.
- Whenever there is a conflict between the function name `Tox_n` and another name used in the same session, use the long form `CLG['Tox_n']`.

EXAMPLES:

```
> with(CLG):
> CLG_n(3);
                                    3

> Tox_n((1+q)^3*e3*e2):                          # expands the input
> Tox_n((1+q)^3*e3*e2,noexpand):                 # does not expand (1+q)^3
> Tox_n((1+q)^3*e3*e2,collect):                  # collects the result
> Tox_n((1+q)^3*e3*e2,noexpand,'e'):             # the most efficient
> Tox_n((1+q)^3*e3*e2,'e',collect):              # specifies a basis
> Tox_n((1+q)^3*e3*e2,noexpand,'e',collect):
> Tox_n(p2^3);

    6       4   2       4   2       2   4       2   2   2       2   4     6
  x1  + 3 x1  x2  + 3 x1  x3  + 3 x1  x2  + 6 x1  x2  x3  + 3 x1  x3  + x2

            4   2       2   4     6
      + 3 x2  x3  + 3 x2  x3  + x3
```

SEE ALSO: `TYP[Sf]`

FUNCTION: `x2m_n` - from the basis of monomials to monomial symmetric function basis

CALLING SEQUENCE:

```
x2m_n(pol)
CLG[x2m_n](pol)
```

PARAMETERS:

`pol` = any expanded polynomial in $x_1, x_2, ...$ (assumed to be symmetrical)

SYNOPSIS:

- The `x2m_n` function converts any symmetric polynomial into a linear combination of monomial symmetric functions in `_CLGn` variables. It is assumed that `pol` is symmetric in the variables `x1`, `x2`, ..., `x.(_CLGn)`.
- Whenever there is a conflict between the function name `x2m_n` and another name used in the same session, use the long form `CLG['x2m_n']`.

EXAMPLES:

```
> with(CLG):
> CLG_n(3):
> x2m_n( expand( (1+x1) * (1+x2) * (1+x3) ) );

                      1 + m[1] + m[1, 1] + m[1, 1, 1]
```

SEE ALSO: `SP[x2X]`

5.16 Noncommutative and quasi-symmetric functions with NCSF

HELP FOR: A package for noncommutative and quasi-symmetric functions

CALLING SEQUENCE:

```
<function>(args)
NCSF[<function>](args)
```

SYNOPSIS:

- This package provides functions to work with noncommutative symmetric functions together with functions to handle quasi-symmetric functions.
- Whenever there is a conflict between a function name in NCSF and another name used in the same session, use the long form NCSF[<function>].
- The available functions are:

NcsfInternal	NcsfMat	NcsfPairing	NcsfPlethysm
NcsfTransAlpha	Qsf2x	QsfMat	Sf2Qsf
ToE	ToF	ToL	ToM
ToPh	ToPs	ToQPh	ToQPs
ToR	ToS		

- For help with a particular function do either ?NCSF[<function>] or ?NCSF,<function> where <function> is one from the above list.

EXAMPLES:

```
> with(NCSF):
> NcsfMat(3, 'Ps', 'S');
```

$$\begin{bmatrix} 3 & -1 & -2 & 1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

FUNCTION: NcsfInternal - internal product

CALLING SEQUENCE:

```
NcsfInternal(ncsf_1, ncsf_2)
```

`NCSF[NcsfInternal](ncsf_1, ncsf_2)`

PARAMETERS:
`ncsf_1, ncsf_2` = any noncommutative symmetric functions

SYNOPSIS:

- The `NcsfInternal` 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
  - `NcsfInternal(S[I],S[J])=0` if $i_1 + i_2 + \cdots + i_p \neq j_1 + j_2 + \cdots + j_q$, otherwise,
  - `NcsfInternal(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}`]... S[`m_{qp}`]`.
- The result is expressed in the `S`-basis.
- Whenever there is a conflict between the function name `NcsfInternal` and another name used in the same session, use the long form `NCSF['NcsfInternal']`.

EXAMPLES:

```
> with(NCSF):
> NcsfInternal(L[2,1,1]+Ph[2,2], R[2,1,1]);

                  3 S[2, 1, 1] - 4 S[2, 2] + 2 S[1, 1, 2]
```

SEE ALSO:

FUNCTION: `NcsfMat` - transition matrix from X-basis to Y-basis

CALLING SEQUENCE:
`NcsfMat(n,X,Y)`
`NCSF[NcsfMat](n,X,Y)`

PARAMETERS:
`X,Y` = two bases
`n` = any positive integer (`n`>0)

SYNOPSIS:

- The `NcsfMat` function computes the transition matrix from the X-basis to the Y-basis.
- In noncommutative theory, we defined following bases:
  - `L`-basis: noncommutative elementary symmetric functions,
  - `Ph`-basis: noncommutative power sums symmetric functions of the second kind,

  - Ps-basis: noncommutative power sums symmetric functions of the first kind,
  - R-basis: ribbon Schur functions,
  - S-basis: noncommutative complete symmetric functions.

- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name NcsfMat and another name used in the same session, use the long form NCSF['NcsfMat'].

EXAMPLES:

```
> with(NCSF):
> NcsfMat(3,'R','S');
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

SEE ALSO: QsfMat

FUNCTION: NcsfPairing - computes the scalar product of quasi-symmetric function and noncommutative symmetric function

CALLING SEQUENCE:

```
NcsfPairing(qsf,ncsf)
NcsfPairing(qsf,ncsf)
NCSF[NcsfPairing](qsf,ncsf)
NCSF[NcsfPairing](qsf,ncsf)
```

PARAMETERS:

qsf = any quasi-symmetric function
ncsf = any noncommutative symmetric function

SYNOPSIS:

- The NcsfPairing function computes the scalar product of a quasi-symmetric function and a noncommutative symmetric function. The scalar product is defined by:

$$< \texttt{F[I]}, \texttt{R[J]} >= 1 \qquad \text{if } I = J \text{ and } 0 \text{ otherwise}$$

  where I and J are two compositions, F is the dual basis of Ribbon Schur basis.

- qsf is any expression in terms of the basic quasi-symmetric functions.
- ncsf is any expression in terms of the basic noncommutative symmetric functions.

- The `qsf` and the `ncsf` functions are expanded. `qsf` (resp. `ncsf`) is expressed on `F` basis (resp. `R` basis). The result is not collected.
- In first argument (resp. second argument), we consider as coefficient noncommutative symmetric functions (resp. quasi-symmetric functions).
- One may add `'noexpand'` just after the argument `ncsf` to choose not to expand `qsf` and `ncsf` before treating it.
- One may collect the result by adding an argument: this is done by `NcsfPairing(qsf, ncsf, 'collect')`.
- Whenever there is a conflict between the function name `NcsfPairing` and another name used in the same session, use the long form `NCSF['NcsfPairing']`.

Examples:

```
> with(NCSF):
> NcsfPairing((1+q)^5*F[3,2],R[3,2]):            # expands the input
> NcsfPairing((1+q)^5*F[3,2],R[3,2],noexpand):   # does not expand (1+q)^5
> NcsfPairing((1+q)^5*F[3,2],R[3,2],collect):    # collects the result
> NcsfPairing(F[3]*R[3],F[3]*R[3]);

                                F[3] R[3]

> NcsfPairing((1-q)^5*F[3,2],R[3,2],noexpand,collect);

                                      5
                               (1 - q)
```

See Also: `ToR ToF`

Function: `NcsfPlethysm` - computes the plethysm of symmetric function and a quasi-symmetric function

Calling Sequence:

```
NcsfPlethysm(sf,qsf)
NcsfPlethysm(sf,qsf)
NCSF[NcsfPlethysm](sf,qsf)
NCSF[NcsfPlethysm](sf,qsf)
```

Parameters:

`sf` = any symmetric function
`qsf` = any quasi-symmetric function

SYNOPSIS:

- The `NcsfPlethysm` function computes the plethysm of a symmetric function and a quasi-symmetric function. The plethysm is defined by: `NcsfPlethysm(pk,M[i1,..,ir]) = M[k*i1, ...,k*ir]`
- `sf` is any expression in terms of the basic symmetric functions.
- `qsf` is any expression in terms of the basic quasi-symmetric functions.
- The `sf` and the `qsf` functions are expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `qsf` to choose not to expand `sf` and `qsf` before treating it.
- One may collect the result by adding an argument: `NcsfPlethysm(sf,qsf,'collect')`.
- Whenever there is a conflict between the function name `NcsfPlethysm` and another name used in the same session, use the long form `NCSF['NcsfPlethysm']`.

EXAMPLES:

```
> with(NCSF):
> NcsfPlethysm((1+q)^3*s[2,1],M[1,1]):          #          expands the input
> NcsfPlethysm((1+q)^3*s[2,1],M[1,1],noexpand):#   does not expand (1+q)^3
> NcsfPlethysm((1+q)^5*s[2,1],M[1,1],collect): #       collects the result
> NcsfPlethysm((1-q)^3*s[1,1],M[1,1],noexpand,collect);

                    3                            3
          - (- 1 + q)  M[1, 2, 1] - 3 (- 1 + q)  M[1, 1, 1, 1]

                      3                          3
             - (- 1 + q)  M[1, 1, 2] - (- 1 + q)  M[2, 1, 1]
```

SEE ALSO:

FUNCTION: `NcsfTransAlpha` - specialization $A/(1-q) \longrightarrow A$ or $A \longrightarrow A/(1-q)$

CALLING SEQUENCE:

```
NcsfTransAlpha(ncsf, specialize, q)
NCSF[NcsfTransAlpha](ncsf, specialize, q)
```

PARAMETERS:

`ncsf` = any noncommutative symmetric function
`specialize` = indicate the type of specialization
`q` = any formal parameter

SYNOPSIS:

- The `NcsfTransAlpha` function computes the direct transformation $A/(1-q) \longrightarrow A$ or the inverse transformation $A \longrightarrow A/(1-q)$. A is a set of variables on the noncommutative symmetric function `ncsf`.
- The `NcsfTransAlpha` function computes the direct transformation $A/(1-q) \longrightarrow A$ when the second argument matches `'direct'`.
- The `NcsfTransAlpha` function computes the inverse transformation $A \longrightarrow A/(1-q)$ when the second argument matches `'inverse'`.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function `ncsf` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `q` to choose not to expand the noncommutative symmetric function `ncsf` before treating it.
- One may collect the result by adding a fourth argument: this is done by `NcsfTransAlpha(ncsf, specialize, q, 'collect')`.
- Whenever there is a conflict between the function name `NcsfTransAlpha` and another name used in the same session, use the long form `NCSF['NcsfTransAlpha']`.

EXAMPLES:

```
> with(NCSF):
> NcsfTransAlpha((1-q)^3*S[2,1],direct,'q'):          # expand the input
> NcsfTransAlpha((1-q)^3*S[2,1],direct,'q',noexpand): # does not expand
                                                      # the input
> NcsfTransAlpha((1-q)^3*S[2,1],direct,'q',collect):  # collect the result
> NcsfTransAlpha((1-q)^3*S[2,1],direct,'q',noexpand): # the most efficient
> NcsfTransAlpha((1-q)^3*S[2,1],direct,'q',noexpand,collect);

                          S[2, 1] (q - 1)   S[1, 1, 1] q
                        - --------------- + ------------
                              q + 1             q + 1

> NcsfTransAlpha((1-q)^3*S[2,1],inverse,'q',noexpand,collect);

                     5
              (q - 1)  (- R[3] - R[2, 1] + q R[1, 2] + q R[1, 1, 1])
```

SEE ALSO:

FUNCTION: `Qsf2x` - from quasi-symmetric function to the basis of monomials

CALLING SEQUENCE:

```
Qsf2x(qsf, var)
NCSF[Qsf2x](qsf, var)
```

PARAMETERS:
`qsf` = any quasi-symmetric function
`var` = a list of variables

SYNOPSIS:

- The `Qsf2x` functions converts a quasi-symmetric function to the basis of monomials.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function `qsf` is expanded and the result is not collected.
- One may add `'noexpand'` just after the argument `var` to choose not to expand the quasi-symmetric function `qsf` before treating it.
- Whenever there is a conflict between the function name `Qsf2x` and another name used in the same session, use the long form `NCSF['Qsf2x']`.

EXAMPLES:

```
> with(NCSF):
> Qsf2x(F[2]&*M[1,1],[x1,x2,x3]);
```

$$x1^3\ x2 + x1^3\ x3 + x2^3\ x3 + 3\ x1^2\ x2\ x3 + 3\ x1\ x2^2\ x3 + x1^2\ x2^2 + x1^2\ x3^2 + x2^2\ x3^2 + 3\ x1\ x2\ x3^2 + x1\ x2^2\ x3 + x1\ x2\ x3^2 + x1^2\ x2\ x3$$

SEE ALSO: `ToE ToQPh ToQPs ToF ToM`

FUNCTION: `QsfMat` - transition matrix from X-basis to Y-basis

CALLING SEQUENCE:
`QsfMat(n,X,Y)`
`NCSF[QsfMat](n,X,Y)`

PARAMETERS:
`X,Y` = two bases
`n` = any positive integer (`n`>0)

SYNOPSIS:

- The `QsfMat` function computes the transition matrix from the X-basis to the Y-basis, where X and Y being bases of quasi-symetric functions.
- In quasi-symmetric theory, we defined following bases:
  - `E`-basis: elementary quasi-symmetric functions,
  - `QPh`-basis: dual basis of noncommutative power sums symmetric functions of the second kind,

  - QPs-basis: dual basis of noncommutative power sums symmetric functions of the first kind,
  - F-basis: ribbon Schur quasi-symmetric function
  - M-basis: monomial quasi-symmetric functions.

- The result is a $2^{(n-1)} \times 2^{(n-1)}$ square matrix.
- Whenever there is a conflict between the function name QsfMat and another name used in the same session, use the long form NCSF['QsfMat'].

Examples:

```
> with(NCSF):
> QsfMat(3,'F','M');
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See Also: NcsfMat

Function: Sf2Qsf - expand a symmetric function on F-basis (dual basis of noncommutative ribbon Schur function)

Calling Sequence:

```
Sf2Qsf(sf)
NCSF[Sf2Qsf](sf)
```

Parameters:

sf = any symmetric function

Synopsis:

- The Sf2Qsf function computes the expansion of sf on F basis (dual basis of noncommutative ribbon Schur function).
- The result is expressed on F basis.
- The input is any expression in terms of the basic symmetric functions.
- The symmetric function sf is expanded and the result is not collected.
- One may add 'noexpand' just after sf to choose not to expand the symmetric function sf before treating it.
- One may collect the result by adding a third argument: this is done by Sf2Qsf(sf,'collect').
- Whenever there is a conflict between the function name Sf2Qsf and another name used in the same session, use the long form NCSF['Sf2Qsf'].

Examples:

```
> with(NCSF):
> Sf2Qsf((1-q)^3*s[2,1]):                  # expand the input
> Sf2Qsf((1-q)^3*s[2,1],noexpand):         # does not expand the input
> Sf2Qsf((1-q)^3*s[2,1],collect):          # collect the result
> Sf2Qsf((1-q)^3*s[2,1]+s[3],noexpand,collect);

                        3                    3
                (1 - q)   F[2, 1] + (1 - q)   F[1, 2] + F[3]
```

See Also: NcsfPairing ToF

Function: ToE - express any quasi-symmetric function in the E-basis (dual basis of noncommutative elementary symmetric function)

Calling Sequence:

```
ToE(qsf)
ToE(qsf, b)
NCSF[ToE](qsf)
NCSF[ToE](qsf, b)
```

Parameters:

qsf = any quasi-symmetric function
b = any name of a known basis

Synopsis:

- The ToE function computes the expansion of qsf in the E-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call ToE(qsf, 'E') does not affect the argument qsf.
- One may add 'noexpand' just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: ToE(qsf, b, 'collect'). For instance, ToE(qsf, 'E', 'collect') may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToE and another name used in the same session, use the long form NCSF['ToE'].

Examples:

```
> with(NCSF):
> ToE((1+q)^2*F[2,1]*F[2]):                      # expands the input
> ToE((1+q)^2*F[2,1]*F[2],noexpand):             # does not expand (1+q)^2
> ToE((1+q)^2*F[2,1]*F[2],collect):              # collects the result
> ToE((1+q)^5*E[2,3]*E[1,2],'E'):                # specifies a basis
> ToE((1+q)^2*F[2,1]*F[2],noexpand,'F',collect); # the most efficient
> ToE((1-q)^2*F[1,2,1]+QPh[2]&*F[1,1],noexpand,collect);

                          2                            2
   - E[1, 2, 1] + (1 - q)  E[1, 1, 1, 1] + ((1 - q)  - 2) E[2, 2]

                    2                              2
   + (- 1 + (1 - q) ) E[2, 1, 1] + (- 1 + (1 - q) ) E[1, 1, 2] - E[4]

   - E[1, 3] - E[3, 1]
```

See Also: `ToQPh ToQPs ToF ToM`

Function: `ToF` - express any quasi-symmetric function in the `F`-basis (dual basis of Ribbon Schur basis)

Calling Sequence:

```
ToF(qsf)
ToF(qsf, b)
NCSF[ToF](qsf)
NCSF[ToF](qsf, b)
```

Parameters:

`qsf` = any quasi-symmetric function
`b` = any name of a known basis

Synopsis:

- The `ToF` function computes the expansion of `qsf` in the `F`-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function `qsf` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `qsf` is solely expressed in terms of the known basis `b`.
- The call `ToF(qsf, 'F')` does not affect the argument `qsf`.
- One may add `'noexpand'` just after the argument `qsf` to choose not to expand the quasi-symmetric function `qsf` before treating it.
- One may collect the result by adding a third argument: `ToF(qsf, b, 'collect')`. For instance, `ToF(qsf, 'F', 'collect')` may be used to collect the argument `qsf`.

- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToF` and another name used in the same session, use the long form `NCSF['ToF']`.

Examples:

```
> with(NCSF):
> ToF((1+q)^5*QPh[2,1]+t*M[3],noexpand,collect);
```

$$(-1/2\,(1+q)^5 - t)\,F[1,2] + (-1/2\,(1+q)^5 + t)\,F[1,1,1] + (1/2\,(1+q)^5 + t)\,F[3] + (1/2\,(1+q)^5 - t)\,F[2,1]$$

See Also: `ToE ToPs ToPh ToM`

Function: `ToL` - express any noncommutative symmetric function in the L-basis (noncommutative elementary symmetric function)

Calling Sequence:

```
ToL(ncsf)
ToL(ncsf, b)
NCSF[ToL](ncsf)
NCSF[ToL](ncsf, b)
```

Parameters:

`ncsf` = any noncommutative symmetric function
`b` = any name of a known basis

Synopsis:

- The `ToL` function computes the expansion of `ncsf` in the L-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function `ncsf` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `ncsf` is solely expressed in terms of the known basis `b`.
- The call `ToL(ncsf, 'L')` does not affect the argument `ncsf`.
- One may add `'noexpand'` just after the argument ncsf to choose not to expand the noncommutative symmetric function `ncsf` before treating it.
- One may collect the result by adding a third argument: `ToL(ncsf, b, 'collect')`. For instance, `ToL(ncsf, 'L', 'collect')` may be used to collect the argument `ncsf`.
- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToL` and another name used in the same session, use the long form `NCSF['ToL']`.

EXAMPLES:

```
> with(NCSF):
> ToL((1+q)^5*R[2,3]):                          # expands the input
> ToL((1+q)^5*R[2,3],noexpand):                 # does not expand (1+q)^5
> ToL((1+q)^5*R[2,3],collect):                  # collects the result
> ToL((1+q)^5*R[2,3],'R'):                      # specifies a basis
> ToL((1+q)^5*R[2,3],noexpand,'R',collect):     # the most efficient
> ToL((1-q)^3*S[3]+S[1,2],noexpand,'S',collect);

                3                3                       3
       (1 - q)  L[3] - (1 - q)  L[2, 1] + (- (1 - q)  - 1) L[1, 2]

                    3
           + ((1 - q)  + 1) L[1, 1, 1]
```

SEE ALSO: ToPh ToPs ToR ToS

FUNCTION: ToM - express any quasi-symmetric function in the M-basis (quasi-monomial symmetric functions)

CALLING SEQUENCE:

```
ToM(qsf)
ToM(qsf, b)
NCSF[ToM](qsf)
NCSF[ToM](qsf, b)
```

PARAMETERS:

qsf = any quasi-symmetric function
b = any name of a known basis

SYNOPSIS:

- The ToM function computes the expansion of qsf in the M-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call ToM(qsf, 'M') does not affect the argument qsf.
- One may add 'noexpand' just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: ToM(qsf, b, 'collect'). For instance, ToM(qsf, 'M', 'collect') may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the &* operator.

- Whenever there is a conflict between the function name ToM and another name used in the same session, use the long form NCSF['ToM'].

EXAMPLES:

```
> with(NCSF):
> ToM((1+q)^5*QPh[2,1,2]+t^3*M[3,2],noexpand,collect);

                  5                    5                    5
        1/6 (1 + q)  M[5] + 1/2 (1 + q)  M[2, 3] + (1 + q)  M[2, 1, 2]

                       5    3
          + (1/2 (1 + q)  + t ) M[3, 2]
```

SEE ALSO: ToE ToF ToPh ToPs

FUNCTION: ToPh - express any noncommutative symmetric function in the Ph-basis (noncommutative power sums symmetric function of the second kind)

CALLING SEQUENCE:

```
ToPh(ncsf)
ToPh(ncsf, b)
NCSF[ToPh](ncsf)
NCSF[ToPh](ncsf, b)
```

PARAMETERS:

`ncsf` = any noncommutative symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The ToPh function computes the expansion of ncsf in the Ph-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function ncsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that ncsf is solely expressed in terms of the known basis b.
- The call ToPh(ncsf, 'Ph') does not affect the argument ncsf.
- One may add 'noexpand' just after the argument ncsf to choose not to expand the noncommutative symmetric function ncsf before treating it.
- One may collect the result by adding a third argument: ToPh(ncsf, b, 'collect'). For instance, ToPh(ncsf, 'Ph', 'collect') may be used to collect the argument ncsf.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToPh and another name used in the same session, use the long form NCSF['ToPh'].

EXAMPLES:

```
> with(NCSF):
> ToPh((1+q)^5*Ps[2,3]):                        # expands the input
> ToPh((1+q)^5*Ps[2,3],noexpand):               # does not expand (1+q)^5
> ToPh((1+q)^5*Ps[2,3],collect):                # collects the result
> ToPh((1+q)^5*Ps[2,3],'Ph'):                   # specifies a basis
> ToPh((1+q)^5*Ps[2,3],noexpand,'Ps',collect): # the most efficient
> ToPh((1-q)^3*S[3]+Ps[1,2],noexpand,collect);

                3                       3
     1/3 (1 - q)  Ph[3] + 1/4 (1 - q)  Ph[2, 1]

                    3                               3
       + (1/4 (1 - q)  + 1) Ph[1, 2] + 1/6 (1 - q)  Ph[1, 1, 1]
```

SEE ALSO: ToL ToPs ToR ToS

FUNCTION: ToPs - express any noncommutative symmetric function in the Ps-basis (noncommutative power sums symmetric function of the first kind)

CALLING SEQUENCE:

```
ToPs(ncsf)
ToPs(ncsf, b)
NCSF[ToPs](ncsf)
NCSF[ToPs](ncsf, b)
```

PARAMETERS:

`ncsf` = any noncommutative symmetric function
`b` = any name of a known basis

SYNOPSIS:

- The `ToPs` function computes the expansion of `ncsf` in the `Ps`-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function `ncsf` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `ncsf` is solely expressed in terms of the known basis `b`.
- The call `ToPs(ncsf, 'Ps')` does not affect the argument `ncsf`.
- One may add `'noexpand'` just after the argument `ncsf` to choose not to expand the noncommutative symmetric function `ncsf` before treating it.
- One may collect the result by adding a third argument: `ToPs(ncsf, b, 'collect')`. For instance, `ToPs(ncsf, 'Ps', 'collect')` may be used to collect the argument `ncsf`.
- The noncommutative multiplication is denoted by the `&*` operator.

- Whenever there is a conflict between the function name ToPs and another name used in the same session, use the long form NCSF['ToPs'].

Examples:

```
> with(NCSF):
> ToPs((1+q)^5*Ph[2,3]):                        # expands the input
> ToPs((1+q)^5*Ph[2,3],noexpand):               # does not expand (1+q)^5
> ToPs((1+q)^5*Ph[2,3],collect):                # collects the result
> ToPs((1+q)^5*Ph[2,3],'Ph'):                   # specifies a basis
> ToPs((1+q)^5*Ph[2,3],noexpand,'Ph',collect): # the most efficient
> ToPs((1-q)^3*S[3]+R[1,2],noexpand,collect);

              3                                  3
   (1/3 (1 - q)  + 1/6) Ps[1, 2] + (1/3 (1 - q)  - 1/3) Ps[3]

                    3                                  3
      + (1/6 (1 - q)  - 1/6) Ps[2, 1] + (1/6 (1 - q)  + 1/3) Ps[1, 1, 1]
```

See Also: ToL ToPh ToR ToS

Function: ToQPh - express any quasi-symmetric function in the QPh-basis (dual basis of noncommutative power sums symmetric function of the second kind)

Calling Sequence:

```
ToQPh(qsf)
ToQPh(qsf, b)
NCSF[ToQPh](qsf)
NCSF[ToQPh](qsf, b)
```

Parameters:

qsf = any quasi-symmetric function
b = any name of a known basis

Synopsis:

- The ToQPh function computes the expansion of qsf in the QPh-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function qsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that qsf is solely expressed in terms of the known basis b.
- The call ToQPh(qsf, 'QPh') does not affect the argument qsf.
- One may add 'noexpand' just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.

- One may collect the result by adding a third argument: `ToQPh(qsf, b, 'collect')`. For instance, `ToQPh(qsf, 'QPh', 'collect')` may be used to collect the argument `qsf`.
- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToQPh` and another name used in the same session, use the long form `NCSF['ToQPh']`.

Examples:

```
> with(NCSF):
> ToQPh((1+q)^2*QPs[2,1]):                        # expands the input
> ToQPh((1+q)^2*QPs[2,1],noexpand):               # does not expand (1+q)^5
> ToQPh((1+q)^2*QPs[2,1],collect):                # collects the result
> ToQPh((1+q)^2*QPh[2,1]*QPh[2],'QPh'):           # specifies a basis
> ToQPh((1+q)^2*QPh[2,1],noexpand,'QPh',collect):# the most efficient
> ToQPh((1+q)^2*QPs[2,1]+M[3,1]+t^3*M[2,2],noexpand,collect);

                      2                    2              3
        - 1/2 (1 + q)  QPh[3] + 3 (1 + q)  QPh[2, 1] + t  QPh[2, 2]

                         3
           + (- 1/2 - 1/2 t ) QPh[4] + QPh[3, 1]
```

See Also: `ToE ToQPs ToF ToM`

Function: `ToQPs` - express any quasi-symmetric function in the `QPs`-basis (dual basis of noncommutative power sums symmetric functions of the first kind)

Calling Sequence:

```
ToQPs(qsf)
ToQPs(qsf, b)
NCSF[ToQPs](qsf)
NCSF[ToQPs](qsf, b)
```

Parameters:

`qsf` = any quasi-symmetric function
`b` = any name of a known basis

Synopsis:

- The `ToQPs` function computes the expansion of `qsf` in the `QPs`-basis.
- The input is any expression in terms of the basic quasi-symmetric functions.
- The quasi-symmetric function `qsf` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `qsf` is solely expressed in terms of the known basis `b`.
- The call `ToQPs(qsf, 'QPs')` does not affect the argument `qsf`.

- One may add 'noexpand' just after the argument qsf to choose not to expand the quasi-symmetric function qsf before treating it.
- One may collect the result by adding a third argument: ToQPs(qsf, b, 'collect'). For instance, ToQPs(qsf, 'QPs', 'collect') may be used to collect the argument qsf.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToQPs and another name used in the same session, use the long form NCSF['ToQPs'].

Examples:

```
> with(NCSF):
> ToQPs((1+q)^2*M[2,1]):                          # expands the input
> ToQPs((1+q)^2*M[2,1],noexpand):                 # does not expand (1+q)^5
> ToQPs((1+q)^2*M[2,1],collect):                  # collects the result
> ToQPs((1+q)^2*QPs[2,1]*QPs[2],'QPs'):           # specifies a basis
> ToQPs((1+q)^2*QPs[2,1],noexpand,'QPs',collect):# the most efficient
> ToQPs((1+q)^5*QPh[2,2]+M[4]+t^3*M[3,1],noexpand,collect);

                3                     3                        5
      (- 1/4 t  + 1) QPs[4] + 1/4 t  QPs[3, 1] + 1/2 (1 + q)  QPs[2, 2]
```

See Also: ToE ToQPh ToF ToM

Function: ToR - express any noncommutative symmetric function in the R-basis (ribbon Schur function)

Calling Sequence:

```
ToR(ncsf)
ToR(ncsf, b)
NCSF[ToR](ncsf)
NCSF[ToR](ncsf, b)
```

Parameters:

ncsf = any noncommutative symmetric function
b = any name of a known basis

Synopsis:

- The ToR function computes the expansion of ncsf in the R-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function ncsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that ncsf is solely expressed in terms of the known basis b.
- The call ToR(ncsf, 'R') does not affect the argument ncsf.

- One may add 'noexpand' just after the argument ncsf to choose not to expand the noncommutative symmetric function ncsf before treating it.
- One may collect the result by adding a third argument: ToR(ncsf, b, 'collect'). For instance, ToR(ncsf, 'R', 'collect') may be used to collect the argument ncsf.
- The noncommutative multiplication is denoted by the &* operator.
- Whenever there is a conflict between the function name ToR and another name used in the same session, use the long form NCSF['ToR'].

EXAMPLES:

```
> with(NCSF):
> ToR((1+q)^5*L[2,3]):                      # expands the input
> ToR((1+q)^5*L[2,3],noexpand):             # does not expand (1+q)^5
> ToR((1+q)^5*L[2,3],collect):              # collects the result
> ToR((1+q)^5*L[2,3],'L'):                  # specifies a basis
> ToR((1+q)^5*L[2,3],noexpand,'L',collect): # the most efficient
> ToR((1+q)^5*L[2,3]+L[2,3],noexpand,'L',collect):
> ToR((1-q)^3*S[3]+L[1,2],noexpand,collect);
```

$$(1 - q)^3 R[3] + R[2, 1] + R[1, 1, 1]$$

SEE ALSO: ToL ToPh ToPs ToS

FUNCTION: ToS - express any noncommutative symmetric function in the S-basis (noncommutative complete symmetric function)

CALLING SEQUENCE:

```
ToS(ncsf)
ToS(ncsf, b)
NCSF[ToS](ncsf)
NCSF[ToS](ncsf, b)
```

PARAMETERS:

ncsf = any noncommutative symmetric function
b = any name of a known basis

SYNOPSIS:

- The ToS function computes the expansion of ncsf in the S-basis.
- The input is any expression in terms of the basic noncommutative symmetric functions.
- The noncommutative symmetric function ncsf is expanded and the result is not collected.
- One may specify by a second argument, say b, that ncsf is solely expressed in terms of the known basis b.

- The call `ToS(ncsf, 'S')` does not affect the argument `ncsf`.
- One may add `'noexpand'` just after the argument `ncsf` to choose not to expand the noncommutative symmetric function `ncsf` before treating it.
- One may collect the result by adding a third argument: `ToS(ncsf, b, 'collect')`. For instance, `ToS(ncsf, 'S', 'collect')` may be used to collect the argument `ncsf`.
- The noncommutative multiplication is denoted by the `&*` operator.
- Whenever there is a conflict between the function name `ToS` and another name used in the same session, use the long form `NCSF['ToS']`.

EXAMPLES:

```
> with(NCSF):
> ToS((1+q)^5*L[2,3]):                        # expands the input
> ToS((1+q)^5*L[2,3],noexpand):               # does not expand (1+q)^5
> ToS((1+q)^5*L[2,3],collect):                # collects the result
> ToS((1+q)^5*L[2,3],'L'):                    # specifies a basis
> ToS((1+q)^5*L[2,3],noexpand,'L',collect):   # the most efficient
> ToS((1-q)^3*Ps[3]+L[1,2],noexpand,collect);

                3                     3                               3
      - (1 - q)  S[2, 1] + ((1 - q)  + 1) S[1, 1, 1] + 3 (1 - q)  S[3]

                      3
          + (- 2 (1 - q)  - 1) S[1, 2]
```

SEE ALSO: `ToL ToPh ToPs ToR`

5.17 Using TAB to manipulate tableaux

HELP FOR: A package for tableaux and contretableaux.

CALLING SEQUENCE:
`TAB[<function>](args)`
`<function>(args)`

SYNOPSIS:

- This package provides functions to work on tableaux and contretableaux. It includes the (nilplactic) Bump process on tableaux and contretableaux, the bijection between words and pairs of tableaux. It also allows to enumerate tableaux, to compute idempotents of the symmetric group algebra and to act on rows (resp. columns) of a tableau (contretableau).
- To get more information on tableaux do either ?`TYP,Tableau` or ?`TYP[Tableau]`.
- To get more information on contretableaux do either ?`TYP,CTableau` or ?`TYP[CTableau]`.
- Whenever there is a conflict between a function name in TAB and another name used in the same session, use the long form `TAB`[<function>].
- The available functions are:

Bump	BumpC	CTab2Mat	CTab2Part
CTab2Word	InvSchensted	ListStdTab	ListTab
ListYama	Mat2CTab	Mat2Tab	Part2ListStdTab
PermOnCTab	PermOnTab	Schensted	StdTab2SgaIdemp
StdTab2Yama	Tab2Mat	Tab2Part	Tab2Word
Word2CTab	Word2Charge	Word2CoCharge	Word2Key
Word2Tab	Yama2StdTab		

- For help with a particular function do either ?`TAB`[<function>] or ?`TAB`,<function> where <function> is one from the above list.

EXAMPLES:

```
> with(TAB):
> CTab2Mat( Word2CTab(w[5,2,4,1,3,3,2,5,4,5,1,7,4,3,1,5,1,1]) );
```

$$\begin{bmatrix} 2 & 3 & 5 & 5 & 5 & 7 \\ & & & & & \\ & 1 & 4 & 4 & 4 & 5 \\ & & & & & \\ & & 1 & 1 & 3 & 3 \end{bmatrix}$$

```
[          ]
[       1  2 ]
[          ]
[          1 ]
```

FUNCTION: 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 word

SYNOPSIS:

- The Bump function inserts a word [x1, x2, ...] into a tableau from the right. It first inserts x1, then x2, and so on.
- The insertion algorithm is due to Schensted.
- When called with the 'nil' option as third argument, the Bump function nilplactically inserts the word into the tableau from the right.
- The nilplactic insertion algorithm makes Bump([], w[1,2,1], 'nil') equal to Bump([], w[2,1,2], 'nil') (Coxeter relations). Whenever the result of the insertion is not a reduced decomposition, it returns [].
- 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, w[4,1]));
```

```
[ 3       ]
[         ]
[ 2  6    ]
[         ]
[ 1  4  8 ]
```

```
> t:=[[2,3], [1,2]]:
> Tab2Mat(Bump(t, w[3,1], 'nil'));
```

```
[ 3       ]
[         ]
[ 2  3    ]
[         ]
[ 1  2  3 ]
```

See Also: TYP[IsTab] TYP[IsWord] Tab2Mat Word2Tab

Function: 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 word

Synopsis:

- The BumpC function inserts a word [x1, x2, ...] into a contretableau from the right. It first inserts x1, then x2, and so on.
- The insertion algorithm is due to Schensted.
- When called with the 'nil' option as third argument, the BumpC function nilplactically inserts the word into the contretableau from the right.
- The nilplactic insertion algorithm makes BumpC([], w[1,2,1], 'nil') to BumpC([], w[2,1, 2], 'nil'). Whenever the result of the insertion is not a reduced decomposition, it returns [].
- 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, w[6,4]));
```

$$\begin{bmatrix} 4 & 5 & 7 \\ 2 & 3 & 6 \\ & & 4 \end{bmatrix}$$

```
> ct:=[[2,1], [3,2]]:
> CTab2Mat(BumpC(ct, w[3,1], 'nil'));
```

$$\begin{bmatrix} 1 & 2 & 3 \\ & 1 & 2 \\ & & 1 \end{bmatrix}$$

See Also: CTab2Mat TYP[IsCTab] Word2CTab

Function: 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]]);
```

$$\begin{bmatrix} 5 & 5 & 6 \\ & & \\ & 4 & 4 \\ & & \\ & & 2 \\ & & \\ & & 1 \end{bmatrix}$$

See Also: Mat2CTab TYP[CTableau]

Function: 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):
> ctp:=[[1], [2,1], [5,4,2,1], [5,4,3,2,1]]:
> CTab2Part(ctp);
```

```
                              [5, 4, 2, 1]
```

```
> CTab2Mat(ctp);
```

```
                              [ 1  2  5  5 ]
                              [             ]
                              [    1  4  4 ]
                              [             ]
                              [       2  3 ]
                              [             ]
                              [       1  2 ]
                              [             ]
                              [          1 ]
```

SEE ALSO: `TYP[CTableau]` `Tab2Part`

FUNCTION: `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):
> ctw:=[[2,1], [4,3], [7,6,5]]:
> CTab2Word(ctw);
```

```
                          [2, 1, 4, 3, 7, 6, 5]
```

```
> CTab2Mat(ctw);
```

```
[ 2  4  7 ]
[         ]
[ 1  3  6 ]
[         ]
[       5 ]
```


See Also: CTab2Mat

Function: InvSchensted - inverse Schensted correspondence

Calling Sequence:

```
InvSchensted(pair)
InvSchensted(pair, 'left')
InvSchensted(pair, 'nil')
TAB[InvSchensted](pair)
TAB[InvSchensted](pair, 'left')
TAB[InvSchensted](pair, 'nil')
```

Parameters:

`pair` = any list of one tableau and one standard tableau

Synopsis:

- The `InvSchensted` function gives the bijection between pairs of tableaux and words. The argument `pair` is a list of two tableaux of the same shape, the second one being standard and called the (right) insertion tableau. The result is the word corresponding to this pair of tableaux.
- When called with a second argument `'left'`, the standard tableau is the left insertion tableau.
- When called with a second argument `'nil'`, the first tableau is a nil-plactic tableau.
- Whenever there is a conflict between the function name `InvSchensted` and another name used in the same session, use the long form `TAB['InvSchensted']`.

Examples:

```
> with(TAB):
> r := Schensted(w[1,4,2,4,2]);

            r := [[[4, 4], [1, 2, 2]], [[3, 5], [1, 2, 4]]]

> InvSchensted(r);

                           w[1, 4, 2, 4, 2]

> InvSchensted(r, 'left');

                           w[4, 1, 4, 2, 2]

> InvSchensted([[[4],[2,5],[1,2,3]], [[6],[3,4],[1,2,5]]], 'nil');
```

```
                          w[4, 5, 1, 2, 3, 1]
```

See Also: Schensted

Function: ListStdTab - generate all standard tableaux

Calling Sequence:

```
ListStdTab(n)
TAB[ListStdTab](n)
```

Parameters:

`n` = any positive integer

Synopsis:

- The `ListStdTab` function generates all standard tableaux on `n` letters.
- When called with a second parameter, say `'nb'`, it returns the number of standard tableaux on `n` letters.
- Whenever there is a conflict between the function name `ListStdTab` and another name used in the same session, use the long form `TAB['ListStdTab']`.

Examples:

```
> with(TAB):
> ListStdTab(3);

         [[[1, 2, 3]], [[2], [1, 3]], [[3], [1, 2]], [[3], [2], [1]]]

> map(Tab2Mat, ListStdTab(3));

                                                  [ 3 ]
                                [ 2    ]  [ 3    ]  [   ]
                [[ 1  2  3 ], [      ], [      ], [ 2 ]]
                                [ 1  3 ]  [ 1  2 ]  [   ]
                                                  [ 1 ]

> ListStdTab(15, 'nb');

                                 10349536
```

See Also: ListTab Part2ListStdTab TYP[Tableau]

FUNCTION: ListTab - generate all tableaux of a given shape and evaluation

CALLING SEQUENCE:

```
ListTab(part, eval)
TAB[ListTab](part, eval)
```

PARAMETERS:

`part` = any list denoting a partition
`eval` = any list denoting an evaluation

SYNOPSIS:

- The `ListTab` function generates all tableaux of a given shape `part` and of a given evaluation `eval`. Note that this vector eval possibly has some 0.
- When called with a third parameter, say `'nb'`, it returns the number of such tableaux.
- Using a third parameter `'mintab'=list_of_lists`, one specifies that the entries of the tableaux are componentwise bigger or equal to those of `'mintab'`: `'mintab'=[[6], [-infinity, 2]]` as well as `'mintab'=[[6], [1,2]]`, gives tableaux `[..., [x, ...], [y, z, ...]]` with $x \geq 6$, $y \geq 1$, $z \geq 2$.
- One can use both `'nb'` and `'mintab'` options as third or fourth argument at the same time.
- The generator is more efficient when `eval` is sorted as an increasing sequence of integers. It can be useful if you solely want the number of tableaux because sorting the evaluation does not change this number.
- Whenever there is a conflict between the function name `ListTab` and another name used in the same session, use the long form `TAB['ListTab']`.

EXAMPLES:

```
> with(TAB):
> l:=ListTab([3,2], [0,1,0,0,1,2,0,0,1]);

   l := [[[6, 9], [2, 5, 6]], [[5, 9], [2, 6, 6]], [[6, 6], [2, 5, 9]]]

> ListTab([3,2],  [0,1,0,0,1,2,0,0,1], 'mintab'=[[6],[-infinity]]);

                 [[[6, 9], [2, 5, 6]], [[6, 6], [2, 5, 9]]]

> map(Tab2Mat, l);

                  [ 6  9    ]  [ 5  9    ]  [ 6  6    ]
                 [[         ], [         ], [         ]]
                  [ 2  5  6 ]  [ 2  6  6 ]  [ 2  5  9 ]

> ListTab([5,5], [1$10], 'nb');
```

See Also: ListStdTab

Function: ListYama - generate all Yamanouchi words of a given evaluation

Calling Sequence:
```
ListYama(part)
TAB[ListYama](part)
```

Parameters:
part = any list denoting a partition

Synopsis:

- The ListYama function computes all Yamanouchi words (also called "lattice permutations") corresponding to a given partition part. By default, it lists left Yamanouchi words. Left and right Yamanouchi words code standard tableaux.
- ListYama(part) = ListYama(part, 'left') but one can also list all right Yamanouchi words of a given evaluation by using ListYama(part, 'right').
- Right Yamanouchi words are exactly the words which are plactically congruent to the word ...3^part[3] 2^part[2] 1^part[1] or, equivalently, give the tableau [..., [3...3], [2...2], [1...1]] by Schensted algorithm. They are in bijection with their right insertion tableau.
- Left and right Yamanouchi words are exchanged when reading words from right to left.
- 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], 'right');

                          [w[1, 2, 1], w[2, 1, 1]]

> ListYama([3,2]);

            [w[1, 2, 1, 2, 1], w[1, 2, 1, 1, 2], w[1, 1, 2, 1, 2],

                      w[1, 1, 1, 2, 2], w[1, 1, 2, 2, 1]]
```

See Also: TYP[IsYama]

Function: 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]]);
```

$$\begin{bmatrix} 2 & 4 & 7 \\ & & \\ 1 & 3 & 6 \\ & & \\ & & 5 \end{bmatrix}$$

```
> Mat2CTab( CTab2Mat([[2,1], [4,3], [7,6,5]]) );

                    [[2, 1], [4, 3], [7, 6, 5]]
```

SEE ALSO: CTab2Mat TYP[CTableau] TYP[IsCTab]

FUNCTION: 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]]);
```

$$\begin{bmatrix} 6 & & \\ 4 & 5 & \\ 1 & 2 & 3 \end{bmatrix}$$

```
> Mat2Tab( Tab2Mat([[6], [4,5], [1,2,3]]) );

                        [[6], [4, 5], [1, 2, 3]]
```

See Also: TYP[IsTab] Tab2Mat TYP[Tableau]

Function: 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):
> lst:=Part2ListStdTab([3,1]);

      lst := [[[2], [1, 3, 4]], [[3], [1, 2, 4]], [[4], [1, 2, 3]]]

> map(Tab2Mat, lst);
```

$$\left[\begin{bmatrix} 2 & & \\ 1 & 3 & 4 \end{bmatrix}, \begin{bmatrix} 3 & & \\ 1 & 2 & 4 \end{bmatrix}, \begin{bmatrix} 4 & & \\ 1 & 2 & 3 \end{bmatrix}\right]$$

See Also: TYP[IsYama] ListStdTab ListYama Tab2Mat TYP[Tableau]

Function: PermOnCTab - action of a permutation on a product of columns

Calling Sequence:

```
PermOnCTab(perm, ctab)
TAB[PermOnCTab](perm, ctab)
```

Parameters:

`perm` = any list denoting a permutation
`ctab` = any list of lists denoting a product of columns

Synopsis:

- The `PermOnCTab` function plactically permutes columns in a product of columns `ctab` (not necessarily a contre-tableau, but obtained from a contretableau by permutation).
- A simple plactic column-transposition transforms a two-column contre-tableau into the two-column tableau which is obtained from it by Schensted algorithm, or conversely. This transformation is also given by the "jeu de taquin".
- When used with the extra argument `'nil'`, it nilplactically permutes columns, exchanging for instance `[[1], [2,1]] and [[2,1], [2]]`, while a plactic column-transposition exchanges `[[1], [2,1]] with [[2,1], [1]]`. Whenever the contretableau is not a reduced decomposition, it returns `[]`.
- Whenever there is a conflict between the function name `PermOnCTab` and another name used in the same session, use the long form `TAB['PermOnCTab']`.

Examples:

```
> with(TAB):
> PermOnCTab([2,1], [[5], [6,2], [7,4,3,1]]);

                    [[5, 2], [6], [7, 4, 3, 1]]

> PermOnCTab([2,1], [[3], [4,3]], 'nil');

                              [[4, 3], [4]]
```

See Also: PermOnTab

Function: PermOnTab - action of a permutation on a product of rows

Calling Sequence:

```
PermOnTab(perm, tab)
TAB[PermOnTab](perm, tab)
```

PARAMETERS:
`perm` = any list denoting a permutation
`tab` = any list of lists denoting a product of rows

SYNOPSIS:

- The `PermOnTab` function plactically permutes rows in a product of rows, say `tab` (not necessarily a tableau, but the image of a tableau under a row-permutation).
- A simple plactic row-transposition transforms a two-row contre-tableau into the two-row tableau which is obtained from it by Schensted algorithm, or conversely. This transformation is also given by the "jeu de taquin".
- When used with the extra argument `'nil'`, it nilplactically permutes rows, exchanging for instance `[[2], [1,2]] and [[1,2], [1]]`, while a plactic row-transposition exchanges `[[2], [1,2]] with [[2,2], [1]]`. Whenever the tableau is not a reduced decomposition, it returns `[]`.
- Whenever there is a conflict between the function name `PermOnTab` and another name used in the same session, use the long form `TAB['PermOnTab']`.

EXAMPLES:

```
> with(TAB):
> PermOnTab([2,1], [[1,3,4,6], [2,5]]);

                          [[3, 6], [1, 2, 4, 5]]

> PermOnTab([2,1], [[4], [3,4]], 'nil');

                              [[3, 4], [3]]
```

SEE ALSO: `PermOnCTab`

FUNCTION: `Schensted` - Schensted correspondence

CALLING SEQUENCE:
`Schensted(w)`
`TAB[Schensted](w)`

PARAMETERS:
`w` = any word

SYNOPSIS:

- The `Schensted` function gives the bijection between words and pairs of tableaux due to Schensted. The result is a list of two tableaux of the same shape, the second one being standard and called the (right) insertion tableau.
- When used with a second parameter `'nil'`, the first tableau in the result is a nil-plactic tableau.
- Whenever there is a conflict between the function name `Schensted` and another name used in the same session, use the long form `TAB['Schensted']`.

EXAMPLES:

```
> with(TAB):
> sch:=Schensted(w[4,1,2,4,2]);

            sch := [[[4, 4], [1, 2, 2]], [[2, 5], [1, 3, 4]]]

> map(Tab2Mat, sch);

                        [ 4  4    ]  [ 2  5    ]
                       [[         ], [         ]]
                        [ 1  2  2 ]  [ 1  3  4 ]

> Schensted(w[4,5,1,2,3,1], 'nil');

            [[[4], [2, 5], [1, 2, 3]], [[6], [3, 4], [1, 2, 5]]]
```

SEE ALSO: InvSchensted

FUNCTION: StdTab2SgaIdemp - build an idempotent from a standard tableau

CALLING SEQUENCE:

```
StdTab2SgaIdemp(t)
TAB[StdTab2SgaIdemp](t)
```

PARAMETERS:

t = any list of list(s) denoting a standard tableau

SYNOPSIS:

- The StdTab2SgaIdemp function computes an idempotent of the symmetric group algebra 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 t1, t2, ..., be the other tableaux obtained from t' by adding n. Then, up to a normalization factor:

  ```
  StdTab2SgaIdemp(t)=StdTab2SgaIdemp(t')*product(SgaJucis(n,n)-c(n,ti))
  ```

  where SgaJucis(n,n) is the last Jucis-Murphy element and c(n,ti) the distance to the diagonal of the letter n in the tableau ti.
- Whenever there is a conflict between the function name StdTab2SgaIdemp and another name used in the same session, use the long form TAB['StdTab2SgaIdemp'].

EXAMPLES:

```
> with(TAB):
> r1:=StdTab2SgaIdemp([[3], [1,2]]):
> r2:=StdTab2SgaIdemp([[3], [1,2,4]]);
```

```
      r2 := 5/48 A[4, 2, 3, 1] + 5/48 A[1, 4, 3, 2] + 1/24 A[1, 2, 4, 3]
         + 1/8 A[1, 2, 3, 4] + 5/48 A[2, 4, 3, 1] - 1/48 A[3, 2, 4, 1]
         + 5/48 A[4, 1, 3, 2] - 1/48 A[1, 3, 4, 2] - 1/48 A[4, 2, 1, 3]
         - 1/48 A[1, 4, 2, 3] - 1/48 A[2, 3, 4, 1] - 1/12 A[3, 4, 2, 1]
         - 1/48 A[2, 4, 1, 3] - 1/12 A[4, 3, 1, 2] - 1/16 A[3, 2, 1, 4]
         - 1/16 A[1, 3, 2, 4] - 1/12 A[4, 3, 2, 1] - 1/48 A[3, 1, 4, 2]
         - 1/12 A[3, 4, 1, 2] - 1/48 A[4, 1, 2, 3] - 1/16 A[2, 3, 1, 4]
         - 1/16 A[3, 1, 2, 4] + 1/8 A[2, 1, 3, 4] + 1/24 A[2, 1, 4, 3]
> (r1 &!* (SgaJucis(4,4) &!* (SgaJucis(4,4) + 2) / 8)) &!- r2;
                                      0
```

See Also: `SGA[SgaJucis]`

Function: `StdTab2Yama` - convert a standard tableau into a Yamanouchi word

Calling Sequence:

```
StdTab2Yama(t)
TAB[StdTab2Yama](t)
```

Parameters:

`t` = any list of list(s) denoting a standard tableau

Synopsis:

- The `StdTab2Yama` function converts any standard tableau into a Yamanouchi word. By default it returns a left Yamanouchi word.
- `StdTab2Yama(t) = StdTab2Yama(t, 'left')` but one can obtain a right Yamanouchi word by using `StdTab2Yama(t, 'right')`.
- Whenever there is a conflict between the function name `StdTab2Yama` and another name used in the same session, use the long form `TAB['StdTab2Yama']`.

Examples:

```
> with(TAB):
> StdTab2Yama([[4], [2, 5], [1, 3, 6]], 'right');
                              w[1, 2, 3, 1, 2, 1]
```

```
> StdTab2Yama([[4], [2, 5], [1, 3, 6]]);

                              w[1, 2, 1, 3, 2, 1]
```

See Also: TYP[IsYama] Yama2StdTab

Function: 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-dimensional 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]

Function: 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]`

FUNCTION: `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):
> tw:=[[7], [2,4,6], [1,3,5]]:
> Tab2Word(tw);
```

```
                    [7, 2, 4, 6, 1, 3, 5]
```

See Also: Word2Tab

Function: Word2CTab - convert a word into a contretableau

Calling Sequence:

```
Word2CTab(w)
TAB[Word2CTab](w)
```

Parameters:

`w` = any word

Synopsis:

- The Word2Tab function inserts a word `[x1, x2, ...]` into an empty contre-tableau from the right. It first inserts `x1`, then `x2`, and so on.
- The insertion algorithm is due to Schensted.
- When called with the `'nil'` option as second argument, the `Word2CTab` function nilplactically inserts the word into the empty contretableau. Whenever the word is not a reduced decomposition, it returns `[]`.
- 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(w[5,6,2,4,3,7,1]) );
```

$$\begin{bmatrix} 5 & 6 & 7 \\ & & \\ & 2 & 4 \\ & & \\ & & 3 \\ & & \\ & & 1 \end{bmatrix}$$

```
> Word2CTab(w[1,2,1,3,2,1], 'nil');

                        [[1], [2, 1], [3, 2, 1]]
```

See Also: BumpC CTab2Mat CTab2Word

Function: Word2Charge - returns the charge of a word

Calling Sequence:

```
Word2Charge(w)
TAB[Word2Charge](w)
```

PARAMETERS:
`w` = any word

SYNOPSIS:

- The `Word2Charge` function computes the charge of the word `w`.
- When called with the second argument `'vector'`, it returns the charge vector, the sum of the components of this vector being the charge. This option is defined only when the word contains at least as many 1's as 2's, as many 2's as 3's and so on.
- Whenever there is a conflict between the function name `Word2Charge` and another name used in the same session, use the long form `TAB['Word2Charge']`.

EXAMPLES:

```
> with(TAB):
> Word2Charge(w[2,3,1,1,2,3,4], 'vector');

                          [0, 1, 0, 0, 1, 1, 2]

> Word2Charge(w[2,3,1,1,2,3,4]);

                                  5
```

SEE ALSO: `Word2CoCharge`

FUNCTION: `Word2CoCharge` - returns the cocharge of a word

CALLING SEQUENCE:
`Word2CoCharge(w)`
`TAB[Word2CoCharge](w)`

PARAMETERS:
`w` = any word

SYNOPSIS:

- The `Word2CoCharge` function computes the cocharge of the word `w`.
- When called with the second argument `'vector'`, it returns the cocharge vector, the sum of the components of this vector being the cocharge. This option is defined only when the word contains at least as many 1's as 2's, as many 2's as 3's and so on.
- Whenever there is a conflict between the function name `Word2CoCharge` and another name used in the same session, use the long form `TAB['Word2CoCharge']`.

EXAMPLES:

```
> with(TAB):
> Word2CoCharge(w[3,2,2,1,4,1,1,3], 'vector');

                          [2, 1, 1, 0, 2, 0, 0, 1]

> Word2CoCharge(w[3,2,2,1,4,1,1,3]);

                                    7
```

SEE ALSO: Word2Charge

FUNCTION: Word2Key - left or right key

CALLING SEQUENCE:

```
Word2Key(w)
TAB[Word2Key](w)
```

PARAMETERS:

`w` = any word

SYNOPSIS:

- The `Word2Key` function returns the left or the right key of a given word. By default, it returns the left key.
- When called with the optionnal parameter `'right'`, it returns the right key corresponding to the word `w`. Using `'left'` as option, is equivalent to `Word2Key(w)`.
- The left key of a word `w` is the set of left columns in the orbit of:

```
PermOnCTab(perm, Word2CTab(w))
```

- The right key of a word `w` is the set of right columns in the orbit of:

```
PermOnCTab(perm, Word2CTab(w))
```

- When called an extra `'nil'` argument, the `Word2Key` returns the nilplactic corresponding key (either left or right). By default, it returns the left nilplactic key. Whenever the word is not a reduced decomposition, it returns `[]`.
- Whenever there is a conflict between the function name `Word2Key` and another name used in the same session, use the long form `TAB['Word2Key']`.

EXAMPLES:

```
> with(TAB):
> Word2Key(w[5,6,2,4,3,7,1]);

                        [[5], [5, 2], [5, 4, 2, 1]]

> Word2Key(w[5,6,2,4,3,7,1], 'right');

                         [[7, 4, 3, 1], [7, 3], [7]]

> Word2Key(w[3,2,1,3,2]);

                              [[3, 2], [3, 2, 1]]

> Word2Key(w[3,2,1,3,2], 'left', 'nil');

                              [[2, 1], [3, 2, 1]]
```

SEE ALSO: PermOnCTab PermOnTab

FUNCTION: Word2Tab - convert a word into a tableau

CALLING SEQUENCE:

```
Word2Tab(w)
TAB[Word2Tab](w)
```

PARAMETERS:

`w` = any word

SYNOPSIS:

- The `Word2Tab` function inserts a word `[x1, x2, ...]` into an empty tableau from the right. It first inserts `x1`, then `x2`, and so on.
- The insertion algorithm is due to Schensted.
- When called with the `'nil'` option as second argument, the `Word2Tab` function nilplactically inserts the word into the empty tableau. Whenever the word is not a reduced decomposition, it returns `[]`.
- 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(w[5,6,2,4,3,7,1]) );
```

```
[ 5       ]
[         ]
[ 4       ]
[         ]
[ 2  6    ]
[         ]
[ 1  3  7 ]
```

```
> Word2Tab(w[1,2,1,3,2,1], 'nil');
```

```
[[3], [2, 3], [1, 2, 3]]
```

See Also: Bump Tab2Mat Tab2Word

Function: 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. By default, yama is a left Yamanouchi word.
- Yama2StdTab(yama) = Yama2StdTab(yama, 'left') but one can also read the Yamanouchi word from right to left by using Yama2StdTab(yama, 'right').
- The letters in the Yamanouchi word indicate the heights of the letters in the standard tableau. The first (resp. last) entry in yama gives the the heigth of the letter 1 in the tableau corresponding to the left (resp. right) Yamanouchi word. Thus the bottom height can be choosen to be any arbitrary integer.
- 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(w[0,2,3,0,2,0], 'right');
```

```
[[4], [2, 5], [1, 3, 6]]
```

```
> Yama2StdTab(w[1,2,3,1,2,1]);
```

```
[[3], [2, 5], [1, 4, 6]]
```

See Also: TYP[IsYama] StdTab2Yama

5.18 Manipulating Schubert polynomials with SP

HELP FOR: A package for Schubert polynomials.

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	NewtonInterp	Sp2TableVar	SpScalarPol
SpSpecialize	TableX	TableXX	TableY
TableYY	ToX	ToXX	ToY
ToYY	Tox	X2Dim	x2X
x2XX			

- For help with a particular function do either ?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)
                ])
```

FUNCTION: `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_n without constant term, use `Flag(n)`. Equivalently, all Schubert polynomials indexed by permutations which cannot be restricted to $\mathfrak{S}_n$, *i.e.* which do not fix $n+1$, $n+2$, ..., are put to zero. The ring can be interpreted as the cohomology ring of the flag manifold of rank n over the field of complex numbers.
- In the case of double Schubert polynomials, the ideal is generated by identifying any symmetric polynomial in x to the same polynomial in y. All double Schubert polynomials indexed by permutations which cannot be restricted to $\mathfrak{S}_n$ are null.
- Use `Flag(-1)` to compute in the ring of polynomials, not modulo the ideal. This is the default case.
- When called without any argument, it returns the current value which is -1 when computing without truncation and a positive integer otherwise.
- 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):
> Flag(-1):
> x2X(x3);
```

```
                          - X[1, 3, 2] + X[1, 2, 4, 3]
```

```
> Flag(3):
> x2X(x3);
```

```
                                - X[1, 3, 2]
```

SEE ALSO: `x2X`

FUNCTION: `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 $f(x_1, x_2, ...)$ in terms of $\mathbb{X}$ Schubert polynomials. It uses the Newton interpolation formula involving the permutations in the symmetric group of degree `n`.
- Don't forget it is an interpolation...
- On the contrary, the function `x2XX` uses Monk formula instead of divided differences to decompose (exactly) a polynomial into the $\mathbb{X}$ basis.
- When called with one argument, the second argument is set to the value of the variable `_Sn` that fixes the degree of the symmetric group.
- The result is not expanded.
- Whenever there is a conflict between the function name `NewtonInterp` and another name used in the same session, use the long form `SP['NewtonInterp']`.

EXAMPLES:

```
> with(SP):
> Flag(-1):
> 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( Tox( NewtonInterp(2*x2 + 4*a^2*x1^2, 4) ) );

                                      2   2
                              2 x2 + 4 a  x1
```

SEE ALSO: `Flag` `x2XX`

FUNCTION: `Sp2TableVar` - variable set for the `SP` package

CALLING SEQUENCE:
`Sp2TableVar(expr)`
`SP[Sp2TableVar](expr)`

PARAMETERS:
`expr` = any expression

SYNOPSIS:

- The `Sp2TableVar` function returns a table whose entries describe the set of all Schubert polynomials and variables that appear in a given expression `expr`. The indices of the table are taken into the global variable `SPBases`.
- For all bases of type `'indexed'` (`SPBasesIndexed`), the entry consists of the set of all permutations that index the corresponding Schubert polynomials in `expr`.
- For all bases of type `'string'` (`SPBasesString`), the entry consists of the set of all integers that index the corresponding variable in `expr`.
- An entry equal to an empty set means that the element does not appear in `expr`.
- One can add a second argument to select some elements among those in `SPBases`. If the second argument is a set of names, then the restriction to this set of indices of the table is returned. In case the second argument is a single name, then the corresponding entry in the table is returned instead of the table.
- Whenever there is a conflict between the function name `Sp2TableVar` and another name used in the same session, use the long form `SP['Sp2TableVar']`.

EXAMPLES:

```
> with(SP):
> Sp2TableVar(x1^3*x4*y4 + q*y6*X[4,1,3,2]*X[2,3,1]);

                        table([
                            x = {1, 4}
                            YY = {}
                            XX = {}
                            X = {[4, 1, 3, 2], [2, 3, 1]}
                            Y = {}
                            y = {4, 6}
                        ])

> Sp2TableVar(x1^3*x4*y4 + q*y6*X[4,1,3,2]*X[2,3,1], {'X', 'x'});

                        table([
                            x = {1, 4}
                            X = {[4, 1, 3, 2], [2, 3, 1]}
                        ])

> Sp2TableVar(x1^3*x4*y4 + q*y6*X[4,1,3,2]*X[2,3,1], 'y');

                                    {4, 6}
```

SEE ALSO:

FUNCTION: `SpScalarPol` - scalar product on polynomials

CALLING SEQUENCE:

```
SpScalarPol(pol_1, pol_2, n)
SP[SpScalarPol](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 `SpScalarPol` function computes the scalar product <`pol_1`, `pol_2`> corresponding to the symmetric group of degree `n`.
- <`pol_1`, `pol_2`> = <`pol_1 * pol_2`, `1`>.
- <`pol`, `1`> 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 `SpScalarPol` and another name used in the same session, use the long form `SP['SpScalarPol']`.

EXAMPLES:

```
> with(SP):
> SpScalarPol(x1^6*x2, x2^2*x3, 3);
```

$$s_{[4,\,2,\,1]}$$

SEE ALSO:

FUNCTION: `SpSpecialize` - specialize a set of variables

CALLING SEQUENCE:

```
SpSpecialize(expr, gname, f, args)
SP[SpSpecialize](expr, gname, f, args)
```

PARAMETERS:

`expr` = any expression
`gname` = any string or indexed
`f` = any procedure
`args` = any (extra) parameters for `f`

SYNOPSIS:

- The `SpSpecialize` function is devoted to specialize a set of variables. This can be all variables generated by the generic name `gname` in such a way that the variable `gname.i` is sent onto `f(i, args)`. This function can also be used to specialize indexed variables named `gname[sequence]` onto `f([sequence], args)`.

- Whenever there is a conflict between the function name `SpSpecialize` and another name used in the same session, use the long form `SP['SpSpecialize']`.

EXAMPLES:

```
> with(SP):
> SpSpecialize(x1 + y3 + xx2 + x3^3, 'x', i -> y.i);

                                        3
                        y1 + y3 + xx2 + y3

> deg:=3:                    # you should use a map to specialize a table...
> map(SpSpecialize, TableX(deg), 'x', proc(i,d) q^(d-i) end, deg);

                        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(SpSpecialize, 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
                      ])

> SpSpecialize(X[3,1,2] + X[1,3,2], 'X', perm -> Y[op(Perm2Code(perm))]);

                       Y[2, 0, 0] + Y[0, 1, 0]
```

See Also: `SG[Perm2Code]` `TableX` `TableXX`

Function: `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.
- When called with the second argument `'Pe'`, it returns the table of all Schubert polynomials in $\mathfrak{S}_n$, expressed in the dual basis of the monomial basis, i.e. the basis of products of elementary symmetric functions on an alphabet flag. For instance, `Pe[1, 2, 0, 1, 0]` stands for the product `e1(x1,x2,x3,x4)*e2(x1,x2,x3)*e1(x1)`.
- 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
```

```
> TableX(3, 'Pe');
```

```
                table([
                  [3, 1, 2] = Pe[1, 1, 0] - Pe[2, 0, 0]
                  [3, 2, 1] = Pe[2, 1, 0]
                  [1, 2, 3] = Pe[0]
                  [2, 3, 1] = Pe[2, 0, 0]
                  [2, 1, 3] = Pe[1, 0]
```

```
    [1, 3, 2] = Pe[1, 0, 0]
  ])
```

See Also: TableXX

Function: 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`

Function: `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.
- When called with the second argument `'Pe'`, it returns the table of all Schubert polynomials in $\mathfrak{S}_n$, expressed in the dual basis of the monomial basis, i.e. the basis of products of elementary symmetric functions on an alphabet flag. For instance, `Pe[1, 2, 0, 1, 0]` stands for the product `e1(x1,x2,x3,x4)*e2(x1,x2,x3)*e1(x1)`.
- 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
```

```
> TableY(3, 'Pe');
```

```
                  table([
                    [0, 0, 0] = Pe[0, 0, 0]
                    [1, 0, 0] = Pe[0, 1, 0]
                    [2, 0, 0] = Pe[1, 1, 0] - Pe[2, 0, 0]
                    [0, 1, 0] = Pe[1, 0, 0]
                    [1, 1, 0] = Pe[2, 0, 0]
                    [2, 1, 0] = Pe[2, 1, 0]
                  ])
```

See Also: `TableYY`

Function: `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`

Function: `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 (`X[perm]`, `Y[code]`), double Schubert polynomials (`XX[perm]`, `YY[code]`), other terms being considered as coefficients.
- The expression `expr` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `expr` is solely expressed in terms of the known basis `b` (`x`, `X`, `Y`, `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`, but it simplifies Schubert polynomials indices.
- 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: `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):
> Flag(-1):
> ToX((1+q)^5*x3*x4):                          # expands the input
> ToX((1+q)^5*x3*x4,noexpand):                 # does not expand (1+q)^5
> ToX((1+q)^5*x3*x4,collect):                  # collects the result
> ToX((1+q)^5*x3*x4,noexpand,'x'):             # the most efficient
> ToX((1+q)^5*x3*x4,'x',collect):              # specifies a basis
> ToX((1+q)^5*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: `Flag` `TYP[Schubert]` `Tox`

Function: `ToXX` - convert any expression to the $\mathbb{X}$ Schubert basis

Calling Sequence:

```
ToXX(expr)
ToXX(expr, b)
SP[ToXX](expr)
SP[ToXX](expr, b)
```

Parameters:

`expr` = any expression
`b` = any name of a known basis

Synopsis:

- The `ToXX` function converts any expression `expr` to the $\mathbb{X}$ Schubert basis. `expr` may involve some x_i's, simple Schubert polynomials (`X[perm]`, `Y[code]`), double Schubert polynomials (`XX[perm]`, `YY[code]`, the second alphabet being the y_i's), other terms being considered as coefficients.
- The expression `expr` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `expr` is solely expressed in terms of the known basis `b` (`x`, `X`, `Y`, `XX`, `YY` and even `y` that is seen as a basis in the package).
- The call `ToXX(expr, 'XX')` does not affect the argument `expr`, but it simplifies Schubert polynomials indices.

- 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: ToXX(expr, b, 'collect'). For instance, ToXX(expr, 'XX', 'collect') may be used to collect the argument expr.
- Whenever there is a conflict between the function name ToXX and another name used in the same session, use the long form SP['ToXX'].

Examples:

```
> with(SP):
> Flag(-1):
> ToXX((1+q)^5*x3*x4):                          # expands the input
> ToXX((1+q)^5*x3*x4,noexpand):                 # does not expand (1+q)^5
> ToXX((1+q)^5*x3*x4,collect):                  # collects the result
> ToXX((1+q)^5*x3*x4,noexpand,'x'):             # the most efficient
> ToXX((1+q)^5*x3*x4,'x',collect):              # specifies a basis
> ToXX((1+q)^5*x3*x4,noexpand,'x',collect):
> ToXX(q^2*x3*X[3,1,2], collect);

  2                   2                               2   2
 q  XX[3, 1, 4, 2] + q  (y2 + y1) XX[2, 1, 4, 3] + q  y1  XX[1, 2, 4, 3]

     2                2                             2   2
   - q  XX[3, 2, 1] - q  (y2 + y1) XX[2, 3, 1] - q  y1  XX[1, 3, 2]

     2                    2                             2   2
   - q  y1 XX[3, 1, 2] + q  y3 (y2 + y1) XX[2, 1] + q  y1  y3 XX[1]
```

See Also: Flag ToYY

Function: 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 **Y** Schubert basis. expr may involve some x_i's, simple Schubert polynomials (X[perm], Y[code]), double Schubert polynomials (XX[perm], YY[code]), other terms being considered as coefficients.

- The expression `expr` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `expr` is solely expressed in terms of the known basis `b` (`x`, `X`, `Y`, `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`, but it simplifies Schubert polynomials indices.
- 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: `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):
> Flag(-1):
> ToY((1+q)^5*x3*x4):                          # expands the input
> ToY((1+q)^5*x3*x4,noexpand):                 # does not expand (1+q)^5
> ToY((1+q)^5*x3*x4,collect):                  # collects the result
> ToY((1+q)^5*x3*x4,noexpand,'x'):             # the most efficient
> ToY((1+q)^5*x3*x4,'x',collect):              # specifies a basis
> ToY((1+q)^5*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: `Flag` `TYP[Schubert]` `Tox`

FUNCTION: `ToYY` - convert any expression to the $\mathbb{Y}$ Schubert basis

CALLING SEQUENCE:

```
ToYY(expr)
ToYY(expr, b)
SP[ToYY](expr)
SP[ToYY](expr, b)
```

PARAMETERS:

`expr` = any expression
`b` = any name of a known basis

SYNOPSIS:

- The `ToYY` function converts any expression `expr` to the $\mathbb{Y}$ Schubert basis. `expr` may involve some x_i's, simple Schubert polynomials (`X[perm]`, `Y[code]`), double Schubert polynomials (`XX[perm]`, `YY[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 `ToYY(expr, 'YY')` does not affect the argument `expr`, but it simplifies Schubert polynomials indices.
- 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: `ToYY(expr, b, 'collect')`. For instance, `ToYY(expr, 'YY', 'collect')` may be used to collect the argument `expr`.
- Whenever there is a conflict between the function name `ToYY` and another name used in the same session, use the long form `SP['ToYY']`.

EXAMPLES:

```
> with(SP):
> Flag(-1):
> ToYY((1+q)^5*x3*x4):                          # expands the input
> ToYY((1+q)^5*x3*x4,noexpand):                 # does not expand (1+q)^5
> ToYY((1+q)^5*x3*x4,collect):                  # collects the result
> ToYY((1+q)^5*x3*x4,noexpand,'x'):             # the most efficient
> ToYY((1+q)^5*x3*x4,'x',collect):              # specifies a basis
> ToYY((1+q)^5*x3*x4,noexpand,'x',collect):
> ToYY(q^2*x3*Y[2,0], collect);

     2                   2                     2       2
  - q  y1 YY[2, 0, 0] + q  YY[2, 0, 1, 0] + (q  y1 + q  y2) YY[1, 0, 1, 0]

        2       2                     2   2
   + (- q  y2 - q  y1) YY[1, 1, 0] + q  y1  YY[0, 0, 1, 0]

      2          2                     2   2                2      2
   + (q  y3 y2 + q  y3 y1) YY[1, 0] - q  y1  YY[0, 1, 0] + q  y3 y1  YY[0]

     2
   - q  YY[2, 1, 0]
```

See Also: Flag ToXX

Function: 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 (X[perm], Y[code]), double Schubert polynomials (XX[perm], YY[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, but it simplifies Schubert polynomials indices.
- 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: 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):
> Flag(-1):
> Tox((1+q)^5*X[3,1,2]):                          # expands the input
> Tox((1+q)^5*X[3,1,2],noexpand):                 # does not expand (1+q)^5
> Tox((1+q)^5*X[3,1,2],collect):                  # collects the result
> Tox((1+q)^5*X[3,1,2],noexpand,'X'):             # the most efficient
> Tox((1+q)^5*X[3,1,2],'X',collect):              # specifies a basis
> Tox((1+q)^5*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`

Function: `X2Dim` - specialization in **X** Schubert basis

Calling Sequence:

```
X2Dim(schub)
SP[X2Dim](schub)
```

Parameters:

`schub` = any expression in the **X** Schubert basis

Synopsis:

- The `X2Dim` function specializes in `x.i=1` all Schubert polynomials in schub, expressed in **X** Schubert basis.
- When used with the second argument `'q'`, it specializes in `x.i=q^(i-1)` all Schubert polynomials.
- Whenever there is a conflict between the function name `X2Dim` and another name used in the same session, use the long form `SP['X2Dim']`.

Examples:

```
> with(SP):
> X2Dim(a*X[3,1,2] - b*X[2,3,1,5,4] + c^3);

                                      3
                              a - 4 b + c

> X2Dim(X[2,3,1,5,4] - a*X[3,1,4,2], 'q');

                        4    3    2          2
                       q  + q  + q  + q - a (q  + q)
```

See Also:

FUNCTION: 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. It uses a different algorithm than ToX (taking the leading monomials successively).
- 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):
> Flag(-1):
> x2X((1+q)^5*x1*x2^2):                          # expands the input
> x2X((1+q)^5*x1*x2^2,noexpand):                 # does not expand (1+q)^5
> x2X((1+q)^5*x1*x2^2,collect):                  # collects the result
> x2X((1+q)^5*x1*x2^2,noexpand,collect):
> x2X(2*x1^3 + 4*q^2*x1^2);
```

$$2\,X[4,\,1,\,2,\,3] + 4\,q^2\,X[3,\,1,\,2]$$

SEE ALSO: Flag

FUNCTION: 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. It uses a different algorithm than `ToXX` (taking the leading monomials successively).
- 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):
> Flag(-1):
> x2XX((1+q)^5*x1*x2^2):                          # expands the input
> x2XX((1+q)^5*x1*x2^2,noexpand):                 # does not expand (1+q)^5
> x2XX((1+q)^5*x1*x2^2,collect):                  # collects the result
> x2XX((1+q)^5*x1*x2^2,noexpand,collect):
> r := x2XX(2*x1^3 + 4*q^2*x1^2, collect);

                3      2   2
     r := (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]

> Tox(r);

                                     3      2   2
                                  2 x1  + 4 q  x1
```

SEE ALSO: `Flag` `NewtonInterp`

5.19 Computations in the ring $\mathbb{Z}[x_1, \ldots, x_n]$ with FM

HELP FOR: A package to compute in the polynomial ring in several variables.

CALLING SEQUENCE:

```
FM[<function>](args)
<function>(args)
```

SYNOPSIS:

- This package provides functions to compute in the polynomial ring in several, say n, variables, seen as a free module of rank $n!$ over the ring of symmetric polynomials in n variables. We choose as basis of this module Schubert polynomials indexed by permutations in the symmetric group of degree n.
- By default, one computes in the polynomial ring in 4 variables x_1, x_2, x_3 and x_4. In this case, x_5, x_6, etc. are considered as scalars.
- Whenever there is a conflict between a function name in FM and another name used in the same session, use the long form FM[<function>].
- The available functions are:

FM_n	ToXfix	ToXfixScal	x2Xfix
x2XfixScal	X2Xfix	Xfix2x	Xfix2X

- For help with a particular function do either ?FM[<function>] or ?FM,<function> where <function> is one from the above list.

EXAMPLES:

```
> with(FM):
> FM_n(3);
```

$$3$$

```
> x2XfixScal(q*x2*x3);
```

$$q\ s[1, 1]\ X[1, 2, 3] - q\ s[1]\ X[2, 1, 3] + q\ X[3, 1, 2]$$

FUNCTION: FM_n - fix the number of variables

CALLING SEQUENCE:

```
FM_n(n)
FM_n(n, 'load')
FM[FM_n](n)
FM[FM_n](n, 'load')
```

PARAMETERS:
`n` = any positive integer

SYNOPSIS:

- The `FM_n` function is used to fix the number of variables. This value is stored in the `_FMn` global variable. The ring of polynomials in $x_1, ..., x_n$, where `n`=`_FMn`, will be considered as a free-module over the ring of symmetric polynomials in $x_1, ..., x_n$, with basis the Schubert polynomials indexed by permutations in the symmetric group $\mathfrak{S}_n$.
- When called without any argument, it returns the current value of `_FMn`.
- When called with a second paramater `'load'`, it loads all pre-computed tables of available data for the given `n`. If one table is not available, it computes it so that it can take some time.
- Whenever there is a conflict between the function name `FM_n` and another name used in the same session, use the long form `FM['FM_n']`.

EXAMPLES:

```
> with(FM):
> FM_n();
```

```
                                    4
```

```
> FM_n(3, 'load');
```

```
                                    3
```

SEE ALSO: `ACE[AceBuildData]` `ACE[AceLoadData]`

FUNCTION: `ToXfix` - convert any expression to the **X** Schubert basis of the ring of polynomials as a free module over **Sym**

CALLING SEQUENCE:
```
ToXfix(expr)
ToXfix(expr, b)
FM[ToXfix](expr)
FM[ToXfix](expr, b)
```

PARAMETERS:
`expr` = any expression
`b` = any name of a known basis

SYNOPSIS:

- The `ToXfix` function converts any expression `expr` to the **X** Schubert basis of the free module over symmetric polynomials. `expr` may involve some x_i's, simple Schubert polynomials (`X[perm]`, `Y[code]`), other terms being considered as coefficients.

- The expression `expr` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `expr` is solely expressed in terms of the known basis `b` (`x`, `X`, `Y`).
- The call `ToXfix(expr, 'X')` does not affect the argument `expr`.
- One may add `'noexpand'` just after the argument `expr` to choose not to expand the expression `expr` before treating it.
- One may collect the result by adding a third argument: `ToXfix(expr, b, 'collect')`. For instance, the following instruction `ToXfix(expr, 'X', 'collect')` may be used to collect the argument `expr`.
- Whenever there is a conflict between the function name `ToXfix` and another name used in the same session, use the long form `FM['ToXfix']`.

EXAMPLES:

```
> with(FM):
> FM_n(4);
```

$$4$$

```
> ToXfix((1+q)^2*x3*x4):                          # expands the input
> ToXfix((1+q)^2*x3*x4,noexpand):                 # does not expand (1+q)^2
> ToXfix((1+q)^2*x3*x4,collect):                  # collects the result
> ToXfix((1+q)^2*x3*x4,noexpand,'x'):             # the most efficient
> ToXfix((1+q)^2*x3*x4,'x',collect):              # specifies a basis
> ToXfix((1+q)^2*x3*x4,noexpand,'x',collect):
> ToXfix(q^2*x3*X[3,1,4,2]*Y[0,1], collect);
```

$$
\begin{aligned}
&- q^2\, X[1, 3, 2, 4] + X[1, 2, 3, 4] + q^2\, s[1]\, X[3, 4, 1, 2] \\
&+ q^2\, s[1]\, X[3, 2, 4, 1] + q^2\, s[1]\, X[4, 1, 3, 2] + q^2\, X[3, 4, 2, 1] \\
&+ q^2\, X[4, 2, 3, 1] - q^2\, s[1, 1, 1]\, X[2, 3, 1, 4] \\
&- q^2\, s[1, 1, 1]\, X[3, 1, 2, 4] + q^2\, s[1, 1, 1, 1]\, X[1, 3, 2, 4]
\end{aligned}
$$

SEE ALSO: `ToXfixScal`

FUNCTION: `ToXfixScal` - convert any expression to the **X** Schubert basis of the ring of polynomials as a free module over **Sym**

CALLING SEQUENCE:

`ToXfixScal(expr)`

```
ToXfixScal(expr, b)
FM[ToXfixScal](expr)
FM[ToXfixScal](expr, b)
```

PARAMETERS:
`expr` = any expression
`b` = any name of a known basis

SYNOPSIS:

- The `ToXfixScal` function converts any expression expr to the **X** Schubert basis of the ring of polynomials as a free module over symmetric polynomials, by using scalar products instead of Monk formula when possible. This function is useful for `_FMn` ≤ 5 especially when pre-computed data tables are loaded. `expr` may involve some x_i's, simple Schubert polynomials (`X[perm]`, `Y[code]`), other terms being considered as coefficients.
- The expression `expr` is expanded and the result is not collected.
- One may specify by a second argument, say `b`, that `expr` is solely expressed in terms of the known basis `b` (`x`, `X`, `Y`).
- The call `ToXfixScal(expr, 'X')` does not affect the argument `expr`.
- One may add `'noexpand'` just after the argument `expr` to choose not to expand the expression `expr` before treating it.
- One may collect the result by adding a third argument: `ToXfixScal(expr, b, 'collect')`. For instance, the next instruction `ToXfixScal(expr, 'X', 'collect')` may be used to collect its argument.
- Whenever there is a conflict between the function name `ToXfixScal` and another name used in the same session, use the long form `FM['ToXfixScal']`.

EXAMPLES:

```
> with(FM):
> FM_n(4);
```

$$4$$

```
> ToXfixScal((1+q)^2*x3*x4):                    # expands the input
> ToXfixScal((1+q)^2*x3*x4,noexpand):           # does not expand (1+q)^2
> ToXfixScal((1+q)^2*x3*x4,collect):            # collects the result
> ToXfixScal((1+q)^2*x3*x4,noexpand,'x'):       # the most efficient
> ToXfixScal((1+q)^2*x3*x4,'x',collect):        # specifies a basis
> ToXfixScal((1+q)^2*x3*x4,noexpand,'x',collect):
> ToXfixScal(q^2*x3*X[3,1,4,2]*Y[0,1], collect);
```

$$- q^2\ X[1, 3, 2, 4]\ X[1, 2, 3, 4] + q^2\ s[1]\ X[3, 4, 1, 2]$$

$$+ q^2\ s[1]\ X[3, 2, 4, 1] + q^2\ s[1]\ X[4, 1, 3, 2] + q^2\ X[3, 4, 2, 1]$$

```
   2                   2
+ q  X[4, 2, 3, 1] - q  s[1, 1, 1] X[2, 3, 1, 4]

   2                              2
- q  s[1, 1, 1] X[3, 1, 2, 4] + q  s[1, 1, 1, 1] X[1, 3, 2, 4]
```

See Also: ToXfix

Function: X2Xfix - from **X** Schubert basis to **X** Schubert basis of the ring of polynomials as a free module over **Sym**

Calling Sequence:

```
X2Xfix(schub)
FM[X2Xfix](schub)
```

Parameters:

schub = any expression in **X** Schubert basis

Synopsis:

- The X2Xfix function converts an expression from **X** Schubert basis to the **X** Schubert basis of the ring of polynomials as a free module over symmetric polynomials.
- All Schubert polynomials are indexed by permutations in the symmetric group of degree _FMn. Coefficients are symmetric polynomials expressed in the Schur basis.
- The result is not expanded.
- One may add 'noexpand' just after the argument schub to choose not to expand the expression schub before treating it.
- One may add 'collect' just after the argument schub or just after the argument 'noexpand' to collect the result.
- Whenever there is a conflict between the function name X2Xfix and another name used in the same session, use the long form FM['X2Xfix'].

Examples:

```
> with(FM):
> FM_n(4);
```

4

```
> X2Xfix((1+q)^5*X[3,5,2,4,1]):                  # expands the input
> X2Xfix((1+q)^5*X[3,5,2,4,1],noexpand):         # does not expand (1+q)^5
> X2Xfix((1+q)^5*X[3,5,2,4,1],collect):          # collects the result
> X2Xfix((1+q)^5*X[3,5,2,4,1],noexpand,collect):
> X2Xfix(X[3,2,1] + q*X[3,2,1] + z*X[4,5,1,3,2], collect);
```

```
(1 + q) X[3, 2, 1, 4] - z s[1, 1, 1, 1] X[2, 4, 1, 3]

   + z s[1, 1, 1] X[3, 4, 1, 2]
```

See Also: `Xfix2X`

Function: `Xfix2X` - from **X** Schubert basis with symmetric functions as coefficients to **X** Schubert basis

Calling Sequence:

```
Xfix2X(schub)
FM[Xfix2X](schub)
```

Parameters:

`schub` = any expression in **X** Schubert basis

Synopsis:

- The `Xfix2X` function converts an expression from **X** Schubert basis of the ring of polynomials as a free module over symmetric polynomials to the **X** Schubert basis. Symmetric polynomials are also converted.
- The result is not expanded.
- One may add `'noexpand'` just after the argument `schub` to choose not to expand the expression `schub` before treating it.
- One may add `'collect'` just after the argument `schub` or just after the argument `'noexpand'` to collect the result.
- We recommend to use the `collect` option to have the result simplified.
- Whenever there is a conflict between the function name `Xfix2X` and another name used in the same session, use the long form `FM['Xfix2X']`.

Examples:

```
> with(FM):
> FM_n(3);
```

$$3$$

```
> Xfix2X(h2*X[1,3,2]);
```

$$X[1, 3, 5, 2, 4] + X[1, 5, 2, 3, 4]$$

See Also: X2Xfix

Function: Xfix2x - from **X** Schubert basis with symmetric functions as coefficients to the basis of monomials

Calling Sequence:

```
Xfix2x(schub)
FM[Xfix2x](schub)
```

Parameters:

`schub` = any expression in **X** Schubert basis

Synopsis:

- The `Xfix2x` function converts an expression from **X** Schubert basis of the ring of polynomials as a free module over symmetric polynomials to the basis of monomials. Symmetric functions are also converted.
- The result is not expanded.
- Whenever there is a conflict between the function name `Xfix2x` and another name used in the same session, use the long form `FM['Xfix2x']`.

Examples:

```
> with(FM):
> FM_n(3);
```

$$3$$

```
> Xfix2x(h2*X[1,3,2] + a*X[3,1,2]);
```

$$(x3^2 + x3\,x2 + x3\,x1 + x2^2 + x2\,x1 + x1^2)\,(x2 + x1) + a\,x1^2$$

See Also: x2Xfix

Function: x2Xfix - from the basis of monomials to **X** Schubert basis of the ring of polynomials as a free module over **Sym**

Calling Sequence:

```
x2Xfix(pol)
FM[x2Xfix](pol)
```

Parameters:

`pol` = any expression in the basis of monomials

SYNOPSIS:

- The `x2Xfix` function converts any expression from the basis of monomials to the **X** Schubert basis of the ring of polynomials as a free module over symmetric polynomials.
- All Schubert polynomials are indexed by permutations in the symmetric group of degree `_FMn`. Coefficients are symmetric polynomials expressed in the Schur basis.
- The result is not expanded.
- One may add `'noexpand'` just after the argument `pol` to choose not to expand the expression `pol` before treating it.
- One may add `'collect'` just after the argument `pol` or just after the argument `'noexpand'` to collect the result.
- Whenever there is a conflict between the function name `x2Xfix` and another name used in the same session, use the long form `FM['x2Xfix']`.

EXAMPLES:

```
> with(FM):
> FM_n(4);
```

$$4$$

```
> x2Xfix((1+q)^5*x1*x3^3):                        # expands the input
> x2Xfix((1+q)^5*x1*x3^3,noexpand):               # does not expand (1+q)^5
> x2Xfix((1+q)^5*x1*x3^3,collect):                # collects the result
> x2Xfix((1+q)^5*x1*x3^3,noexpand,collect):
> x2Xfix(q*x2^2*x4 - x3^3*x4, collect);
```

$$\begin{aligned}
&(s[2,\,2] - q\,s[1,\,1,\,1])\,X[1,\,2,\,3,\,4] - X[1,\,2,\,4,\,3]\,s[2,\,1] \\
&+ (-q\,s[1] - s[1,\,1])\,X[2,\,3,\,1,\,4] + (s[2] + s[1,\,1])\,X[1,\,3,\,4,\,2] \\
&+ s[1,\,1]\,X[1,\,4,\,2,\,3] + (-s[1] + q)\,X[2,\,3,\,4,\,1] \\
&- q\,s[1]\,X[3,\,1,\,2,\,4] + X[3,\,4,\,1,\,2] + (-q - s[1])\,X[1,\,4,\,3,\,2] \\
&+ q\,X[3,\,1,\,4,\,2] + X[2,\,4,\,3,\,1] + q\,s[1,\,1]\,X[1,\,3,\,2,\,4] \\
&+ q\,X[4,\,1,\,2,\,3] + q\,X[2,\,4,\,1,\,3]
\end{aligned}$$

SEE ALSO: `FM_n` `x2XfixScal` `Xfix2x`

FUNCTION: `x2XfixScal` - from the basis of monomials to **X** Schubert basis of the ring of polynomials as a free module over **Sym**

CALLING SEQUENCE:

`x2XfixScal(pol)`

`FM[x2XfixScal](pol)`

PARAMETERS:

`pol` = any expression in the basis of monomials

SYNOPSIS:

- The `x2XfixScal` function converts any expression from the basis of monomials to the **X** Schubert basis of the ring of polynomials as a free module over symmetric polynomials by computing scalar products with the adjoint basis of the **X** Schubert basis.
- All Schubert polynomials are indexed by permutations in the symmetric group of degree `_FMn`. Coefficients are symmetric polynomials expressed in the Schur basis.
- This function is quite efficient for small values of `_FM`: for instance, `_FM` ≤ 5.
- The result is not expanded but collected.
- Whenever there is a conflict between the function name `x2XfixScal` and another name used in the same session, use the long form `FM['x2XfixScal']`.

EXAMPLES:

```
> with(FM):
> FM_n(4);
```

$$4$$

```
> x2XfixScal(q*x2^2*x4 - x3^3*x4);

   (s[2, 2] - q s[1, 1, 1]) X[1, 2, 3, 4] - X[1, 2, 4, 3] s[2, 1]

 + (- q s[1] - s[1, 1]) X[2, 3, 1, 4] + (s[2] + s[1, 1]) X[1, 3, 4, 2]

 + s[1, 1] X[1, 4, 2, 3] + (- s[1] + q) X[2, 3, 4, 1]

 - q s[1] X[3, 1, 2, 4] + X[3, 4, 1, 2] + (- q - s[1]) X[1, 4, 3, 2]

 + q X[3, 1, 4, 2] + X[2, 4, 3, 1] + q s[1, 1] X[1, 3, 2, 4]

 + q X[4, 1, 2, 3] + q X[2, 4, 1, 3]
```

SEE ALSO: `FM_n` `x2Xfix` `Xfix2x`

6

About ACE 3.0

ACE is an environment for computations in algebraic combinatorics. It has been developped by several persons and is distributed for several systems. This chapter clusters some informations about ACE. The next table gives for each author, the corresponding package(s):

Author(s)	**Package(s)**	**Email**
Sébastien Veigneau	ACE SG, BN, SGA, NCA, IDCA, HEKA, BNA, SYMF, CLG, TAB, SP, FM, FREE, (CG) ERR, TYP, TEX	Sebastien.Veigneau@univ-mlv.fr
Vincent Prosper	PART, COMP, SFA	Vincent.Prosper@univ-mlv.fr
Bun-Chan-Vorac Ung	NCSF	Ung@univ-mlv.fr
Renaud Eppstein	CG	Renaud.Eppstein@univ-mlv.fr

Packages in ACE *3.0.*

Bibliography

Le nombre de livres écrits sur l'électron en cinquante ans
est sans doute plus grand que le nombre de livres écrits sur la
lune en cinq cents ans. Exister par le livre, c'est déjà une existence,
une existence si humaine, si solidement humaine!

L'activité rationaliste de la physique contemporaine, Bachelard.

[1] G.Duchamp, D.Krob, A.Lascoux, B.Leclerc, T.Scharf and J.-Y.Thibon, *Euler-Poincare characteristic and polynomial representations of Iwahori-Hecke algebras*, Publications of the Research Institute for Mathematical Sciences (Kyoto), **31**, (2), 179–201, 1995.

[2] A.Kerber, *Algebraic combinatorics via finite group actions*, Mannheim:BI-Wissenschaftsverlag, 1991.

[3] A.Kerber, A.Kohnert and A.Lascoux, *SYMMETRICA, an object oriented computer algebra system for the symmetric group*, J. Symbolic Comp. **14** (1992), 195–203.

[4] D.Krob, A.Lascoux, B.Leclerc, J.-Y.Thibon, B.C.V.Ung and S.Veigneau, *Algebraic Combinatorics with MAPLE and ACE*, Maple Technical Newsletter **4** (1997), 43–50.

[5] A.Lascoux and V.Prosper, *$\mu-EC$, a MuPAD Environment for Combinatorics*, preprint, Université de Marne-la-Vallée, 1998.

[6] J.-Y.Thibon and S.Veigneau, *Using ACE, an Algebraic Combinatorics Environment for MAPLE*, Proceedings of the 8-*th* Conference "Formal Power Series and Algebraic Combinatorics", University of Minnesota, 1996.

[7] B.C.V.Ung and S.Veigneau, *ACE, un environnement de calcul en Combinatoire Algébrique*, Proceedings of the 7-*th* Conference "Formal Power Series and Algebraic Combinatorics", Université de Marne-la-Vallée 1995, 557–562.

[8] S.Veigneau, *ACE, an Algebraic Combinatorics Environment for the computer algebra system MAPLE*, *User's Reference Manual*, *Version 1.0*, Publ. IGM **95-14**, Université de Marne-la-Vallée, 1995.

[9] S.Veigneau, *ACE, an Algebraic Combinatorics Environment for the computer algebra system MAPLE*, *User's Reference Manual*, *Version 2.0*, Publ. IGM **97-05**, Université de Marne-la-Vallée, 1997.

[10] S.Veigneau, *Calcul Symbolique et Calcul Distribué en Combinatoire Algébrique*, thesis, Publ. IGM **96-39**, 227 pages, December 10, 1996.

Characters of classical groups

[11] G.R.E.Black, R.C.King and B.G.Wybourne, *Kronecker products for compact semisimple Lie groups*, J. Phys. A: Math. Gen., 1983.

[12] H.Georgi, *Lie algebras in particle physics*, Benjamin, Reading Mass., 1982.

[13] R.C.King, *S-functions and characters of Lie algebras and Lie groups*, Topics in Algebra, Banach Center Publication, **26**, PWN, Warsaw, 1990.

[14] R.C.King, *Modification rules and products of irreducible representations of the unitary, orthogonal and symplectic groups*, J. Math. Phys. **12**, 1971, 1588–1598.

[15] K.Koike and I.Terada, *Young-diagrammatic methods for the representation theory of the classical group of type B_n, C_n, D_n*, J. Algebra **107**, 1987, 466–511.

[16] C.Kostka, *Über den Zusammenhang zwischen einingen Formen von symmetrischen Funktionen*, Crelle's Journal **93**, 89–123.

[17] D.E.Littlewood, *On certain symmetric functions*, J. London. Math. Soc. **11**, 1961, 485–498.

[18] D.E.Littlewood, *Products and plethysms of characters with orthogonal, symplectic and symmetric groups*, J. Can. Math. **10**, 1958, 17–32.

[19] D.E.Littlewood and A.R.Richardson, *Group characters and algebra*, Phil. Trans. R. Soc. **233**, 99–141.

[20] G.Lusztig, *Singularities, character formulas and a q-analog of weight multiplicities*, Analyse et topologie des espaces singuliers (II-III), Astérisque **101–102**, 1983.

[21] R.V.Moody and J.Patera, *Fast recursion formula for weight multiplicities*, Bull. American Math. Soc. **7**, 237–242.

[22] D.H.Sattinger and O.L.Weaver, *Lie groups and algebras with applications to physics, geometry and mechanics*, Springer-Verlag **61**, New York, 1986.

[23] T.Scharf and J.-Y.Thibon, *A Hopf-algebra approach to inner plethysm*, Advances in Math. **104**, 1994.

[24] J.-P.Serre, *Complex semisimple Lie algebras*, Springer-Verlag, New York, 1987.

[25] D.Stanton, *Invariant theory and tableaux*, Springer-Verlag, New York, 1986.

[26] H.Weyl, *Classical groups*, Princeton University Press, Princeton, NJ, 1939.

[27] B.G.Wybourne, *Symmetry principles and atomic spectroscopy*, Wiley, 1969.

[28] B.G.Wybourne, *SCHUR an interactive programme for calculating properties of Lie groups.*

Fock space representation of $U_q(\widehat{sl}_n)$

[29] S.Ariki, *On the decomposition numbers of the Hecke algebra of $G(m,1,n)$*, preprint 1996.

[30] I.Grojnowski, *Affine Hecke algebras (and affine quantum GL_n) at roots of unity*, Internat. Res. Math. Notices **5** (1994), 215–217.

[31] T.Hayashi, *q-analogues of Clifford and Weyl algebras – spinor and oscillator representations of quantum enveloping algebras*, Commun. Math. Phys., **127** (1990), 129–144.

[32] G.James, *The decomposition matrices of $GL_n(q)$ for $n \leq 10$*, Proc. London Math. Soc., **60** (1990), 225–265.

[33] G.James, A.Kerber, *The representation theory of the symmetric group*, Addison-Wesley, 1981.

[34] M.Kashiwara, *On crystal bases of the q-analogue of universal enveloping algebras*, Duke Math. J. **63** (1991), 465–516.

[35] M.Kashiwara, *Crystal bases of modified quantized enveloping algebra*, Duke Math. J. **73** (1994), 383–413.

[36] M.Kashiwara, T.Miwa, E.Stern, *Decomposition of q-deformed Fock spaces*, preprint RIMS-1035 (1995), Selecta Mathematica (New series), (1996).

[37] A.S.Kleshchev, *Branching rules for the modular representations of symmetric groups III; some corollaries and a problem of Mullineux*, J. London Math. Soc., **2** (1995).

[38] A.Lascoux, B.Leclerc, J.-Y. Thibon, *Hecke algebras at roots of unity and crystal bases of quantum affine algebras*, Commun. Math. Phys. (to appear).

[39] A.Lascoux, B.Leclerc, J.-Y.Thibon, *Ribbon tableaux, Hall-Littlewood functions, quantum affine algebras and unipotent varieties*, preprint, 1995.

[40] B.Leclerc, J.-Y.Thibon, *Canonical bases of q-deformed Fock spaces*, preprint q-alg/9602025, Internat. Res. Math. Notices, (to appear).

[41] K.C.Misra, T.Miwa, *Crystal base of the basic representation of $U_q(\widehat{sl}_n)$*, Commun. Math. Phys., **134** (1990), 79–88.

Noncommutative symmetric functions

[42] F.Bergeron and D.Krob, *Acyclic complexes related to noncommutative symmetric functions*, preprint, 1995, to appear in J. Alg. Comb.

[43] R.W.Carter, *Representation theory of the 0-Hecke algebra*, J. Algebra, **15**, (1986), 89–103.

[44] G.Duchamp, A.Klyachko, D.Krob and J.-Y.Thibon, *Noncommutative symmetric functions III: Deformations of Cauchy and convolution algebras*, Publ. IGM **96-1**, Université de Marne-la-Vallée, 1996.

[45] G.Duchamp, D.Krob, B.Leclerc and J.-Y.Thibon, *Déformations de projecteurs de Lie*, C.R. Acad. Sci. Paris **319** (1994), 909–914.

[46] G.Duchamp, D.Krob, B.Leclerc and J.-Y.Thibon, *Fonctions quasi-symétriques, fonctions symétriques non commutatives et algèbre de Hecke à q=0*, C.R. Acad. Sci. Paris **322** (1996), 107–112.

[47] A.M.Garsia and C.Reutenauer, *A decomposition of Solomon's descent algebra*, Adv. in Math, **77** (1989), 189–262.

[48] 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.

[49] I.M.Gelfand and V.S.Retakh, *Determinants of matrices over noncommutative rings*, Funct. Anal. Appl. **25**, (1991), 91–102.

[50] 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.

[51] I.Gessel, *Multipartite P-partitions and inner product of skew Schur functions*, Contemporary Math. **34** (1984), 289–301.

[52] A.Klyachko, *Lie elements in the tensor algebra*, Siberian Math. J. **15** (1974), 1296–1304.

[53] D.Krob, B.Leclerc and J.-Y.Thibon, *Noncommutative symmetric functions II: Transformations of alphabets*, Publ. IGM **95-5**, Université de Marne-la-Vallée, 1995.

[54] D.Krob and J.-Y.Thibon, *Noncommutative symmetric functions IV*, preprint LITP, Université Paris 7, 1996.

[55] B.Leclerc, T.Scharf and J.-Y.Thibon, *Noncommutative cyclic characters of symmetric groups*, preprint, 1994, to appear in J. Comb. Theory A.

[56] P.A.MacMahon, *Combinatory analysis*, Cambridge University Press, 1915, 1916; Chelsea reprint, 1960.

[57] C. Malvenuto, *Produits et coproduits des fonctions quasi-symmétriques et de l'algèbre des descentes (Thesis)*, LACIM, UQAM, Montréal, 1993.

[58] C.Malvenuto and C.Reutenauer, *Duality between quasi-symmetric functions and the Solomon descent algebra*, preprint UQAM, Montréal, 1993.

[59] B.Mielnik and J.Plebański, *Combinatorial approach to Baker-Campbell-Hausdorff exponents*, Ann. Inst. Henri Poincaré, Section A vol. **XII** (1970), 215–254.

[60] C.Reutenauer, *Free Lie algebras*, Oxford University Press, 1993.

[61] T.Scharf and J.-Y. Thibon, *On Witt vectors and symmetric functions*, to appear in Algebra Colloquium.

[62] L.Solomon, *A Mackey formula in the group ring of a Coxeter group*, J. Algebra **41** (1976), 255–268.

Schubert polynomials

[63] 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.

[64] I.V.Cherednik, *A new interpretation of Gelfand-Tzetlin bases*, Duke. Math. J. **54** (1987), 563–577.

[65] M.Demazure, *Désingularisation des variétés de Schubert*, Annales E.N.S. **6** (1974), 163–172.

[66] S.Fomin and A.N.Kirillov, *Combinatorial* B_n*-analogues of Schubert polynomials*, preprint, 1994.

[67] W.Fulton, *Flags, Schubert polynomials, degeneracy loci and determinantal formulas*, Duke Math.J. **65** (1991), 381-420.

[68] F.Hirzebruch, *Topological Methods in Algebraic Geometry*, 3rd ed., Springer-Verlag, 1978.

[69] A.Kohnert, *Weintrauben, Polynome, Tableaux*, Bayreuther Math. Schr. **38** (1991), 1–97.

[70] A.Kohnert and S.Veigneau, *Using Schubert toolkit to compute with polynomials in several variables*, Proceedings of the 8-*th* Conference "Formal Power Series and Algebraic Combinatorics", Minneapolis.

[71] A.Lascoux, *Polynômes de Schubert. Une approche historique*, Actes du 4^e colloque "Séries formelles et combinatoire algébrique", Montréal 1992, Publ. LACIM, UQAM, 1992.

[72] A.Lascoux, B.Leclerc and J.-Y.Thibon, Flag Varieties and the Yang-Baxter Equation, preprint, 1995.

[73] A.Lascoux, B.Leclerc and J.-Y.Thibon, *Twisted action of the symmetric group on the cohomology of a flag manifold*, Banach Center Publ. **36**, 1996, 111–124.

[74] A.Lascoux and M.P.Schützenberger, *Schubert polynomials and the Littlewood-Richardson Rule*, Lett. Math. Phys. **10** (1985), 111–124.

[75] A.Lascoux and M.P.Schützenberger, *Symmetry and Flag manifolds*, Springer L.N. **996** (1983), 118–144.

[76] A.Lascoux, *Classes de Chern des variétés de drapeaux*, C.R. Acad. Sci. Paris **295** (1982), 393–398.

[77] A.Lascoux and M.P.Schützenberger, *Polynômes de Schubert*, C.R. Acad. Sci. Paris **294** (1982), 447–450.

[78] A.Lascoux and M.P.Schützenberger, *Structure de Hopf de l'anneau de cohomologie et de l'anneau de Grothendieck d'une variété de drapeaux*, C.R. Acad. Sci. Paris **295** (1982), 629–633.

[79] I.G.Macdonald, *Notes on Schubert polynomials*, Publ. LACIM **6**, UQAM, Montréal, 1991.

[80] D.Monk, *the Geometry of flags manifolds*, Proc. London M.S. (3)**9** (1959), 253–286.

[81] P.Pragacz and J.Ratajski, *Formulas for Lagrangian and orthogonal degeneraci loci; The* $\widetilde{Q}$*-Polynomials Approach*, to appear in Compositio Math.

[82] V.Prosper, *Action of Divided Differences on Modified Powers*, preprint, Université de Marne-la-Vallée, 1998.

[83] S.Veigneau, *SP, a package on Schubert polynomials realized with the computer algebra system MAPLE*, Publ. IGM **95-1**, Université de Marne-la-Vallée, 1995, Journal of Symbolic Computation **23** (1997), 413–425.

Schubert polynomials and computations in $\mathbb{Z}[x_1, \ldots, x_n]$

[84] S.Fomin and C.Greene, *Noncommutative Schur functions and their applications*, preprint, 1995.

[85] S.Fomin and A.Kirillov, *Yang-Baxter equation, symmetric functions and Schubert polynomials*, Proceedings of the 5-*th* Conference "Formal Power Series and Algebraic Combinatorics", Florence 1993, 215–229.

[86] W.Fulton and A.Lascoux, *A Pieri formula in the Grothendieck ring of a flag bundle*, Duke Math.J. **76** (1994), 711–729.

[87] A.Kohnert and S.Veigneau, *Using Schubert Basis to Compute with Multivariate Polynomials*, Advances in Applied Mathematics **19** (1997), 45–60.

Symmetric functions

[88] I.G.Macdonald, *Symmetric functions and Hall polynomials*, 2nd ed., Oxford: Clarendon Press, 1995.

[89] D.Knutson, *λ-rings and the representation theory of the symmetric group*, Springer L.N.M. **308**, 1973.

[90] D.E.Littlewood, *The theory of group characters*, 2nd ed., Oxford, 1950.

[91] V.Prosper, *SFA, a package on symmetric functions considered as operators over the ring of polynomials for the computer algebra system Maple*, submitted to Journal of Symbolic Computation, 1998.

[92] J.Stembridge, *SF, a maple package for symmetric functions*, University of Michigan, 1993.

Words and tableaux

[93] D.E.Knuth, *Permutations, matrices and generalized young tableaux*, Pacific J. Math. **34** (1970), 709–727.

[94] A.Lascoux and M.P.Schützenberger, *Keys and standard bases, in invariant theory and tableaux*, D. Stanton ed., Springer-Verlag (1990), 125–144.

[95] A.Lascoux and M.P.Schützenberger, *Le monoïde plaxique*, Quaderni de la Ricerca Scientifica, **109** (1981), 129–156.

[96] C.Schensted, *Longest increasing and decreasing subsequences*, Canad. J. Math. **13** (1961), 179–191.

Index

Les dictionnaires sont les seuls ouvrages dont les mots
sont classés selon un certain ordre—l'alphabétique—qui vaut bien
tous les autres : ordre du jour ou de bataille, ordre des choses ou ordre établi,
ordre des avocats ou ordre dorique,
ordre monastique ou—le pire de tous—l'ordre pour l'ordre.

Georges Elgozy.

A

ACE, 128–135
    ACE, 128
    AceBanner, 129
    AceBuildData, 130
    AceLoadData, 131
    AceSysInfo, 132
    AceTime, 133
    AceUnloadData, 134
    AceWhichIs, 134
AceBanner, 129
AceBuildData, 130
AceLoadData, 131
AceSysInfo, 132
AceTime, 133
AceUnloadData, 134
AceWhichIs, 134

B

Background
    BN, *58*
    BnBar2Code, *59*
    BnBar2Mat, *59*
    BnCode2Bar, *59*
    BnCode2Interval, *58*
    BnCode2Inv, *59*
    BnCode2Length, *58*
    BnCode2ListRd, *58*
    BnCode2Palin, *59*
    BnCode2Rd, *58*
    BnGaMult, *60*
    BnGaOnPol, *60*
    BnGaYang, *61*
    BnGenCode, *58*
    BnIdcaMult, *60*
    BnIdcaOnPol, *60*
    BnListCode, *58*
    BnMat2Bar, *59*
    BnMultBar, *59*
    BnMultCode, *59*
    BnNcaMult, *60*
    BnNcaOnPol, *60*
    BnNcaYang, *61*
    BnPalin2Code, *59*
    BnTranspoPalin, *59*
    BnaAdd, *60*
    BnaCode2Bar, *61*
    BnaCode2Palin, *61*
    BnaMinus, *60*
    BumpC, *63*
    Bump, *62*
    CTab2Mat, *63*
    CTab2Word, *63*
    CgMultiplicity, *40*
    CgPositiveRoots, *40*
    CgRho, *40*
    CgSchurSeries, *36*
    FM_n, *80*
    Frob2Part, *36*
    HEKA, *34*
    HekaOnPol, *34*
    HekaYang, *34*
    IDCA, *31*, *33*
    IdcaOnPol, *33*
    IdcaYang, *33*
    ListCompo, *49*
    ListPerm, *27*
    ListTab, *63*
    ListYama, *64*
    NCA, *30*, *33*
    NcaOnPol, *33*
    NcaYang, *33*
    NcsfInternal, *52*
    NcsfTransAlpha, *54*
    O2s, *37*
    ODimRep, *39*
    OJtMat, *37*
    PART, *36*
    Part2Frob, *36*
    Part2ListStdTab, *63*
    Perm2Betti, *31*
    Perm2Code, *28*
    Perm2CycleType, *42*
    Perm2Cycle, *42*
    Perm2Desc, *28*
    Perm2Interval, *31*

Perm2Length, *28*
Perm2ListInv, *28*
Perm2ListRd, *29*
Perm2Rd, *29*
SGA, *31*
SG, *27*
SP, *33*
Schensted, *65*
Sf2Char, *42*
SfInternal, *43*
SfScalar, *41*
SfZee, *41*
Sga2Carre, *32*
Sga2Nabla, *32*
SgaJucis, *31*
SgaOnFree, *65*
Sp2s, *37*
SpDimRep, *38*
SpJtMat, *37*
StdTab2SgaIdemp, *31*
Tab2Mat, *62*
Tab2Part, *63*
Tab2Word, *62*
ToO, *38*
ToSp, *38*, *39*
ToS, *53*
ToXfixScal, *80*
ToXfix, *80*
Toe, *41*
Toh, *41*
Tom, *43*
Top, *41*
Tos, *41*
Word2CTab, *63*
Word2Charge, *65*
Word2CoCharge, *65*
Yama2StdTab, *63*

BN, 246–257
BN, 246
BnBar2Code, 247
BnBar2Mat, 247
BnCode2Bar, 248
BnCode2Interval, 248
BnCode2Inv, 249
BnCode2Length, 250
BnCode2ListRd, 250
BnCode2Palin, 251
BnCode2Rd, 252
BnGenCode, 252
BnListCode, 253
BnMat2Bar, 254
BnMultBar, 254
BnMultCode, 255
BnPalin2Code, 256
BnRd2Code, 256
BnTranspoPalin, 257

BNA, 258–275
BNA, 258
Bna2Table, 267
BnaAdd, 268
BnaCode2Bar, 268
BnaCode2Palin, 269
BnaMinus, 270
BnaNormal, 270
BnaPalin2Code, 271
BnGaMult, 259
BnGaOnPol, 259
BnGaYang, 260
BnIdcaMult, 261
BnIdcaOnPol, 262
BnIdcaYang, 263
BnNcaMult, 264
BnNcaOnPol, 265
BnNcaYang, 266
Rd2BnGa, 272
Rd2BnIdca, 273
Rd2BnNca, 273
Table2Bna, 274

Bna2Table, 267
BnaAdd, 267
BnaCode2Bar, 268
BnaCode2Palin, 269
BnaMinus, 270
BnaNormal, 270
BnaPalin2Code, 271
BnBar2Code, 247
BnBar2Mat, 247
BnCode2Bar, 248
BnCode2Interval, 248
BnCode2Inv, 249
BnCode2Length, 250
BnCode2ListRd, 250
BnCode2Palin, 251
BnCode2Rd, 252
BnGaMult, 259
BnGaOnPol, 259
BnGaYang, 260
BnGenCode, 252
BnIdcaMult, 261
BnIdcaOnPol, 262
BnIdcaYang, 263
BnListCode, 253
BnMat2Bar, 254
BnMultBar, 254
BnMultCode, 255
BnNcaMult, 264
BnNcaOnPol, 265
BnNcaYang, 266
BnPalin2Code, 256
BnRd2Code, 256
BnTranspoPalin, 257
Border2Part, 136
Bump, 361
BumpC, 362

C

CG, 237–245
CG, 237
CgMultiplicity, 237
CgPositiveRoots, 238
CgRho, 239
CgSchurSeries, 239
O2s, 240
ODimRep, 241
OJtMat, 241
Sp2s, 242
SpDimRep, 243
SpJtMat, 243
ToO, 244
ToSp, 245
CgMultiplicity, 237
CgPositiveRoots, 238
CgRho, 239
CgSchurSeries, 239
Char2Sf, 277
CLG, 330–340
CLG, 330
CLG_n, 331
Gauss, 331
GlDimRep, 332
Toe_n, 333
Toh_n, 334
Tom_n, 335
Top_n, 336
Tos_n, 337

Tox_n, 338
x2m_n, 339
CLG_n, 331
Code2Perm, 161
COMP, 154–159
COMP, 154
CompCompo, 154
Compo2Conjugate, 155
Compo2MajorIndex, 156
Compo2Mat, 156
ListCompo, 157
ListCompoFatter, 158
ListCompoFiner, 159
CompCompo, 154
Compo2Conjugate, 155
Compo2MajorIndex, 156
Compo2Mat, 156
CompPart, 137
CompPerm, 161
CTab2Mat, 363
CTab2Part, 363
CTab2Word, 364
Cycle2Perm, 162

D

Desc2ListPerm, 163
Diagonal2Part, 138

E

Exp2Part, 139

F

FactorPerm, 164
Flag, 383
Flag variety, 113
FM, 401–409
FM, 401
FM_n, 401
ToXfix, 402
ToXfixScal, 403
X2Xfix, 405
x2Xfix, 407
x2XfixScal, 408
Xfix2X, 406
Xfix2x, 407
FM_n, 401
FREE, 316–329
FREE, 316
Free2NilFree, 317
Free2Plax, 317
Free2PlaxClass, 318
Free2Pol, 318
Free2StdFree, 319
Free2Table, 320
FreeAdd, 320
FreeConcat, 321
FreeMinus, 322
FreeNormal, 323
FreeShuffle, 323
IdcaOnFree, 324
NilFree2Free, 325
Pol2Free, 326
Pol2Row, 326
Sf2Free, 327
SgaOnFree, 328
Table2Free, 328
Free2NilFree, 317
Free2Plax, 317
Free2PlaxClass, 318
Free2Pol, 318
Free2StdFree, 319
Free2Table, 320
FreeAdd, 320
FreeConcat, 321
FreeMinus, 322
FreeNormal, 323
FreeShuffle, 323
Frob2Part, 139

G

Gauss, 331
GenPerm, 164
GlDimRep, 332
Grothendieck polynomials, 107

H

HEKA, 224–236
HEKA, 224
Heka2Table, 225
HekaAdd, 225
HekaCharTable, 226
HekaJucis, 227
HekaMinus, 228
HekaMult, 229
HekaNormal, 230
HekaOnPol, 231
HekaOnX, 232
HekaOnXfix, 233
HekaYang, 234
Rd2Heka, 235
Table2Heka, 235
Heka2Table, 225
HekaAdd, 225
HekaCharTable, 226
HekaJucis, 227
HekaMinus, 228
HekaMult, 229
HekaNormal, 230
HekaOnPol, 231
HekaOnX, 232
HekaOnXfix, 233
HekaYang, 234

I

IDCA, 213–223
IDCA, 213
Idca2Demazure, 214
Idca2Table, 214
IdcaAdd, 215
IdcaMinus, 216
IdcaMult, 217
IdcaNormal, 217
IdcaOnPol, 218
IdcaOnX, 219
IdcaOnXfix, 220
IdcaYang, 221
Rd2Idca, 222
Table2Idca, 223
Idca2Demazure, 214
Idca2Table, 214
IdcaAdd, 215
IdcaMinus, 216
IdcaMult, 217
IdcaNormal, 217
IdcaOnFree, 324
IdcaOnPol, 218
IdcaOnX, 219

IdcaOnXfix, 220
IdcaYang, 221
Installation, 95
  MACINTOSH, 103
  MS-DOS, 100
  UNIX, 95
InvSchensted, 365

K

Key polynomials, 110

L

_liblist, 98, 102, 105
libname, 98, 102, 105
Lie idempotent, 114
ListCompo, 157
ListCompoFatter, 158
ListCompoFiner, 159
ListPart, 140
ListPartIn, 141
ListPerm, 165
ListSkewDiag, 142
ListStdTab, 366
ListTab, 366
ListYama, 368

M

Mat2CTab, 368
Mat2Perm, 167
Mat2Tab, 369
Modified powers, 120
MultPerm, 167

N

NCA, 200–212
  NCA, 200
  Nca2Table, 201
  NcaAdd, 201
  NcaMinus, 202
  NcaMult, 203
  NcaNormal, 204
  NcaOnPol, 204
  NcaOnX, 205
  NcaOnXfix, 207
  NcaOnXX, 206
  NcaOnY, 208
  NcaOnYY, 209
  NcaYang, 210
  Rd2Nca, 211
  Table2Nca, 212
Nca2Table, 201
NcaAdd, 201
NcaMinus, 202
NcaMult, 203
NcaNormal, 204
NcaOnPol, 204
NcaOnX, 205
NcaOnXfix, 207
NcaOnXX, 206
NcaOnY, 208
NcaOnYY, 209
NcaYang, 210
NCSF, 341–359
  NCSF, 341
  NcsfInternal, 341
  NcsfMat, 342
  NcsfPairing, 343
  NcsfPlethysm, 344
  NcsfTransAlpha, 345
  Qsf2x, 346
  QsfMat, 347
  Sf2Qsf, 348
  ToE, 349
  ToF, 350
  ToL, 351
  ToM, 352
  ToPh, 353
  ToPs, 354
  ToQPh, 355
  ToQPs, 356
  ToR, 357
  ToS, 358
NcsfInternal, 341
NcsfMat, 342
NcsfPairing, 343
NcsfPlethysm, 344
NcsfTransAlpha, 345
NewtonInterp, 383
NilFree2Free, 325

O

O2s, 240
ODimRep, 241
OJtMat, 241

P

PART, 136–153
  Border2Part, 136
  CompPart, 137
  Diagonal2Part, 138
  Exp2Part, 139
  Frob2Part, 139
  ListPart, 140
  ListPartIn, 141
  ListSkewDiag, 142
  PART, 136
  Part2Border, 144
  Part2Conjugate, 145
  Part2Diagonal, 145
  Part2Exp, 146
  Part2Frob, 147
  Part2ListHook, 147
  Part2Mat, 148
  Part2PCore, 149
  PartOrderMat, 150
  PCore2Part, 143
  SkewPart2Mat, 152
  SplitPart, 153
Part2Border, 144
Part2Conjugate, 145
Part2Diagonal, 145
Part2Exp, 146
Part2Frob, 147
Part2ListHook, 147
Part2ListStdTab, 370
Part2Mat, 148
Part2PCore, 149
Part2SgaCIdemp, 187
Part2SgCC, 168
PartOrderMat, 150
PCore2Part, 143
Perm2Betti, 169
Perm2Code, 169
Perm2Cycle, 170
Perm2CycleType, 171

Perm2Desc, 171
Perm2Interval, 172
Perm2Inv, 173
Perm2Length, 173
Perm2ListInv, 174
Perm2ListRd, 175
Perm2Mat, 175
Perm2Rd, 177
Perm2Rise, 177
Perm2Rothe, 178
Perm2RRep, 176
Perm2Sign, 179
PermOnCTab, 371
PermOnTab, 371
Pol2Free, 326
Pol2Row, 326
Pol2SfA, 303
Projective degree, 112

Q

Qsf2x, 346
QsfMat, 347

R

RandPerm, 181
Rd2BnGa, 272
Rd2BnIdca, 273
Rd2BnNca, 273
Rd2Heka, 235
Rd2Idca, 222
Rd2Nca, 211
Rd2Perm, 181
Rd2Sga, 187
RRep2Perm, 180

S

Schensted, 372
Schubert variety, 112
Sf2Char, 277
Sf2Free, 327
Sf2Qsf, 348
Sf2SfA, 303
Sf2Table, 278
Sf2TableVar, 279
SFA, 302–315
  Pol2SfA, 303
  Sf2SfA, 304
  SFA, 302
  SfA2Sf, 304
  SfA2TableVar, 305
  SfACollect, 307
  SfAExpand, 307
  SfAOmega, 309
  SfAVars, 310
  ToeA, 311
  TohA, 312
  TomA, 312
  TopA, 313
  TosA, 314
SfA2Sf, 304
SfA2TableVar, 305
SfACollect, 306
SfAddBasis, 280
SfAExpand, 307
SfAOmega, 309
SfAVars, 310
SfCCProd, 282
SfDiff, 283
SfDualBasis, 284
SfEval, 286
SfInternal, 286
SfJtMat, 287
SfMat, 288
SfOmega, 289
SfPlethysm, 290
SfScalar, 291
SfTheta, 292
SfZee, 293
SG, 160–185
  Code2Perm, 161
  CompPerm, 161
  Cycle2Perm, 162
  Desc2ListPerm, 163
  FactorPerm, 164
  GenPerm, 164
  ListPerm, 165
  Mat2Perm, 167
  MultPerm, 167
  Part2SgCC, 168
  Perm2Betti, 169
  Perm2Code, 169
  Perm2Cycle, 170
  Perm2CycleType, 171
  Perm2Desc, 171
  Perm2Interval, 172
  Perm2Inv, 173
  Perm2Length, 173
  Perm2ListInv, 174
  Perm2ListRd, 175
  Perm2Mat, 175
  Perm2Rd, 177
  Perm2Rise, 177
  Perm2Rothe, 178
  Perm2RRep, 176
  Perm2Sign, 179
  RandPerm, 181
  Rd2Perm, 181
  RRep2Perm, 180
  SG, 160
  SgCharTable, 182
  SgDimRep, 183
  SgTranspo, 184
  Vect2Perm, 185
SGA, 186–199
  Part2SgaCIdemp, 187
  Rd2Sga, 187
  SGA, 186
  Sga2Carre, 188
  Sga2Nabla, 189
  Sga2Table, 190
  SgaAdd, 191
  SgaJucis, 191
  SgaMinus, 192
  SgaMult, 193
  SgaNormal, 194
  SgaOnPol, 195
  SgaOnX, 195
  SgaOnXfix, 196
  SgaYang, 198
  Table2Sga, 198
Sga2Carre, 188
Sga2Nabla, 189
Sga2Table, 190
SgaAdd, 191
SgaJucis, 191
SgaMinus, 192
SgaMult, 193
SgaNormal, 194
SgaOnFree, 328
SgaOnPol, 195
SgaOnX, 195
SgaOnXfix, 196

SgaYang, 198
SgCharTable, 182
SgDimRep, 183
SgTranspo, 184
SkewPart2Mat, 152
SP, 382–400
  Flag, 383
  NewtonInterp, 383
  SP, 382
  Sp2TableVar, 384
  SpScalarPol, 386
  SpSpecialize, 386
  TableX, 388
  TableXX, 389
  TableY, 390
  TableYY, 391
  ToX, 392
  Tox, 397
  ToXX, 393
  ToY, 394
  ToYY, 395
  X2Dim, 398
  x2X, 399
  x2XX, 399
Sp2s, 242
Sp2TableVar, 384
SpDimRep, 243
SpJtMat, 243
SplitPart, 153
SpScalarPol, 385
SpSpecialize, 386
StdTab2SgaIdemp, 373
StdTab2Yama, 374
SYMF, 276–301
  Char2Sf, 277
  Sf2Char, 277
  Sf2Table, 278
  Sf2TableVar, 279
  SfAddBasis, 281
  SfCCProd, 282
  SfDiff, 283
  SfDualBasis, 284
  SfEval, 286
  SfInternal, 286
  SfJtMat, 287
  SfMat, 288
  SfOmega, 289
  SfPlethysm, 290
  SfScalar, 291
  SfTheta, 292
  SfZee, 293
  SYMF, 276
  Table2Sf, 294
  Toc, 295
  Toh, 297
  Tom, 298
  Top, 299
  Tos, 300

T

TAB, 360–381
  Bump, 361
  BumpC, 362
  CTab2Mat, 363
  CTab2Part, 363
  CTab2Word, 364
  InvSchensted, 365
  ListStdTab, 366
  ListTab, 367
  ListYama, 368
  Mat2CTab, 368
  Mat2Tab, 369
  Part2ListStdTab, 370
  PermOnCTab, 371
  PermOnTab, 371
  Schensted, 372
  StdTab2SgaIdemp, 373
  StdTab2Yama, 374
  TAB, 360
  Tab2Mat, 375
  Tab2Part, 375
  Tab2Word, 376
  Word2Charge, 377
  Word2CoCharge, 378
  Word2CTab, 377
  Word2Key, 379
  Word2Tab, 380
  Yama2StdTab, 381
Tab2Mat, 375
Tab2Part, 375
Tab2Word, 376
Table2Bna, 274
Table2Free, 328
Table2Heka, 235
Table2Idca, 223
Table2Nca, 212
Table2Sf, 294
Table2Sga, 198
TableX, 388
TableXX, 389
TableY, 390
TableYY, 391
Toc, 295
Todd class, 113
ToE, 349
Toe, 296
Toe_n, 333
ToeA, 311
ToF, 350
Toh, 297
Toh_n, 334
TohA, 312
ToL, 351
ToM, 352
Tom, 298
Tom_n, 335
TomA, 312
ToO, 244
Top, 299
Top_n, 336
TopA, 313
ToPh, 353
ToPs, 354
ToQPh, 355
ToQPs, 356
ToR, 357
ToS, 358
Tos, 300
Tos_n, 337
TosA, 314
ToSp, 245
ToX, 392
Tox, 397
Tox_n, 338
ToXfix, 402
ToXfixScal, 403
ToXX, 393
ToY, 394
ToYY, 395

V

Vect2Perm, 185

W

`Word2Charge`, 377
`Word2CoCharge`, 378
`Word2CTab`, 377
`Word2Key`, 379
`Word2Tab`, 380

X

`X2Dim`, 398
`x2m_n`, 339
`x2X`, 399
`X2Xfix`, 405
`x2Xfix`, 407
`x2XfixScal`, 408
`x2XX`, 399
`Xfix2X`, 406
`Xfix2x`, 407

Y

`Yama2StdTab`, 381