SPECHT

A GAP package for calculating the

Decomposition matrices of Hecke algebras of type A

Version 2.4

Andrew Mathas

School of Mathematics, University of Sydney
Sydney N.S.W. 2006 Australia,

e–mail: mathas@maths.usyd.edu.au

The latest version of SPECHT *can be obtained from*
`http://www.maths.usyd.edu.au:8000/u/mathas/specht`

October 1997

Contents

1 Decomposition numbers of Hecke algebras of type A **1**

1.1 Specht . . . 4

Decomposition numbers of the symmetric groups . . . 7

Hecke algebras over fields of positive characteristic . . . 7

The Fock space and Hecke algebras over fields of characteristic zero . . . 8

1.2 Schur . . . 9

1.3 DecompositionMatrix . . . 10

1.4 CrystalDecompositionMatrix . . . 11

1.5 DecompositionNumber . . . 12

Partitions in SPECHT . . . 12

Inducing and restricting modules . . . 13

1.6 InducedModule . . . 13

1.7 SInducedModule . . . 15

1.8 RestrictedModule . . . 15

1.9 SRestrictedModule . . . 16

Operations on decomposition matrices . . . 16

1.10 InducedDecompositionMatrix . . . 17

1.11 IsNewIndecomposable . . . 17

1.12 InvertDecompositionMatrix . . . 19

1.13 AdjustmentMatrix . . . 19

1.14 SaveDecompositionMatrix . . . 20

1.15 CalculateDecompositionMatrix . . . 20

1.16 MatrixDecompositionMatrix . . . 21

1.17 DecompositionMatrixMatrix . . . 21

1.18 AddIndecomposable . . . 22

1.19 RemoveIndecomposable . . . 22

1.20 MissingIndecomposables . . . 22
Calculating dimensions . . . 23
1.21 SimpleDimension . . . 23
1.22 SpechtDimension . . . 23
Combinatorics on Young diagrams . . . 23
1.23 Schaper . . . 24
1.24 IsSimpleModule . . . 24
1.25 MullineuxMap . . . 25
1.26 MullineuxSymbol . . . 26
1.27 PartitionMullineuxSymbol . . . 26
1.28 GoodNodes . . . 26
1.29 NormalNodes . . . 27
1.30 GoodNodeSequence . . . 27
1.31 PartitionGoodNodeSequence . . . 27
1.32 GoodNodeLatticePath . . . 28
1.33 LittlewoodRichardsonRule . . . 28
1.34 InverseLittlewoodRichardsonRule . . . 29
1.35 EResidueDiagram . . . 29
1.36 HookLengthDiagram . . . 30
1.37 RemoveRimHook . . . 30
1.38 AddRimHook . . . 31
Operations on partitions . . . 31
1.39 ECore . . . 31
1.40 IsECore . . . 32
1.41 EQuotient . . . 32
1.42 CombineEQuotientECore . . . 32
1.43 EWeight . . . 32
1.44 ERegularPartitions . . . 33
1.45 IsERegular . . . 33
1.46 ConjugatePartition . . . 33
1.47 BetaSet . . . 33
1.48 PartitionBetaSet . . . 33
1.49 ETopLadder . . . 34
1.50 Dominates . . . 34
1.51 LengthLexicographic . . . 34
1.52 Lexicographic . . . 34

1.53 ReverseDominance . . . 35
Miscellaneous functions on modules . . . 35
1.54 Specialized . . . 35
1.55 ERegulars . . . 36
1.56 SplitECores . . . 36
1.57 Coefficient . . . 37
1.58 InnerProduct . . . 37
1.59 SpechtPrettyPrint . . . 37
Semi–standard and standard tableaux . . . 37
1.60 SemiStandardTableaux . . . 38
1.61 StandardTableaux . . . 38
1.62 ConjugateTableau . . . 38
1.63 ShapeTableau . . . 38
1.64 TypeTableau . . . 39
Installing SPECHT . . . 39
Installing SPECHTs online documentation . . . 39

Chapter 1

Decomposition numbers of Hecke algebras of type A

This package contains functions for computing the decomposition matrices for Iwahori–Hecke algebras of the symmetric groups. As the (modular) representation theory of these algebras closely resembles that of the (modular) representation theory of the symmetric groups — indeed, the later is a special case of the former — many of the combinatorial tools from the representation theory of the symmetric group are included in the package.

These programs grew out of the attempts by Gordon James and myself [JM1] to understand the decomposition matrices of Hecke algebras of type **A** when $q = -1$. The package is now much more general and its highlights include:

1. SPECHT provides a means of working in the Grothendieck ring of a Hecke algebra H using the three natural bases corresponding to the Specht modules, projective indecomposable modules, and simple modules.

2. For Hecke algebras defined over fields of characteristic zero we have implemented the algorithm of Lascoux, Leclerc, and Thibon [LLT] for computing decomposition numbers and "crystallized decomposition matrices". In principle, this gives all of the decomposition matrices of Hecke algebras defined over fields of characteristic zero.

3. We provide a way of inducing and restricting modules. In addition, it is possible to "induce" decomposition matrices; this function is quite effective in calculating the decomposition matrices of Hecke algebras for small n.

4. The q–analogue of Schaper's theorem [JM2] is included, as is Kleshchev's [K] algorithm of calculating the Mullineux map. Both are used extensively when inducing decomposition matrices.

5. SPECHT can be used to compute the decomposition numbers of q–Schur algebras (and the general linear groups), although there is less direct support for these algebras. The decomposition matrices for the q–Schur algebras defined over fields of characteristic zero for $n < 11$ and all e are included in SPECHT.

6. The Littlewood–Richard rule, its inverse, and functions for many of the standard operations on partitions (such as calculating cores, quotients, and adding and removing hooks), are included.

7. The decomposition matrices for the symmetric groups S_n are included for $n < 15$ and for all primes.

The modular representation theory of Hecke algebras

The "modular" representation theory of the Iwahori–Hecke algebras of type **A** was pioneered by Dipper and James [DJ1,DJ2]; here we briefly outline the theory, referring the reader to the references for details. The definition of the Hecke algebra can be found in Chapter 81; see also 81.1.

Given a commutative integral domain R and a non–zero unit q in R, let $H = H_{R,q}$ be the Hecke algebra of the symmetric group S_n on n symbols defined over R and with parameter q. For each partition μ of n, Dipper and James defined a **Specht module** $\mathtt{S}(\mu)$. Let `rad` $\mathtt{S}(\mu)$ be the radical of $\mathtt{S}(\mu)$ and define $\mathtt{D}(\mu)$=$\mathtt{S}(\mu)$/`rad` $\mathtt{S}(\mu)$. When R is a field, $\mathtt{D}(\mu)$ is either zero or absolutely irreducible. Henceforth, we will always assume that `R` is a field.

Given a non–negative integer i, let $[i]_q = 1 + q + \ldots + q^{i-1}$. Define e to be the smallest non–negative integer such that $[e]_q = 0$; if no such integer exists, we set e equal to 0. *Many of the functions in this package depend upon e;* the integer e is the Hecke algebras analogue of the characteristic of the field in the modular representation theory of finite groups.

A partition $\mu = (\mu_1, \mu_2, \ldots)$ is e–**singular** if there exists an integer i such that $\mu_i = \mu_{i+1} = \cdots = \mu_{i+e-1} > 0$; otherwise, μ is e–**regular**. Dipper and James [DJ1] showed that $\mathtt{D}(\nu) \neq (0)$ if and only if ν is e–regular and that the $\mathtt{D}(\nu)$ give a complete set of non–isomorphic irreducible H–modules as ν runs over the e–regular partitions of n. Further, $\mathtt{S}(\mu)$ and $\mathtt{S}(\nu)$ belong to the same block if and only if μ and ν have the same e-core [DJ2,JM2]. Note that these results depend only on e and not directly on R or q.

Given two partitions μ and ν, where ν is e–regular, let $d_{\mu\nu}$ be the composition multiplicity of $\mathtt{D}(\nu)$ in $\mathtt{S}(\mu)$. The matrix $D = (d_{\mu\nu})$ is the **decomposition matrix** of `H`. When the rows and columns are ordered in a way compatible with dominance, D is lower unitriangular.

The indecomposable H-modules $\mathtt{P}(\nu)$ are indexed by e-regular partitions ν. By general arguments, $\mathtt{P}(\nu)$ has the same composition factors as $\sum_\mu d_{\mu\nu}\mathtt{S}(\mu)$; so these linear combinations of modules become identified in the Grothendieck ring of `H`. Similarly, $\mathtt{D}(\nu) = \sum_\mu d^{-1}_{\nu\mu}\mathtt{S}(\mu)$ in the Grothendieck ring. These observations are the basis for many of the computations in SPECHT.

Two small examples

Because of the algorithm of [LLT], in principle, all of decomposition matrices for all Hecke algebras defined over fields of characteristic zero are known and available using SPECHT. The algorithm is recursive; however, it is quite quick and, as with a car, you need never look at the engine:

```
gap> H:=Specht(4);    #  e=4, R a field of characteristic 0
Specht(e=4, S(), P(), D(), Pq())
gap> InducedModule(H.P(12,2));
P(13,2)+P(12,3)+P(12,2,1)+P(10,3,2)+P(9,6)
```

The [LLT] algorithm was applied 24 times during this calculation.

For Hecke algebras defined over fields of positive characteristic the major tool provided by SPECHT, apart from the decomposition matrices contained in the libraries, is a way of "inducing" decomposition matrices. This makes it fairly easy to calculate the associated

decomposition matrices for "small" n. For example, the SPECHT libraries contain the decomposition matrices for the symmetric groups S_n over fields of characteristic 3 for $n < 15$. These matrices were calculated by SPECHT using the following commands:

```
gap> H:=Specht(3,3);    #  e=3, R field of characteristic 3
Specht(e=3, p=3, S(), P(), D())
gap> d:=DecompositionMatrix(H,5);  #  known for n < 2e
5     | 1
4,1   | . 1
3,2   | . 1 1
3,1^2 | . . . 1
2^2,1 | 1 . . . 1
2,1^3 | . . . . 1
1^5   | . . 1 . .
gap> for n in [6..14] do
>       d:=InducedDecompositionMatrix(d); SaveDecompositionMatrix(d);
>    od;
```

The function `InducedDecompositionMatrix` contains almost every trick that I know for computing decomposition matrices (except using the spin groups). I would be very happy to hear of any improvements.

SPECHT can also be used to calculate the decomposition numbers of the q–Schur algebras; although, as yet, here no additional routines for calculating the projective indecomposables indexed by e–singular partitions. Such routines will probably be included in a future release, together with the (conjectural) algorithm [LT] for computing the decomposition matrices of the q–Schur algebras over fields of characteristic zero.

In the next release of SPECHT, I will also include functions for computing the decomposition matrices of Hecke algebras of type **B**, and more generally those of the Ariki–Koike algebras. As with the Hecke algebra of type **A**, there is an algorithm for computing the decomposition matrices of these algebras when `R` is a field of characteristic zero [M].

Credits

I would like to thank Gordon James, Johannes Lipp, and Klaus Lux for their comments and suggestions.

If you find SPECHT useful please let me know. I would also appreciate hearing any suggestions, comments, or improvements. In addition, if SPECHT does play a significant role in your research, please send me a copy of the paper(s) and please cite SPECHT in your references.

The lastest version of SPECHT can be obtained from http://maths.usyd.edu.au:8000/u/mathas/specht.

Andrew Mathas[1]
mathas@maths.usyd.edu.au
University of Sydney, 1997.

References

[A] S. Ariki, *On the decomposition numbers of the Hecke algebra of* $G(m, 1, n)$, J. Math. Kyoto Univ., **36** (1996), 789–808.

[1] Supported in part by SERC grant GR/J37690

[B] J. Brundan, *Modular branching rules for quantum GL_n and the Hecke algebra of type* **A**, Proc. London Math. Soc (3), to appear.

[DJ1] R. Dipper and G. James, *Representations of Hecke algebras of general linear groups*, Proc. London Math. Soc. (3), **52** (1986), 20–52.

[DJ2] R. Dipper and G. James, *Blocks and idempotents of Hecke algebras of general linear groups*, Proc. London Math. Soc. (3), **54** (1987), 57–82.

[G] M. Geck, *Brauer trees of Hecke algebras*, Comm. Alg., **20** (1992), 2937–2973.

[Gr] I. Grojnowski, *Affine Hecke algebras (and affine quantum GL_n) at roots of unity*, IMRN **5** (1994), 215–217.

[J] G. James, *The decomposition matrices of $GL_n(q)$ for $n \leq 10$*, Proc. London Math. Soc., **60** (1990), 225–264.

[JK] G. James and A. Kerber, *The representation theory of the symmetric group*, **16**, Encyclopedia of Mathematics, Addison–Wesley, Massachusetts (1981).

[JM1] G. James and A. Mathas, *Hecke algebras of type* **A** *at $q = -1$*, J. Algebra, **184** (1996), 102–158.

[JM2] G. James and A. Mathas, *A q–analogue of the Jantzen–Schaper Theorem*, Proc. London Math. Soc. (3), **74**, 1997, 241–274.

[K] A. Kleshchev, *Branching rules for modular representations III*, J. London Math. Soc., **54**, 1996, 25–38.

[LLT] A. Lascoux, B. Leclerc, and J-Y. Thibon, *Hecke algebras at roots of unity and crystal bases of quantum affine algebras*, Comm. Math. Phys., **181** (1996), 205–263.

[LT] B. Leclerc and J-Y. Thibon, *Canonical bases and q–deformed Fock spaces*, Int. Research Notices **9** (1996), 447–456.

[M] A. Mathas, *Canonical bases and the decomposition matrices of Ariki–Koike algebras*, preprint 1996.

1.1 Specht

`Specht(`e`)`
`Specht(`e`, `p`)`
`Specht(`e`, `p`, `*val* `[,`*HeckeRing*`])`

Let R be a field of characteristic 0, q a non–zero element of R, and let e be the smallest positive integer such that

$$1 + q + \ldots + q^{e-1} = 0$$

(we set $e = 0$ if no such integer exists). The record returned by `Specht(`e`)` allows calculations in the Grothendieck rings of the Hecke algebras `H` of type **A** which are defined over R and have parameter q. (The Hecke algebra is described in Chapter 81; see also `Hecke` 81.1.) Below we also describe how to consider Hecke algebras defined over fields of positive characteristic.

`Specht` returns a record which contains, among other things, functions `S` , `P`, and `D` which correspond to the Specht modules, projective indecomposable modules, and the simple modules for the family of Hecke algebras determined by R and q. SPECHT allows manipulation

of arbitrary linear combinations of these "modules", as well as a way of inducing and restricting them, "multiplying" them, and converting between these three natural bases of the Grothendieck ring. Multiplication of modules corresponds to taking a tensor product, and then inducing (thus giving a module for a larger Hecke algebra).

```
gap> RequirePackage("specht"); H:=Specht(5);
Specht(e=5, S(), P(), D(), Pq())
gap> H.D(3,2,1);
D(3,2,1)
gap> H.S( last );
S(6)-S(4,2)+S(3,2,1)
gap> InducedModule(H.P(3,2,1));
P(4,2,1)+P(3,3,1)+P(3,2,2)+2*P(3,2,1,1)
gap> H.S(last);
S(4,2,1)+S(3,3,1)+S(3,2,2)+2*S(3,2,1,1)+S(2,2,2,1)+S(2,2,1,1,1)
gap> H.D(3,1)*H.D(3);
D(7)+2*D(6,1)+D(5,2)+D(5,1,1)+2*D(4,3)+D(4,2,1)+D(3,3,1)
gap> RestrictedModule(last);
4*D(6)+3*D(5,1)+5*D(4,2)+2*D(4,1,1)+2*D(3,3)+2*D(3,2,1)
gap> H.S(last);
S(6)+3*S(5,1)+3*S(4,2)+2*S(4,1,1)+2*S(3,3)+2*S(3,2,1)
gap> H.P(last);
P(6)+3*P(5,1)+2*P(4,2)+2*P(4,1,1)+2*P(3,3)
```

The way in which the partitions indexing the modules are printed can be changed using `SpechtPrettyPrint` 1.59.

There is also a function `Schur` 1.2 for doing calculations with the q–Schur algebra. See `DecompositionMatrix` 1.3, and `CrystalDecompositionMatrix` 1.4.

This function requires the package "specht" (see 56.1).

The functions `H.S`, `H.P`, and `H.D`

The functions `H.S`, `H.P`, and `H.D` return records which correspond to Specht modules, projective indecomposable modules, and simple modules respectively. Each of these three functions can be called in four different ways, as we now describe.

`H.S`(μ) `H.P`(μ) `H.D`(μ)

In the first form, μ is a partition (either a list, or a sequence of integers), and the corresponding Specht module, PIM, or simple module (respectively), is returned.

```
gap> H.P(4,3,2);
P(4,3,2)
```

`H.S`(x) `H.P`(x) `H.D`(x)

Here, x is an H–module. In this form, `H.S` rewrites x as a linear combination of Specht modules, if possible. Similarly, `H.P` and `H.D` rewrite x as a linear combination of PIMs and simple modules respectively. These conversions require knowledge of the relevant decomposition matrix of H; if this is not known then `false` is returned (over fields of characteristic

zero, all of the decomposition matrices are known via the algorithm of [LLT]; various other decomposition matrices are included with SPECHT). For example, `H.S(H.P(`μ`))` returns

$$\sum_{\nu} d_{\nu\mu} \mathtt{S}(\nu),$$

or `false` if some of these decomposition multiplicities are not known.

```
gap> H.D( H.P(4,3,2) );
D(5,3,1)+2*D(4,3,2)+D(2,2,2,2,1)
gap> H.S( H.D( H.S(1,1,1,1,1) ) );
-S(5)+S(4,1)-S(3,1,1)+S(2,1,1,1)
```

As the last example shows, SPECHT does not always behave as expected. The reason for this is that Specht modules indexed by e–singular partitions can always be written as a linear combination of Specht modules which involve only e–regular partitions. As such, it is not always clear when two elements are equal in the Grothendieck ring. Consequently, to test whether two modules are equal you should first rewrite both modules in the `D`–basis; this is *not* done by SPECHT because it would be very inefficient.

`H.S(`d, μ`)` `H.P(`d, μ`)` `H.D(`d, μ`)`

In the third form, d is a decomposition matrix and μ is a partition. This is useful when you are trying to calculate a new decomposition matrix d because it allows you to do calculations using the known entries of d to deduce information about the unknown ones. When used in this way, `H.P` and `H.D` use d to rewrite `P(`μ`)` and `D(`μ`)` respectively as a linear combination of Specht modules, and `H.S` uses d to write `S(`μ`)` as a linear combination of simple modules. If the values of the unknown entries in d are needed, `false` is returned.

```
gap> H:=Specht(3,3);    #  e = 3, p = 3 = characteristic of R
Specht(e=3, p=3, S(), P(), D())
gap>  d:=InducedDecompositionMatrix(DecompositionMatrix(H,14));;
#  Inducing....
The following projectives are missing from <d>:
    [ 15 ]  [ 8, 7 ]
gap> H.P(d,4,3,3,2,2,1);
S(4,3,3,2,2,1)+S(4,3,3,2,1,1,1)+S(4,3,2,2,2,1,1)+S(3,3,3,2,2,1,1)
gap> H.S(d,7, 3, 3, 2);
D(11,2,1,1)+D(10,3,1,1)+D(8,5,1,1)+D(8,3,3,1)+D(7,6,1,1)+D(7,3,3,2)
gap> H.D(d,14,1);
false
```

The final example returned `false` because the partitions `(14,1)` and `(15)` have the same 3–core (and `P(15)` is missing from d).

`H.S(`d, x`)` `H.P(`d, x`)` `H.D(`d, x`)`

In the final form, d is a decomposition matrix and x is a module. All three functions rewrite x in their respective basis using d. Again this is only useful when you are trying to calculate a new decomposition matrix because, for any "known" decomposition matrix d, `H.S(`x`)` and `H.S(`d, x`)` are equivalent (and similarly for `H.P` and `H.D`).

```
gap> H.S(d, H.D(d,10,5) );
```

```
-S(13,2)+S(10,5)
```

Decomposition numbers of the symmetric groups

The last example looked at Hecke algebras with parameter q=1 and R a field of characteristic 3 (so e=3); that is, the group algebra of the symmetric group over a field of characteristic 3. More, generally, the command `Specht(`p`, `p`)` can be used to consider the group algebras of the symmetric groups over fields of characteristic p (i.e. e=p, and `R` a field of characteristic p).

For example, the dimensions of the simple modules of S_6 over fields of characteristic 5 can be computed as follows:

```
gap> H:=Specht(5,5);; SimpleDimension(H,6);
6       : 1
5,1     : 5
4,2     : 8
4,1^2   : 10
3^2     : 5
3,2,1   : 8
3,1^3   : 10
2^3     : 5
2^2,1^2 : 1
2,1^4   : 5
```

Hecke algebras over fields of positive characteristic

To consider Hecke algebras defined over arbitrary fields `Specht` must also be supplied with a **valuation map** *val* as an argument. The function *val* is a map from some PID into the natural numbers; at present it is needed only by functions which rely (at least implicitly), upon the q–analogue of Schaper's theorem. In general, *val* depends upon q and the characteristic of R; full details can be found in [JM2].

Over fields of characteristic zero, and in the symmetric group case, the function *val* is automatically defined by `Specht`. When R is a field of characteristic zero, $val([i]_q)$ is 1 if e divides i and 0 otherwise (this is the valuation map associated to the prime ideal in $\mathbf{C}[v]$ generated by the e–th cyclotomic polynomial). When $q = 1$ and R is a field of characteristic p, *val* is the usual p–adic valuation map.

As another example, if $q = 4$ and R is a field of characteristic 5 (so $e = 2$), then the valuation map sends the integer x to $\nu_5([4]_x)$ where $[4]_x$ is interpreted as an integer and ν_5 is the usual 5–adic valuation. To consider this Hecke algebra one could proceed as follows:

```
gap> val:=function(x) local v;
>       x:=Sum([0..x-1],v->4^v);  # x->[x]_q
>       v:=0; while x mod 5=0 do x:=x/5; v:=v+1; od;
>       return v;
>     end;;
gap> H:=Specht(2,5,val,"e2q4");
```

```
Specht(e=2, p=5, S(), P(), D(), HeckeRing="e2q4")
```

Notice the string "e2q4" which was also passed to `Specht` in this example. Although it is not strictly necessary, it is a good idea when using a "non–standard" valuation map *val* to specify the value of `H.HeckeRing`=*HeckeRing*. This string is used for internal bookkeeping by SPECHT; in particular, it is used to determine filenames when reading and saving decomposition matrices. If a "standard" valuation map is used then *HeckeRing* is set to the string "*e*<*e*>*p*<*p*>"; otherwise it defaults to "unknown". The function `SaveDecompositionMatrix` will not save any decomposition matrix for any Hecke algebra `H` with `H.HeckeRing`="unknown".

The Fock space and Hecke algebras over fields of characteristic zero

For Hecke algebras H defined over fields of characteristic zero Lascoux, Leclerc and Thibon [LLT] have described an easy, inductive, algorithm for calculating the decomposition matrices of H. Their algorithm really calculates the **canonical basis**, or (global) **crystal basis** of the Fock space; results of Grojnowski–Lusztig [Gr] show that computing this basis is equivalent to computing the decomposition matrices of H (see also [A]).

The **Fock space** $\mathcal{F}$ is an (integrable) module for the quantum group $U_q(\widehat{sl}_e)$ of the affine special linear group. $\mathcal{F}$ is a free $\mathbf{C}[\mathtt{v}]$–module with basis the set of all Specht modules $\mathtt{S}(\mu)$ for all partitions μ of all integers

$$\mathcal{F} = \bigoplus_{n \geq 0} \bigoplus_{\mu \vdash n} \mathbf{C}[\mathtt{v}]\ \mathtt{S}(\mu);$$

here `v=H.info.Indeterminate` is an indeterminate over the integers (or strictly, $\mathbf{C}$). The canonical basis elements $\mathtt{Pq}(\mu)$ for the $U_q(\widehat{sl}_e)$–submodule of $\mathcal{F}$ generated by the 0–partition are indexed by e–regular partitions μ. Moreover, under **specialization**, $\mathtt{Pq}(\mu)$ maps to $\mathtt{P}(\mu)$. An eloquent description of the algorithm for computing $\mathtt{H.Pq}(\mu)$ can be found in [LLT].

To access the elements of the Fock space SPECHT provides the functions:

$\mathtt{H.Pq}(\mu)$ $\quad$ $\mathtt{H.Sq}(\mu)$

Notice that, unlike `H.P` and `H.S` the only arguments which `H.Pq` and `H.Sq` accept are partitions. (Given that our indeterminate is `v` these functions should really be called `H.Pv` and `H.Sv`; here "q" stands for "quantum.)

The function `H.Pq` computes the canonical basis element $\mathtt{Pq}(\mu)$ of the Fock space corresponding to the e–regular partition μ (there is a canonical basis — defined using a larger quantum group — for the whole of the Fock space [LT]; conjecturally, this basis can be used to compute the decomposition matrices for the q–Schur algebra over fields of characteristic zero). The second function returns a standard basis element $\mathtt{S}(\mu)$ of $\mathcal{F}$.

```
gap> H:=Specht(4);
Specht(e=4, S(), P(), D(), Pq())
gap> H.Pq(6,2);
S(6,2)+v*S(5,3)
gap> RestrictedModule(last);
S(6,1)+(v + v^(-1))*S(5,2)+v*S(4,3)
```

```
gap> H.P(last);
P(6,1)+(v + v^(-1))*P(5,2)
gap> Specialized(last);
P(6,1)+2*P(5,2)
gap> H.Sq(5,3,2);
S(5,3,2)
gap> InducedModule(last,0);
v^(-1)*S(5,3,3)
```

The modules returned by `H.Pq` and `H.Sq` behave very much like elements of the Grothendieck ring of H; however, they should be considered as elements of the Fock space. The key difference is that when induced or restricted "quantum" analogues of induction and restriction are used. These analogues correspond to the action of $U_q(\widehat{sl}_e)$ on $\mathcal{F}$ [LLT].

In effect, the functions `H.Pq` and `H.Sq` allow computations in the Fock space, using the functions `InducedModule` 1.6 and `RestrictedModule` 1.8. The functions `H.S`, `H.P`, and `H.D` can also be applied to elements of the Fock space, in which case they have the expected effect. In addition, any element of the Fock space can be specialized to give the corresponding element of the Grothendieck ring of `H` (it is because of this correspondence that we do not make a distinction between elements of the Fock space and the Grothendieck ring of `H`).

When working over fields of characteristic zero SPECHT will automatically calculate any canonical basis elements that it needs for computations in the Grothendieck ring of H. If you are not interested in the canonical basis elements you need never work with them directly. If, for some reason, you do not want SPECHT to use the canonical basis elements to calculate decomposition numbers then all you need to do is `Unbind(H.Pq)`.

1.2 Schur

`Schur(`*e*`)`
`Schur(`*e*`, `*p*`)`
`Schur(`*e*`, `*p*`, `*val*` [,`*HeckeRing*`])`

This function behaves almost identically to the function `Specht` (see 1.1), the only difference being that the three functions in the record `S` returned by `Schur` are called `S.W`, `S.P`, and `S.F` and that they correspond to the q-Weyl modules, the projective decomposable modules, and the simple modules of the q–Schur algebra respectively. Note that our labeling of these modules is non–standard, following that used by James in [J]. The standard labeling can be obtained from ours by replacing all partitions by their conjugates.

Almost all of the functions in SPECHT which accept a `Specht` record H will also accept a record S returned by `Schur`

In the current version of SPECHT the decomposition matrices of q–Schur algebras are not fully supported. The `InducedDecompositionMatrix` function can be applied to these matrices; however there are no additional routines available for calculating the columns corresponding to e–singular partitions. The decomposition matrices for the q–Schur algebras defined over a field of characteristic 0 for $n \leq 10$ are in the SPECHT libraries.

```
gap> S:=Schur(2);
Schur(e=2, W(), P(), F(), Pq())
```

```
gap> InducedDecompositionMatrix(DecompositionMatrix(S,3));
#  The following projectives are missing from d
#  [ 2, 2 ]
4      | 1                    #  DecompositionMatrix(S,4) returns the
3,1    | 1 1                  #  full decomposition matrix. The point
2^2    | . 1 .                #  of this example is to emphasize the
2,1^2  | 1 1 . 1              #  current limitations of Schur.
1^4    | 1 . . 1 1
```

Note that when S is defined over a field of characteristic zero then it contains a function `S.Pq` for calculating canonical basis elements (see `Specht` 1.1); currently `S.Pq`(μ) is implemented only for e–regular partitions. There is also a function `H.Wq`.

See also `Specht` 1.1. This function requires the package "specht" (see 56.1).

1.3 DecompositionMatrix

`DecompositionMatrix`(*H*, *n* [,*Ordering*])
`DecompositionMatrix`(*H*, *filename* [,*Ordering*])

The function `DecompositionMatrix` returns the decomposition matrix `D` of `H`(S_n) where `H` is a Hecke algebra record returned by the function `Specht` (or `Schur`). `DecompositionMatrix` first checks to see whether the required decomposition matrix exists as a library file (checking first in the current directory, next in the directory specified by `SpechtDirectory`, and finally in the SPECHT libraries). If `H.Pq` exists, `DecompositionMatrix` next looks for **crystallized decomposition matrices** (see `CrystalDecompositionMatrix` 1.4). If the decomposition matrix `d` is not stored in the library `DecompositionMatrix` will calculate `d` when `H` is a Hecke algebra with a base field `R` of characteristic zero, and will return `false` otherwise (in which case the function `CalculateDecompositionMatrix` 1.15 can be used to force SPECHT to try and calculate this matrix).

For Hecke algebras defined over fields of characteristic zero, SPECHT uses the algorithm of [LLT] to calculate decomposition matrices (this feature can be disabled by unbinding `H.Pq`). The decomposition matrices for the q–Schur algebras for $n \leq 10$ are contained in the SPECHT library, as are those for the symmetric group over fields of positive characteristic when $n < 15$.

Once a decomposition matrix is known, SPECHT keeps an internal copy of it which is used by the functions `H.S`, `H.P`, and `H.D`; these functions also read decomposition matrix files as needed.

If you set the variable `SpechtDirectory`, then SPECHT will also search for decomposition matrix files in this directory. The files in the current directory override those in `SpechtDirectory` and those in the SPECHT libraries.

In the second form of the function, when a *filename* is supplied, `DecompositionMatrix` will read the decomposition matrix in the file *filename*, and this matrix will become SPECHT's internal copy of this matrix.

By default, the rows and columns of the decomposition matrices are ordered lexicographically. This can be changed by supplying `DecompositionMatrix` with an ordering function such as `LengthLexicographic` or `ReverseDominance`. You do not need to specify the ordering you want every time you call `DecompositionMatrix`; SPECHT will keep the same

ordering until you change it again. This ordering can also be set "by hand" using the variable `H.Ordering`.

```
gap> DecompositionMatrix(Specht(3),6,LengthLexicographic);
6       | 1
5,1     | 1 1
4,2     | . . 1
3^2     | . 1 . 1
4,1^2   | . 1 . . 1
3,2,1   | 1 1 . 1 1 1
2^3     | 1 . . . . 1
3,1^3   | . . . . 1 1
2^2,1^2 | . . . . . . 1
2,1^4   | . . . 1 . 1 .
1^6     | . . . 1 . . .
```

Once you have a decomposition matrix it is often nice to be able to print it. The on screen version is often good enough; there is also a `TeX` command which generates a LaTeX version. There are also functions for converting SPECHT decomposition matrices into GAP matrices and visa versa (see `MatrixDecompositionMatrix` 1.16 and `DecompositionMatrixMatrix` 1.17).

Using the function `InducedDecompositionMatrix` (see 1.10), it is possible to induce a decomposition matrix. See also `SaveDecompositionMatrix` 1.14 and `IsNewIndecomposable` 1.11, `Specht` 1.1, `Schur` 1.2, and `CrystalDecompositionMatrix` 1.4. This function requires the package "specht" (see 56.1).

1.4 CrystalDecompositionMatrix

`CrystalDecompositionMatrix(`*H*`, `*n*` [,`*Ordering*`])`
`CrystalDecompositionMatrix(`*H*`, `*filename*` [,`*Ordering*`])`

This function is similar to `DecompositionMatrix`, except that it returns a **crystallized decomposition matrix**. The columns of decomposition matrices correspond to projective indecomposables; the columns of crystallized decomposition matrices correspond to the canonical basis elements of the Fock space (see 1.1). Consequently, the entries in these matrices are polynomials (in `v`), and by specializing (i.e. setting `v` equal to 1; see 1.54), the decomposition matrices of *H* are obtained (see 1.1).

Crystallized decomposition matrices are defined only for Hecke algebras over a base field of characteristic zero. Unlike "normal" decomposition matrices, crystallized decomposition matrices cannot be induced.

```
gap> CrystalDecompositionMatrix(Specht(3), 6);
6       |   1
5,1     |   v   1
4,2     |   .   .   1
4,1^2   |   .   v   .   1
3^2     |   .   v   .   .   1
3,2,1   |   v v^2   .   v   v   1
3,1^3   |   .   .   . v^2   .   v
2^3     | v^2   .   .   .   .   v
```

```
2^2,1^2|   .   .   .   .   .   .   1
2,1^4  |   .   .   .   .   v v^2   .
1^6    |   .   .   .   . v^2   .   .
gap> Specialized(last);   # set v equal to 1.
6      | 1
5,1    | 1 1
4,2    | . . 1
4,1^2  | . 1 . 1
3^2    | . 1 . . 1
3,2,1  | 1 1 . 1 1 1
3,1^3  | . . . 1 . 1
2^3    | 1 . . . . 1
2^2,1^2| . . . . . . 1
2,1^4  | . . . . 1 1 .
1^6    | . . . . 1 . .
```

See also `Specht` 1.1, `Schur` 1.2, `DecompositionMatrix` 1.3, and `Specialized` 1.54. This function requires the package "specht" (see 56.1).

1.5 DecompositionNumber

`DecompositionNumber(`H`, `μ`, `ν`)`
`DecompositionNumber(`d`, `μ`, `ν`)`

This function attempts to calculate the decomposition multiplicity of `D`(ν) in `S`(μ) (equivalently, the multiplicity of `S`(μ) in `P`(ν)). If `P`(ν) is known, we just look up the answer; if not `DecompositionNumber` tries to calculate the answer using "row and column removal" (see [J,Theorem 6.18]).

```
gap> H:=Specht(6);;
gap> DecompositionNumber(H,[6,4,2],[6,6]);
0
```

This function requires the package "specht" (see 56.1).

Partitions in SPECHT

Many of the functions in SPECHT take partitions as arguments. Partitions are usually represented by lists in GAP. In SPECHT, all the functions which expect a partition will accept their argument either as a list or simply as a sequence of numbers. So, for example:

```
gap> H:=Specht(4);; H.S(H.P(6,4));
S(6,4)+S(6,3,1)+S(5,3,1,1)+S(3,3,2,1,1)+S(2,2,2,2,2)
gap> H.S(H.P([6,4]));
S(6,4)+S(6,3,1)+S(5,3,1,1)+S(3,3,2,1,1)+S(2,2,2,2,2)
```

Some functions require more than one argument, but the convention still applies.

```
gap> ECore(3, [6,4,2]);
[ 6, 4, 2 ]
gap> ECore(3, 6,4,2);
```

```
[ 6, 4, 2 ]
gap> GoodNodes(3, 6,4,2);
[ false, false, 3 ]
gap> GoodNodes(3, [6,4,2], 2);
3
```

Basically, it never hurts to put the extra brackets in, and they can be omitted so long as this is not ambiguous. One function where the brackets are needed is `DecompositionNumber`; this is clear because the function takes two partitions as its arguments.

Inducing and restricting modules

SPECHT provides four functions `InducedModule`, `RestrictedModule`, `SInducedModule` and `SRestrictedModule` for inducing and restricting modules. All functions can be applied to Specht modules, PIMs, and simple modules. These functions all work by first rewriting all modules as a linear combination of Specht modules (or q–Weyl modules), and then inducing and restricting. Whenever possible the induced or restricted module will be written in the original basis.

All of these functions can also be applied to elements of the Fock space (see 1.1); in which case they correspond to the action of the generators E_i and F_i of $U_q(\widehat{sl_e})$ on $\mathcal{F}$. There is also a function `InducedDecompositionMatrix` 1.10 for inducing decomposition matrices.

1.6 InducedModule

`InducedModule(`x`)`
`InducedModule(`x`, `r_1` [,`r_2`, ...])`

There is an natural embedding of `H`(S_n) in `H`(S_{n+1}) which in the usual way lets us define an **induced** `H`(S_{n+1})–module for every `H`(S_n)–module. The function `InducedModule` returns the induced modules of the Specht modules, principal indecomposable modules, and simple modules (more accurately, their image in the Grothendieck ring).

There is also a function `SInducedModule` (see 1.7) which provides a much faster way of r–inducing s times (and inducing s times).

Let μ be a partition. Then the induced module `InducedModule(S(`μ`))` is easy to describe: it has the same composition factors as $\sum$ `S`(ν) where ν runs over all partitions whose diagrams can be obtained by adding a single node to the diagram of μ.

```
gap> H:=Specht(2,2);
Specht(e=2, p=2, S(), P(), D())
gap> InducedModule(H.S(7,4,3,1));
S(8,4,3,1)+S(7,5,3,1)+S(7,4,4,1)+S(7,4,3,2)+S(7,4,3,1,1)
gap> InducedModule(H.P(5,3,1));
P(6,3,1)+2*P(5,4,1)+P(5,3,2)
gap> InducedModule(H.D(11,2,1));
#  D(x), unable to rewrite x as a sum of simples
```

```
S(12,2,1)+S(11,3,1)+S(11,2,2)+S(11,2,1,1)
```

When inducing indecomposable modules and simple modules, `InducedModule` first rewrites these modules as a linear combination of Specht modules (using known decomposition matrices), and then induces this linear combination of Specht modules. If possible SPECHT then rewrites the induced module back in the original basis. Note that in the last example above, the decomposition matrix for S_{15} is not known by SPECHT; this is why `InducedModule` was unable to rewrite this module in the `D`–basis.

r–Induction

`InducedModule`(x, r_1 [, r_2, ...])

Two Specht modules `S`(μ) and `S`(ν) belong to the same block if and only if the corresponding partitions μ and ν have the same e–core [JM2] (see 1.39). Because the e–core of a partition is determined by its (multiset of) e–residues, if `S`(μ) and `S`(ν) appear in `InducedModule(S(`τ`))`, for some partition τ, then `S`(μ) and `S`(ν) belong to the same block if and only if μ and ν can be obtained by adding a node of the same e–residue to the diagram of τ. The second form of `InducedModule` allows one to induce "within blocks" by only adding nodes of some fixed e–residue r; this is known as **r-induction**. Note that $0 \leq r < e$.

```
gap> H:=Specht(4); InducedModule(H.S(5,2,1));
S(6,2,1)+S(5,3,1)+S(5,2,2)+S(5,2,1,1)
gap> InducedModule(H.S(5,2,1),0);
0*S()
gap> InducedModule(H.S(5,2,1),1);
S(6,2,1)+S(5,3,1)+S(5,2,1,1)
gap> InducedModule(H.S(5,2,1),2);
0*S()
gap> InducedModule(H.S(5,2,1),3);
S(5,2,2)
```

The function `EResidueDiagram` (1.35), prints the diagram of μ, labeling each node with its e–residue. A quick check of this diagram confirms the answers above.

```
gap> EResidueDiagram(H,5,2,1);
  0  1  2  3  0
  3  0
  2
```

"Quantized" induction

When `InducedModule` is applied to the canonical basis elements `H.Pq`(μ) (or more generally elements of the Fock space; see 1.1), a "quantum analogue" of induction is applied. More precisely, the function `InducedModule(*,i)` corresponds to the action of the generator F_i of the quantum group $U_q(\widehat{sl_e})$ on $\mathcal{F}$ [LLT].

```
gap> H:=Specht(3);; InducedModule(H.Pq(4,2),1,2);
S(6,2)+v*S(4,4)+v^2*S(4,2,2)
gap> H.P(last);
P(6,2)
```

See also `SInducedModule` 1.7, `RestrictedModule` 1.8, and `SRestrictedModule` 1.9. This function requires the package "specht" (see 56.1).

1.7 SInducedModule

`SInducedModule(`x`, `s`)`
`SInducedModule(`x`, `s`, `r`)`

The function `SInducedModule`, standing for "string induction", provides a more efficient way of r–inducing s times (and a way of inducing s times if the residue r is omitted); r–induction is explained in 1.6.

```
gap> H:=Specht(4);; SInducedModule(H.P(5,2,1),3);
P(8,2,1)+3*P(7,3,1)+2*P(7,2,2)+6*P(6,3,2)+6*P(6,3,1,1)+3*P(6,2,1,1,1)
+2*P(5,3,3)+P(5,2,2,1,1)
gap> SInducedModule(H.P(5,2,1),3,1);
P(6,3,1,1)
gap> InducedModule(H.P(5,2,1),1,1,1);
6*P(6,3,1,1)
```

Note that the multiplicity of each summand of `InducedModule(x,r,...,r)` is divisible by $s!$ and that `SInducedModule` divides by this constant.

As with `InducedModule` this function can also be applied to elements of the Fock space (see 1.1), in which case the quantum analogue of induction is used.

See also `InducedModule` 1.6. This function requires the package "specht" (see 56.1).

1.8 RestrictedModule

`RestrictedModule(`x`)`
`RestrictedModule(`x`, `r_1` [, `r_2`, ...])`

Given a module x for `H`(S_n) `RestrictedModule` returns the corresponding module for `H`(S_{n-1}). The restriction of the Specht module `S`(μ) is the linear combination of Specht modules $\sum$ `S`(ν) where ν runs over the partitions whose diagrams are obtained by deleting a node from the diagram of μ. If only nodes of residue r are deleted then this corresponds to first restricting `S`(μ) and then taking one of the block components of the restriction; this process is known as **r-restriction** (cf. r–induction in 1.6).

There is also a function `SRestrictedModule` (see 1.9) which provides a faster way of r–restricting s times (and restricting s times).

When more than one residue if given to `RestrictedModule` it returns

`RestrictedModule`$(x, r_1, r_2, ..., r_k)$ = `RestrictedModule(RestrictedModule`$(x, r_1), r_2, ..., r_k)$

(cf. `InducedModule` 1.6).

```
gap> H:=Specht(6);; RestrictedModule(H.P(5,3,2,1),4);
2*P(4,3,2,1)
gap> RestrictedModule(H.D(5,3,2),1);
D(5,2,2)
```

"Quantized" restriction

As with `InducedModule`, when `RestrictedModule` is applied to the canonical basis elements `H.Pq`(μ) a quantum analogue of restriction is applied; this time, `RestrictedModule(*,i)` corresponds to the action of the generator E_i of $U_q(\widehat{sl_e})$ on $\mathcal{F}$ [LLT].

See also `InducedModule` 1.6, `SInducedModule` 1.7, and `SRestrictedModule` 1.9. This function requires the package "specht" (see 56.1).

1.9 SRestrictedModule

`SRestrictedModule(`x`, `s`)`
`SRestrictedModule(`x`, `s`, `r`)`

As with `SInducedModule` this function provides a more efficient way of r–restricting s times, or restricting s times if the residue r is omitted (cf. `SInducedModule` 1.7).

```
gap> H:=Specht(6);; SRestrictedModule(H.S(4,3,2),3);
3*S(4,2)+2*S(4,1,1)+3*S(3,3)+6*S(3,2,1)+2*S(2,2,2)
gap> SRestrictedModule(H.P(5,4,1),2,4);
P(4,4)
```

See also `InducedModule` 1.6, `SInducedModule` 1.7, and `RestrictedModule` 1.8. This function requires the package "specht" (see 56.1).

Operations on decomposition matrices

SPECHT is a package for computing decomposition matrices; this section describes the functions available for accessing these matrices directly. In addition to decomposition matrices, SPECHT also calculates the "crystallized decomposition matrices" of [LLT], and the "adjustment matrices" introduced by James [J] (and Geck [G]).

Throughout SPECHT we place an emphasis on calculating the projective indecomposable modules, and hence upon the columns of decomposition matrices. This approach seems more efficient than the traditional approach of calculating decomposition matrices by rows; ideally both approaches should be combined (as is done by `IsNewIndecomposable`).

In principle, all decomposition matrices for all Hecke algebras defined over a field of characteristic zero are available from within SPECHT. In addition, the decomposition matrices for all q–Schur algebras with $n \leq 10$ and all values of e and the p–modular decomposition matrices of the symmetric groups S_n for $n < 15$ are in the SPECHT library files.

If you are using SPECHT regularly to do calculations involving certain values of e it would be advantageous to have SPECHT calculate and save the first 20 odd decomposition matrices that you are interested in. So, for $e = 4$ use the commands:

```
gap> H:=Specht(4);; for n in [8..20] do
>      SaveDecompositionMatrix(DecompositionMatrix(H,n));
>    od;
```

Alternatively, you could save the crystallized decomposition matrices. Note that for $n < 2e$ the decomposition matrices are known (by SPECHT) and easy to compute.

1.10 InducedDecompositionMatrix

`InducedDecompositionMatrix(`d`)`

If d is the decomposition matrix of `H`(S_n), then `InducedDecompositionMatrix(`d`)` attempts to calculate the decomposition matrix of `H`(S_{n+1}). It does this by extracting each projective indecomposable from d and inducing these modules to obtain projective modules for `H`(S_{n+1}). `InducedDecompositionMatrix` then tries to decompose these projectives using the function `IsNewIndecomposable` (see 1.11). In general there will be columns of the decomposition matrix which `InducedDecompositionMatrix` is unable to decompose and these will have to be calculated "by hand". `InducedDecompositionMatrix` prints a list of those columns of the decomposition matrix which it is unable to calculate (this list is also printed by the function `MissingIndecomposables(`d`)`).

```
gap> gap> d:=DecompositionMatrix(Specht(3,3),14);;
gap> InducedDecompositionMatrix(d);;
#  Inducing....
The following projectives are missing from <d>:
    [ 15 ]  [ 8, 7 ]
```

Note that the missing indecomposables come in "pairs" which map to each other under the Mullineux map (see `MullineuxMap` 1.25).

Almost all of the decomposition matrices included in SPECHT were calculated directly by `InducedDecompositionMatrix`. When n is "small" `InducedDecompositionMatrix` is usually able to return the full decomposition matrix for `H`(S_{n+1}).

Finally, although the `InducedDecompositionMatrix` can also be applied to the decomposition matrices of the q–Schur algebras (see `Schur` 1.2), `InducedDecompositionMatrix` is much less successful in inducing these decomposition matrices because it contains no special routines for dealing with the indecomposable modules of the q–Schur algebra which are indexed by e–singular partitions. Note also that we use a non–standard labeling of the decomposition matrices of q–Schur algebras; see 1.2.

1.11 IsNewIndecomposable

`IsNewIndecomposable(`d`, `x` [,`μ`])`

`IsNewIndecomposable` is the function which does all of the hard work when the function `InducedDecompositionMatrix` is applied to decomposition matrices (see 1.10). Given a projective module x, `IsNewIndecomposable` returns `true` if it is able to show that x is indecomposable (and this indecomposable is not already listed in d), and `false` otherwise. `IsNewIndecomposable` will also print a brief description of its findings, giving an upper and lower bound on the **first** decomposition number μ for which it is unable to determine the multiplicity of `S`(μ) in x.

`IsNewIndecomposable` works by running through all of the partitions ν such that `P`(ν) could be a summand of x and it uses various results, such as the q-Schaper theorem of [JM2] (see `Schaper` 1.23), the Mullineux map (see `MullineuxMap` 1.25), and inducing simple modules, to determine if `P`(ν) does indeed split off. In addition, if d is the decomposition matrix for `H`(S_n) then `IsNewIndecomposable` will probably use some of the decomposition matrices of `H`(S_m) for $m \leq n$, if they are known. Consequently it is a good idea to save decomposition matrices as they are calculated (see 1.14).

For example, in calculating the 2–modular decomposition matrices of S_r the first projective which `InducedDecompositionMatrix` is unable to calculate is `P`(10).

```
gap> H:=Specht(2,2);;
gap> d:=InducedDecompositionMatrix(DecompositionMatrix(H,9));;
#  Inducing.
#  The following projectives are missing from d
#  [ 10 ]
```

(In fact, given the above commands, SPECHT will return the full decomposition matrix for S_{10} because this matrix is in the library; these were the commands that I used to calculate the decomposition matrix in the library.)

By inducing `P`(9) we can find a projective H–module which contains `P`(10). We can then use `IsNewIndecomposable` to try and decompose this induced module into a sum of PIMs.

```
gap> SpechtPrettyPrint(); x:=InducedModule(H.P(9),1);
S(10)+S(9,1)+S(8,2)+2S(8,1^2)+S(7,3)+2S(7,1^3)+3S(6,3,1)+3S(6,2^2)
+4S(6,2,1^2)+2S(6,1^4)+4S(5,3,2)+5S(5,3,1^2)+5S(5,2^2,1)+2S(5,1^5)
+2S(4^2,2)+2S(4^2,1^2)+2S(4,3^2)+5S(4,3,1^3)+2S(4,2^3)+5S(4,2^2,1^2)
+4S(4,2,1^4)+2S(4,1^6)+2S(3^3,1)+2S(3^2,2^2)+4S(3^2,2,1^2)
+3S(3^2,1^4)+3S(3,2^2,1^3)+2S(3,1^7)+S(2^3,1^4)+S(2^2,1^6)+S(2,1^8)
+S(1^10)
gap> IsNewIndecomposable(d,x);
#  The multiplicity of S(6,3,1) in P(10) is at least 1 and at most 2.
false
gap> x;
S(10)+S(9,1)+S(8,2)+2S(8,1^2)+S(7,3)+2S(7,1^3)+2S(6,3,1)+2S(6,2^2)
+3S(6,2,1^2)+2S(6,1^4)+3S(5,3,2)+4S(5,3,1^2)+4S(5,2^2,1)+2S(5,1^5)
+2S(4^2,2)+2S(4^2,1^2)+2S(4,3^2)+4S(4,3,1^3)+2S(4,2^3)+4S(4,2^2,1^2)
+3S(4,2,1^4)+2S(4,1^6)+2S(3^3,1)+2S(3^2,2^2)+3S(3^2,2,1^2)
+2S(3^2,1^4)+2S(3,2^2,1^3)+2S(3,1^7)+S(2^3,1^4)+S(2^2,1^6)+S(2,1^8)
+S(1^10)
```

Notice that some of the coefficients of the Specht modules in x have changed; this is because `IsNewIndecomposable` was able to determine that the multiplicity of `S`(6,3,1) was at most 2 and so it subtracted one copy of `P`(6,3,1) from x.

In this case, the multiplicity of `S`(6,3,1) in `P`(10) is easy to resolve because general theory says that this multiplicity must be odd. Therefore, $x - $ `P`$(6,3,1)$ is projective. After subtracting `P`(6,3,1) from x we again use `IsNewIndecomposable` to see if x is now indecomposable. We can tell `IsNewIndecomposable` that all of the multiplicities up to and including `S`(6,3,1) have already been checked by giving it the addition argument μ=[6,3,1].

```
gap> x:=x-H.P(d,6,3,1);; IsNewIndecomposable(d,x,6,3,1);
true
```

Consequently, $x =$ `P`(10) and we add it to the decomposition matrix d (and save it).

```
gap> AddIndecomposable(d,x); SaveDecompositionMatrix(d);
```

A full description of what `IsNewIndecomposable` does can be found by reading the comments in `specht.g`. Any suggestions or improvements on this function would be especially welcome.

See also `DecompositionMatrix` 1.3 and `InducedDecompositionMatrix` 1.10. This function requires the package "specht" (see 56.1).

1.12 InvertDecompositionMatrix

`InvertDecompositionMatrix(d)`

Returns the inverse of the (e–regular part of) d, where d is a decomposition matrix, or crystallized decomposition matrix, of a Hecke algebra or q–Schur algebra. If part of the decomposition matrix d is unknown then `InvertDecompositionMatrix` will invert as much of d as possible.

```
gap> H:=Specht(4);; d:=CrystalDecompositionMatrix(H,5);;
gap> InvertDecompositionMatrix(d);
5     |   1
4,1   |   .   1
3,2   |  -v   .   1
3,1^2|   .   .   .   1
2^2,1| v^2   .  -v   .   1
2,1^3|   .   .   .   .   .   1
```

See also `DecompositionMatrix` 1.3, and `CrystalDecompositionMatrix` 1.4. This function requires the package "specht" (see 56.1).

1.13 AdjustmentMatrix

`AdjustmentMatrix(`*dp*`,` *d*`)`

James [J] noticed, and Geck [G] proved, that the decomposition matrices dp for Hecke algebras defined over fields of positive characteristic admit a factorization

$$dp = d * a$$

where d is a decomposition matrix for a suitable Hecke algebra defined over a field of characteristic zero, and a is the so–called **adjustment matrix**. This function returns the adjustment matrix a.

```
gap> H:=Specht(2);; Hp:=Specht(2,2);;
gap> d:=DecompositionMatrix(H,13);; dp:=DecompositionMatrix(Hp,13);;
gap> a:=AdjustmentMatrix(dp,d);
13      | 1
12,1    | . 1
11,2    | 1 . 1
10,3    | . . . 1
10,2,1  | . . . . 1
9,4     | 1 . 1 . . 1
9,3,1   | 2 . . . . . 1
8,5     | . 1 . . . . . 1
8,4,1   | 1 . . . . . . . 1
8,3,2   | . 2 . . . . . 1 . 1
7,6     | 1 . . . . 1 . . . . 1
7,5,1   | . . . . . . 1 . . . . 1
7,4,2   | 1 . 1 . . 1 . . . . 1 . 1
7,3,2,1| . . . . . . . . . . . . . 1
```

```
6,5,2  | . 1 . . . . . 1 . 1 . . . . 1
6,4,3  | 2 . . . 1 . . . . . . . . . . 1
6,4,2,1| . 2 . 1 . . . . . . . . . . . . 1
5,4,3,1| 4 . 2 . . . . . . . . . . . . . . 1
gap> MatrixDecompositionMatrix(dp)=
>           MatrixDecompositionMatrix(d)*MatrixDecompositionMatrix(a);
true
```

In the last line we have checked our calculation.

See also `DecompositionMatrix` 1.3, and `CrystalDecompositionMatrix` 1.4. This function requires the package "specht" (see 56.1).

1.14 SaveDecompositionMatrix

`SaveDecompositionMatrix(`d`)`
`SaveDecompositionMatrix(`d`,` *filename*`)`

The function `SaveDecompositionMatrix` saves the decomposition matrix d. After a decomposition matrix has been saved, the functions `H.S`, `H.P`, and `H.D` will automatically access it as needed. So, for example, before saving d in order to retrieve the indecomposable `P(`μ`)` from d it is necessary to type `H.P(`d`,` μ`)`; once d has been saved, the command `H.P(`μ`)` suffices.

Since `InducedDecompositionMatrix(`d`)` consults the decomposition matrices for smaller n, if they are available, it is advantageous to save decomposition matrices as they are calculated. For example, over a field of characteristic 5, the decomposition matrices for the symmetric groups S_n with $n \leq 20$ can be calculated as follows:

```
gap> H:=Specht(5,5);;
gap> d:=DecompositionMatrix(H,9);;
gap> for r in [10..20] do
>      d:=InducedDecompositionMatrix(d);
>      SaveDecompositionMatrix(d);
>    od;
```

If your Hecke algebra record `H` is defined using a non–standard valuation map (see 1.1) then it is also necessary to set the string "`H.HeckeRing`", or to supply the function with a *filename* before it will save your matrix. `SaveDecompositionMatrix` will also save adjustment matrices and the various other matrices that appear in SPECHT (they can be read back in using `DecompositionMatrix`). Each matrix has a default filename which you can over ride by supplying a *filename*. Using non–standard file names will stop SPECHT from automatically accessing these matrices in future.

See also 1.3 `DecompositionMatrix` 1.3 and `CrystalDecompositionMatrix` 1.4. This function requires the package "specht" (see 56.1).

1.15 CalculateDecompositionMatrix

`CalculateDecompositionMatrix(H,n)`

`CalculateDecompositionMatrix(H,n)` is similar to the function `DecompositionMatrix` 1.3 in that both functions try to return the decomposition matrix `d` of `H(`S_n`)`; the difference is

that this function tries to calculate this matrix whereas the later reads the matrix from the library files (in characteristic zero both functions apply the algorithm of [LLT] to compute `d`). In effect this function is only needed when working with Hecke algebras defined over fields of positive characteristic (or when you wish to avoid the libraries).

For example, if you want to do calculations with the decomposition matrix of the symmetric group S_{15} over a field of characteristic two, `DecompositionMatrix` returns false whereas `CalculateDecompositionMatrix`; returns a part of the decomposition matrix.

```
gap> H:=Specht(2,2);
Specht(e=2, p=2, S(), P(), D())
gap> d:=DecompositionMatrix(H,15);
#  This decomposition matrix is not known; use CalculateDecompositionMatrix()
#  or InducedDecompositionMatrix() to calculate with this matrix.
false
gap> d:=CalculateDecompositionMatrix(H,15);;
#  Projective indecomposable P(6,4,3,2) not known.
#  Projective indecomposable P(6,5,3,1) not known.
...
gap> MissingIndecomposables(d);
The following projectives are missing from <d>:
    [ 15 ]  [ 14, 1 ]  [ 13, 2 ]  [ 12, 3 ]  [ 12, 2, 1 ]  [ 11, 4 ]
[ 11, 3, 1 ]  [ 10, 5 ]  [ 10, 4, 1 ]  [ 10, 3, 2 ]  [ 9, 6 ]  [ 9, 5, 1 ]
[ 9, 4, 2 ]  [ 9, 3, 2, 1 ]  [ 8, 7 ]  [ 8, 6, 1 ]  [ 8, 5, 2 ]  [ 8, 4, 3]
[ 8, 4, 2, 1 ]  [ 7, 6, 2 ]  [ 7, 5, 3 ]  [ 7, 5, 2, 1 ]  [ 7, 4, 3, 1 ]
[ 6, 5, 4 ]  [ 6, 5, 3, 1 ]  [ 6, 4, 3, 2 ]
```

Actually, you are much better starting with the decomposition matrix of S_{14} and then applying `InducedDecompositionMatrix` to this matrix.

See also 1.3 `DecompositionMatrix`. This function requires the package "specht" (see 56.1).

1.16 MatrixDecompositionMatrix

`MatrixDecompositionMatrix(`*d*`)`

Returns the GAP matrix corresponding to the SPECHT decomposition matrix *d*. The rows and columns of *d* are ordered by `H.Ordering`.

```
gap> MatrixDecompositionMatrix(DecompositionMatrix(Specht(3),5));
[ [ 1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 1, 1, 0, 0 ],
  [ 0, 0, 0, 1, 0 ], [ 1, 0, 0, 0, 1 ], [ 0, 0, 0, 0, 1 ],
  [ 0, 0, 1, 0, 0 ] ]
```

See also `DecompositionMatrix` 1.3 and `DecompositionMatrixMatrix` 1.17. This function requires the package "specht" (see 56.1).

1.17 DecompositionMatrixMatrix

`DecompositionMatrixMatrix(`*H*`, `*m*`, `*n*`)`

Given a Hecke algebra *H*, a GAP matrix *m*, and an integer *n* this function returns the SPECHT decomposition matrix corresponding to *m*. If `p` is the number of partitions of *n*

and `r` the number of e–regular partitions of n, then m must be either $r \times r$, $p \times r$, or $p \times p$. The rows and columns of m are assumed to be indexed by partitions ordered by `H.Ordering` (see 1.1).

```
gap> H:=Specht(3);;
gap> m:=[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 1, 0 ],
>         [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ] ];;
gap> DecompositionMatrixMatrix(H,m,4);
4    | 1
3,1  | . 1
2^2  | 1 . 1
2,1^2| . . . 1
1^4  | . . 1 .
```

See also `DecompositionMatrix` 1.3 and `MatrixDecompositionMatrix` 1.16. This function requires the package "specht" (see 56.1).

1.18 AddIndecomposable

`AddIndecomposable(`d`, `x`)`

`AddIndecomposable(`d`, `x`)` inserts the indecomposable module x into the decomposition matrix d. If d already contains the indecomposable x then a warning is printed. The function `AddIndecomposable` also calculates `MullineuxMap(`x`)` (see 1.25) and adds this indecomposable to d (or checks to see that it agrees with the corresponding entry of d if this indecomposable is already by d).

See `IsNewIndecomposable` 1.11 for an example. See also `DecompositionMatrix` 1.3 and `CrystalDecompositionMatrix` 1.4. This function requires the package "specht" (see 56.1).

1.19 RemoveIndecomposable

`RemoveIndecomposable(`d`, `μ`)`

The function `RemoveIndecomposable` removes the column from d which corresponds to `P(`μ`)`. This is sometimes useful when trying to calculate a new decomposition matrix using SPECHT and want to test a possible candidate for a yet to be identified PIM.

See also `DecompositionMatrix` 1.3 and `CrystalDecompositionMatrix` 1.4. This function requires the package "specht" (see 56.1).

1.20 MissingIndecomposables

`MissingIndecomposables(`d`)`

The function `MissingIndecomposables` prints the list of partitions corresponding to the indecomposable modules which are not listed in d.

See also `DecompositionMatrix` 1.3 and `CrystalDecompositionMatrix` 1.4. This function requires the package "specht" (see 56.1).

Calculating dimensions

SPECHT has two functions for calculating the dimensions of modules of Hecke algebras; `SimpleDimension` and `SpechtDimension`. As yet, SPECHT does not know how to calculate the dimensions of modules for q–Schur algebras (these depend upon q).

1.21 SimpleDimension

`SimpleDimension(`d`)`
`SimpleDimension(`H`, `n`)`
`SimpleDimension(`H` | `d`, `μ`)`

In the first two forms, `SimpleDimension` prints the dimensions of all of the simple modules specified by d or for the Hecke algebra `H`(S_n) respectively. If a partition μ is supplied, as in the last form, then the dimension of the simple module `D`(μ) is returned. At present the function is not implemented for the simple modules of the q–Schur algebras.

```
gap> H:=Specht(6);;
gap> SimpleDimension(H,11,3);
272
gap> d:=DecompositionMatrix(H,5);; SimpleDimension(d,3,2);
5
gap> SimpleDimension(d);
5     : 1
4,1   : 4
3,2   : 5
3,1^2 : 6
2^2,1 : 5
2,1^3 : 4
1^5   : 1
```

This function requires the package "specht" (see 56.1).

1.22 SpechtDimension

`SpechtDimension(`μ`)`

Calculates the dimension of the Specht module `S`(μ), which is equal to the number of standard μ-tableaux; the answer is given by the hook length formula (see [JK]).

```
gap> SpechtDimension(6,3,2,1);
5632
```

See also `SimpleDimension` 1.21. This function requires the package "specht"(see 56.1).

Combinatorics on Young diagrams

These functions range from the representation theoretic q–Schaper theorem and Kleshchev's algorithm for the Mullineux map through to simple combinatorial operations like adding and removing rim hooks from Young diagrams.

1.23 Schaper

`Schaper(`H`, `μ`)`

Given a partition μ, and a Hecke algebra H, `Schaper` returns a linear combination of Specht modules which have the same composition factors as the sum of the modules in the "Jantzen filtration" of `S(`μ`)`; see [JM2]. In particular, if ν strictly dominates μ then `D(`ν`)` is a composition factor of `S(`μ`)` if and only if it is a composition factor of `Schaper(`μ`)`.

`Schaper` uses the valuation map `H.valuation` attached to H (see 1.1 and [JM2]).

One way in which the q–Schaper theorem can be applied is as follows. Suppose that we have a projective module x, written as a linear combination of Specht modules, and suppose that we are trying to decide whether the projective indecomposable `P(`μ`)` is a direct summand of x. Then, providing that we know that `P(`ν`)` is not a summand of x for all (e–regular) partitions ν which strictly dominate μ (see 1.50), `P(`μ`)` is a summand of x if and only if `InnerProduct(Schaper(H,`μ`),x)` is non–zero (note, in particular, that we don't need to know the indecomposable `P(`μ`)` in order to perform this calculation).

The q–Schaper theorem can also be used to check for irreduciblity; in fact, this is the basis for the criterion employed by `IsSimpleModule`.

```
gap> H:=Specht(2);;
gap> Schaper(H,9,5,3,2,1);
S(17,2,1)-S(15,2,1,1,1)+S(13,2,2,2,1)-S(11,3,3,2,1)+S(10,4,3,2,1)-S(9,8,3)
-S(9,8,1,1,1)+S(9,6,3,2)+S(9,6,3,1,1)+S(9,6,2,2,1)
gap> Schaper(H,9,6,5,2);
0*S(0)
```

The last calculation shows that `S(`9,6,5,2`)` is irreducible when R is a field of characteristic 0 and e=2 (cf. `IsSimpleModule(H,9,6,5,2)`).

This function requires the package "specht" (see 56.1).

1.24 IsSimpleModule

`IsSimpleModule(`H`, `μ`)`

μ an e–regular partition.

Given an e–regular partition μ, `IsSimpleModule(`H`, `μ`)` returns `true` if `S(`μ`)` is simple and `false` otherwise. This calculation uses the valuation function `H.valuation`; see 1.1. Note that the criterion used by `IsSimpleModule` is completely combinatorial; it is derived from the q–Schaper theorem [JM2].

```
gap> H:=Specht(3);;
gap> IsSimpleModule(H,45,31,24);
false
```

See also `Schaper` 1.23. This function requires the package "specht" (see 56.1).

1.25 MullineuxMap

`MullineuxMap`(e | H , μ)
`MullineuxMap`(d, μ)
`MullineuxMap`(x)

Given an integer e, or a SPECHT record H, and a partition μ, `MullineuxMap`(e, μ) returns the image of μ under the Mullineux map; which we now explain.

The sign representation `D`(1^n) of the Hecke algebra is the (one dimensional) representation sending T_w to $(-1)^{\ell(w)}$. The Hecke algebra H is not a Hopf algebra so there is no well defined action of H upon the tensor product of two H–modules; however, there is an outer automorphism # of H which corresponds to tensoring with `D`(1^n). This sends an irreducible module `D`(μ) to an irreducible `D`$(\mu)^{\#} \cong$ `D`($\mu^{\#}$) for some e–regular partition $\mu^{\#}$. In the symmetric group case, Mullineux gave a conjectural algorithm for calculating $\mu^{\#}$; consequently the map sending μ to $\mu^{\#}$ is known as the **Mullineux map**.

Deep results of Kleshchev [K] for the symmetric group give another (proven) algorithm for calculating the partition $\mu^{\#}$ (Ford and Kleshchev have deduced Mullineux's conjecture from this). Using the canonical basis, it was shown by [LLT] that the natural generalization of Kleshchev's algorithm to H gives the Mullineux map for Hecke algebras over fields of characteristic zero. The general case follows from this, so the Mullineux map is now known for all Hecke algebras.

Kleshchev's map is easy to describe; he proved that if *gns* is any good node sequence for μ, then the sequence obtained from *gns* by replacing each residue r by $-r \bmod e$ is a good node sequence for $\mu^{\#}$ (see `GoodNodeSequence` 1.30).

```
gap> MullineuxMap(Specht(2),12,5,2);
[ 12, 5, 2 ]
gap> MullineuxMap(Specht(4),12,5,2);
[ 4, 4, 4, 2, 2, 1, 1, 1 ]
gap> MullineuxMap(Specht(6),12,5,2);
[ 4, 3, 2, 2, 2, 2, 2, 1, 1 ]
gap> MullineuxMap(Specht(8),12,5,2);
[ 3, 3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1 ]
gap> MullineuxMap(Specht(10),12,5,2);
[ 3, 3, 3, 3, 2, 1, 1, 1, 1, 1 ]
```

`MullineuxMap`(d, μ)

The Mullineux map can also be calculated using a decomposition matrix. To see this recall that "tensoring" a Specht module `S`(μ) with the sign representation yields a module isomorphic to the dual of `S`(λ), where λ is the partition conjugate to μ. It follows that $d_{\mu\nu} = d_{\lambda\nu^{\#}}$ for all e–regular partitions ν. Therefore, if μ is the last partition in the lexicographic order such that $d_{\mu\nu} \neq 0$ then we must have $\nu^{\#} = \lambda$. The second form of `MullineuxMap` uses d to calculate $\mu^{\#}$ rather than the Kleshchev–[LLT] result.

`MullineuxMap`(x)

In the third form, x is a module, and `MullineuxMap` returns $x^{\#}$, the image of x under #. Note that the above remarks show that `P`(μ) is mapped to `P`($\mu^{\#}$) via the Mullineux map; this observation is useful when calculating decomposition matrices (and is used by the function `InducedDecompositionMatrix`).

See also GoodNodes 1.28 and GoodNodeSequence 1.30 . This function requires the package "specht" (see 56.1).

1.26 MullineuxSymbol

MullineuxSymbol(e|H, μ)

Returns the Mullineux symbol of the e–regular partition μ.

```
gap> MullineuxSymbol(5,[8,6,5,5]);
[ [ 10, 6, 5, 3 ], [ 4, 4, 3, 2 ] ]
```

See also PartitionMullineuxSymbol 1.27. This function requires the package "specht" (see 56.1).

1.27 PartitionMullineuxSymbol

PartitionMullineuxSymbol(e|H, ms)

Given a Mullineux symbol ms, this function returns the corresponding e–regular partition.

```
gap> PartitionMullineuxSymbol(5, MullineuxSymbol(5,[8,6,5,5]) );
[ 8, 6, 5, 5 ]
```

See also MullineuxSymbol 1.26. This function requires the package "specht" (see 56.1).

1.28 GoodNodes

GoodNodes(e|H, μ)
GoodNodes(e|H, μ, r)

Given a partition and an integer e, Kleshchev [K] defined the notion of **good node** for each residue r ($0 \leq r < e$). When e is prime and μ is e–regular, Kleshchev showed that the good nodes describe the restriction of the socle of D(μ) in the symmetric group case. Brundan [B] has recently generalized this result to the Hecke algebra.

By definition, there is at most one good node for each residue r, and this node is a removable node (in the diagram of μ). The function GoodNodes returns a list of the rows of μ which end in a good node; the good node of residue r (if it exists) is the $(r+1)$–st element in this list. In the second form, the number of the row which ends with the good node of residue r is returned; or false if there is no good node of residue r.

```
gap> GoodNodes(5,[5,4,3,2]);
[ false, false, 2, false, 1 ]
gap> GoodNodes(5,[5,4,3,2],0);
false
gap> GoodNodes(5,[5,4,3,2],4);
1
```

The good nodes also determine the Kleshchev–Mullineux map (see GoodNodeSequence 1.30 and MullineuxMap 1.25). This function requires the package "specht" (see 56.1).

1.29 NormalNodes

`NormalNodes(`e`|`H`, `μ`)`
`NormalNodes(`e`|`H`, `μ`, `r`)`

Returns the numbers of the rows of μ which end in one of Kleshchev's [K] normal nodes. In the second form, only those rows corresponding to normal nodes of the specified residue are returned.

```
gap> NormalNodes(5,[6,5,4,4,3,2,1,1,1]);
[ [ 1, 4 ], [  ], [  ], [ 2, 5 ], [  ] ]
gap> NormalNodes(5,[6,5,4,4,3,2,1,1,1],0);
[ 1, 4 ]
```

See also `GoodNodes` 1.28. This function requires the package "specht" (see 56.1).

1.30 GoodNodeSequence

`GoodNodeSequence(`e`|`H`, `μ`)`
`GoodNodeSequences(`e`|`H`, `μ`)`

μ an e–regular partition.

Given an e–regular partition μ of n, a **good node sequence** for μ is a sequence gns of n residues such that μ has a good node of residue r, where r is the last residue in gns, and the first $n-1$ residues in gns are a good node sequence for the partition obtained from μ by deleting its (unique) good node with residue r (see `GoodNodes` 1.28). In general, μ will have more than one good node sequence; however, any good node sequence uniquely determines μ (see `PartitionGoodNodeSequence` 1.31).

```
gap> H:=Specht(4);; GoodNodeSequence(H,4,3,1);
[ 0, 3, 1, 0, 2, 2, 1, 3 ]
gap> GoodNodeSequence(H,4,3,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3 ]
gap> GoodNodeSequence(H,4,4,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3, 2 ]
gap> GoodNodeSequence(H,5,4,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3, 2, 0 ]
```

The function `GoodNodeSequences` returns the list of all good node sequences for μ.

```
gap> GoodNodeSequences(H,5,2,1);
[ [ 0, 1, 2, 3, 3, 2, 0, 0 ], [ 0, 3, 1, 2, 2, 3, 0, 0 ],
  [ 0, 1, 3, 2, 2, 3, 0, 0 ], [ 0, 1, 2, 3, 3, 0, 2, 0 ],
  [ 0, 1, 2, 3, 0, 3, 2, 0 ], [ 0, 1, 2, 3, 3, 0, 0, 2 ],
  [ 0, 1, 2, 3, 0, 3, 0, 2 ] ]
```

The good node sequences determine the Mullineux map (see `GoodNodes` 1.28 and `MullineuxMap` 1.25). This function requires the package "specht" (see 56.1).

1.31 PartitionGoodNodeSequence

`PartitionGoodNodeSequence(`e`|`H`, `gns`)`

Given a good node sequence *gns* (see `GoodNodeSequence` 1.30), this function returns the unique e–regular partition corresponding to *gns* (or `false` if in fact *gns* is not a good node sequence).

```
gap> H:=Specht(4);;
gap> PartitionGoodNodeSequence(H,0, 3, 1, 0, 2, 2, 1, 3, 3, 2);
[ 4, 4, 2 ]
```

See also `GoodNodes` 1.28, `GoodNodeSequence` 1.30 and `MullineuxMap` 1.25. This function requires the package "specht" (see 56.1).

1.32 GoodNodeLatticePath

`GoodNodeLatticePath(`e`|`H`, `μ`)`
`GoodNodeLatticePaths(`e`|`H`, `μ`)`
`LatticePathGoodNodeSequence(`e`|`H`, `*gns*`)`

The function `GoodNodeLatticePath` returns a sequence of partitions which give a path in the e–good partition lattice from the empty partition to μ. The second function returns the list of all paths in the e–good partition lattice which end in μ, and the third function returns the path corresponding to a given good node sequence *gns*.

```
gap> GoodNodeLatticePath(3,3,2,1);
[ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 1, 1 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ]
gap> GoodNodeLatticePaths(3,3,2,1);
[ [ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 1, 1 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ],
  [ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 2 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ] ]
gap> GoodNodeSequence(4,6,3,2);
[ 0, 3, 1, 0, 2, 2, 3, 3, 0, 1, 1 ]
gap> LatticePathGoodNodeSequence(4,last);
[ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 2 ], [ 3, 2 ], [ 3, 2, 1 ], [ 4, 2, 1 ],
  [ 4, 2, 2 ], [ 5, 2, 2 ], [ 6, 2, 2 ], [ 6, 3, 2 ] ]
```

See also `GoodNodes` 1.28. This function requires the package "specht" (see 56.1).

1.33 LittlewoodRichardsonRule

`LittlewoodRichardsonRule(`μ`, `ν`)`
`LittlewoodRichardsonCoefficient(`μ`, `ν`, `τ`)`

Given partitions μ of n and ν of m the module $\mathtt{S}(\mu)\otimes\mathtt{S}(\nu)$ is naturally an $\mathtt{H}(S_n\times S_m)$-module and, by inducing, we obtain an $\mathtt{H}(S_{n+m})$-module. This module has the same composition factors as

$$\sum_{\nu} a^{\lambda}_{\mu\nu}\mathtt{S}(\lambda),$$

where the sum runs over all partitions λ of $n+m$ and the integers $a^{\lambda}_{\mu\nu}$ are the Littlewood–Richardson coefficients. The integers $a^{\lambda}_{\mu\nu}$ can be calculated using a straightforward combinatorial algorithm known as the Littlewood–Richardson rule (see [JK]).

The function `LittlewoodRichardsonRule` returns an (unordered) list of partitions of $n+m$ in which each partition λ occurs $a^{\lambda}_{\mu\nu}$ times. The Littlewood-Richardson coefficients are independent of e; they can be read more easily from the computation `S(`μ`)*S(`ν`)`.

```
gap> H:=Specht(0);;      # the generic Hecke algebra with R=C[q]
gap> LittlewoodRichardsonRule([3,2,1],[4,2]);
[ [ 4, 3, 2, 2, 1 ],[ 4, 3, 3, 1, 1 ],[ 4, 3, 3, 2 ],[ 4, 4, 2, 1, 1 ],
 [ 4, 4, 2, 2 ],[ 4, 4, 3, 1 ],[ 5, 2, 2, 2, 1 ],[ 5, 3, 2, 1, 1 ],
  [ 5, 3, 2, 2 ],[ 5, 4, 2, 1 ],[ 5, 3, 2, 1, 1 ],[ 5, 3, 3, 1 ],
  [ 5, 4, 1, 1, 1 ],[ 5, 4, 2, 1 ],[ 5, 5, 1, 1 ],[ 5, 3, 2, 2 ],
  [ 5, 3, 3, 1 ],[ 5, 4, 2, 1 ],[ 5, 4, 3 ],[ 5, 5, 2 ],[ 6, 2, 2, 1, 1],
  [ 6, 3, 1, 1, 1 ],[ 6, 3, 2, 1 ],[ 6, 4, 1, 1 ],[ 6, 2, 2, 2 ],
  [ 6, 3, 2, 1 ],[ 6, 4, 2 ],[ 6, 3, 2, 1 ],[ 6, 3, 3 ],[ 6, 4, 1, 1 ],
  [ 6, 4, 2 ], [ 6, 5, 1 ], [ 7, 2, 2, 1 ], [ 7, 3, 1, 1 ], [ 7, 3, 2 ],
  [ 7, 4, 1 ] ]
gap> H.S(3,2,1)*H.S(4,2);
S(7,4,1)+S(7,3,2)+S(7,3,1,1)+S(7,2,2,1)+S(6,5,1)+2*S(6,4,2)+2*S(6,4,1,1)
+S(6,3,3)+3*S(6,3,2,1)+S(6,3,1,1,1)+S(6,2,2,2)+S(6,2,2,1,1)+S(5,5,2)
+S(5,5,1,1)+S(5,4,3)+3*S(5,4,2,1)+S(5,4,1,1,1)+2*S(5,3,3,1)+2*S(5,3,2,2)
+2*S(5,3,2,1,1)+S(5,2,2,2,1)+S(4,4,3,1)+S(4,4,2,2)+S(4,4,2,1,1)+S(4,3,3,2)
+S(4,3,3,1,1)+S(4,3,2,2,1)
gap> LittlewoodRichardsonCoefficient([3,2,1],[4,2],[5,4,2,1]);
3
```

The function `LittlewoodRichardsonCoefficient` returns a single Littlewood–Richardson coefficient (although you are really better off asking for all of them, since they will all be calculated anyway).

See also `InducedModule` 1.6 and `InverseLittlewoodRichardsonRule` 1.34. This function requires the package "specht" (see 56.1).

1.34 InverseLittlewoodRichardsonRule

`InverseLittlewoodRichardsonRule(`τ`)`

Returns a list of all pairs of partitions $[\mu, \nu]$ such that the Littlewood-Richardson coefficient $a^{\tau}_{\mu\nu}$ is non-zero (see 1.33). The list returned is unordered and $[\mu, \nu]$ will appear $a^{\tau}_{\mu\nu}$ times in it.

```
gap> InverseLittlewoodRichardsonRule([3,2,1]);
[ [ [  ],[ 3, 2, 1 ] ],[ [ 1 ],[ 3, 2 ] ],[ [ 1 ],[ 2, 2, 1 ] ],
  [ [ 1 ],[ 3, 1, 1 ] ],[ [ 1, 1 ],[ 2, 2 ] ],[ [ 1, 1 ],[ 3, 1 ] ],
  [ [ 1, 1 ],[ 2, 1, 1 ] ],[ [ 1, 1, 1 ],[ 2, 1 ] ],[ [ 2 ],[ 2, 2 ] ],
  [ [ 2 ],[ 3, 1 ] ],[ [ 2 ],[ 2, 1, 1 ] ],[ [ 2, 1 ],[ 3 ] ],
  [ [ 2, 1 ],[ 2, 1 ] ],[ [ 2, 1 ],[ 2, 1 ] ],[ [ 2, 1 ],[ 1, 1, 1 ] ],
  [ [ 2, 1, 1 ],[ 2 ] ],[ [ 2, 1, 1 ],[ 1, 1 ] ],[ [ 2, 2 ],[ 2 ] ],
  [ [ 2, 2 ],[ 1, 1 ] ],[ [ 2, 2, 1 ],[ 1 ] ],[ [ 3 ],[ 2, 1 ] ],
  [ [ 3, 1 ],[ 2 ] ],[ [ 3, 1 ],[ 1, 1 ] ],[ [ 3, 1, 1 ],[ 1 ] ],
  [ [ 3, 2 ],[ 1 ] ],[ [ 3, 2, 1 ],[ ] ] ]
```

See also `LittlewoodRichardsonRule` 1.33.

This function requires the package "specht" (see 56.1).

1.35 EResidueDiagram

`EResidueDiagram(`$H \mid e,\ \mu$`)`

`EResidueDiagram(`x`)`

The e–residue of the (i,j)–th node in the diagram of a partition μ is $(j-i) \bmod e$. `EResidueDiagram(`e`, `μ`)` prints the diagram of the partition μ replacing each node with its e-residue.

If x is a module then `EResidueDiagram(`x`)` prints the e–residue diagrams of all of the e–regular partitions appearing in x (such diagrams are useful when trying to decide how to restrict and induce modules and also in applying results such as the "Scattering theorem" of [JM1]). It is not necessary to supply the integer e in this case because x "knows" the value of e.

```
gap> H:=Specht(2);; EResidueDiagram(H.S(H.P(7,5)));
[ 7, 5 ]
   0   1   0   1   0   1   0
   1   0   1   0   1
[ 6, 5, 1 ]
   0   1   0   1   0   1
   1   0   1   0   1
   0
[ 5, 4, 2, 1 ]
   0   1   0   1   0
   1   0   1   0
   0   1
   1
#  There are 3 2-regular partitions.
```

This function requires the package "specht" (see 56.1).

1.36 HookLengthDiagram

`HookLengthDiagram(`μ`)`

Prints the diagram of μ, replacing each node with its hook length (see [JK]).

```
gap> HookLengthDiagram(11,6,3,2);
 14 13 11  9  8  7  5  4  3  2  1
  8  7  5  3  2  1
  4  3  1
  2  1
```

This function requires the package "specht" (see 56.1).

1.37 RemoveRimHook

`RemoveRimHook(`μ`, `row`, `col`)`

Returns the partition obtained from μ by removing the (row, col)–th rim hook from (the diagram of) μ.

```
gap> RemoveRimHook([6,5,4],1,2);
[ 4, 3, 1 ]
gap> RemoveRimHook([6,5,4],2,3);
[ 6, 3, 2 ]
```

```
gap> HookLengthDiagram(6,5,4);
   8   7   6   5   3   1
   6   5   4   3   1
   4   3   2   1
```

See also `AddRimHook` 1.38. This function requires the package "specht" (see 56.1).

1.38 AddRimHook

`AddRimHook(`μ`, `r`, `h`);`

Returns a list $[\nu, l]$ where ν is the partition obtained from μ by adding a rim hook of length h with its "foot" in the r–th row of (the diagram of) μ and l is the leg length of the wrapped on rim hook (see, for example, [JK]). If the resulting diagram ν is not the diagram of a partition then `false` is returned.

```
gap> AddRimHook([6,4,3],1,3);
[ [ 9, 4, 3 ], 0 ]
gap> AddRimHook([6,4,3],2,3);
false
gap> AddRimHook([6,4,3],3,3);
[ [ 6, 5, 5 ], 1 ]
gap> AddRimHook([6,4,3],4,3);
[ [ 6, 4, 3, 3 ], 0 ]
gap> AddRimHook([6,4,3],5,3);
false
```

See also `RemoveRimHook` 1.37. This function requires the package "specht" (see 56.1).

Operations on partitions

This section contains functions for manipulating partitions and also several useful orderings on the set of partitions.

1.39 ECore

`ECore(`H`|`e`, `μ`)`

The e-core of a partition μ is what remains after as many rim e-hooks as possible have been removed from the diagram of μ (that this is well defined is not obvious; see [JK]). Thus, `ECore(`μ`)` returns the e–core of the partition μ,

```
gap> H:=Specht(6);; ECore(H,16,8,6,5,3,1);
[ 4, 3, 1, 1 ]
```

The e–core is calculated here using James' notation of an **abacus**; there is also an `EAbacus` function; but it is more "pretty" than useful.

See also `IsECore` 1.40, `EQuotient` 1.41, and `EWeight` 1.43. This function requires the package "specht" (see 56.1).

1.40 IsECore

`IsECore(`H | e, μ`)`

Returns `true` if μ is an e–core and `false` otherwise; see `ECore` 1.39.

See also `ECore` 1.39. This function requires the package "specht" (see 56.1).

1.41 EQuotient

`EQuotient(`H | e, μ`)`

Returns the e-quotient of μ; this is a sequence of e partitions whose definition can be found in [JK].

```
gap> H:=Specht(8);; EQuotient(H,22,18,16,12,12,1,1);
[ [ 1, 1 ], [  ], [  ], [  ], [  ], [ 2, 2 ], [  ], [ 1 ] ]
```

See also `ECore` 1.39 and `CombineEQuotientECore` 1.42. This function requires the package "specht" (see 56.1).

1.42 CombineEQuotientECore

`CombineEQuotientECore(`H | e, Q, C`)`

A partition is uniquely determined by its e-quotient and its e-core (see 1.41 and 1.39). `CombineEQuotientECore(`e, Q, C`)` returns the partition which has e–quotient Q and e–core C. The integer e can be replaced with a record H which was created using the function `Specht`.

```
gap> H:=Specht(11);; mu:=[100,98,57,43,12,1];;
gap> Q:=EQuotient(H,mu);
[ [ 9 ], [  ], [  ], [  ], [  ], [  ], [ 3 ], [ 1 ], [ 9 ], [  ], [ 5 ] ]
gap> C:=ECore(H,mu);
[ 7, 2, 2, 1, 1, 1 ]
gap> CombineEQuotientECore(H,Q,C);
[ 100, 98, 57, 43, 12, 1 ]
```

See also `ECore` 1.39 and `EQuotient` 1.41. This function requires the package "specht" (see 56.1).

1.43 EWeight

`EWeight(`H | e, μ`)`

The e–weight of a partition is the number of e–hooks which must be removed from the partition to reach the e–core (see `ECore` 1.39).

```
gap> EWeight(6,[16,8,6,5,3,1]);
5
```

This function requires the package "specht" (see 56.1).

1.44 ERegularPartitions

`ERegularPartitions(`H`|`e`, `n`)`

A partition $\mu = (\mu_1, \mu_2, \ldots)$ is e–**regular** if there is no integer i such that $\mu_i = \mu_{i+1} = \cdots = \mu_{i+e-1} > 0$. The function `ERegularPartitions(`$e$`, `$n$`)` returns the list of e–regular partitions of n, ordered reverse lexicographically (see 1.52).

```
gap> H:=Specht(3);
Specht(e=3, S(), P(), D(), Pq());
gap> ERegularPartitions(H,6);
[ [ 2, 2, 1, 1 ], [ 3, 2, 1 ], [ 3, 3 ], [ 4, 1, 1 ], [ 4, 2 ],
  [ 5, 1 ], [ 6 ] ]
```

This function requires the package "specht" (see 56.1).

1.45 IsERegular

`IsERegular(`H`|`e`, `μ`)`

Returns `true` if μ is e–regular and `false` otherwise.

This functions requires the package "specht" (see 56.1).

1.46 ConjugatePartition

`ConjugatePartition(`μ`)`

Given a partition μ, `ConjugatePartition(`μ`)` returns the partition whose diagram is obtained by interchanging the rows and columns in the diagram of μ.

```
gap> ConjugatePartition(6,4,3,2);
[ 4, 4, 3, 2, 1, 1 ]
```

This function requires the package "specht" (see 56.1).

1.47 BetaSet

`BetaSet(`μ`)`

This function returns a **set** of beta numbers (i.e. first column hook lengths; see [JK]) corresponding to the partition μ.

```
gap> BetaSet([5,4,2,2]);
[ 2, 3, 6, 8 ]
```

See also `PartitionBetaSet` 1.48. This function requires the package "specht" (see 56.1).

1.48 PartitionBetaSet

`PartitionBetaSet(`bn`)`

Given a **set** of beta numbers bn (see `BetaSet` 1.47), this function returns the corresponding partition. Note in particular that bn must be a set of integers.

```
gap> PartitionBetaSet([ 2, 3, 6, 8 ]);
[ 5, 4, 2, 2 ]
```

This function requires the package "specht" (see 56.1).

1.49 ETopLadder

`ETopLadder(`H | e, μ`)`

The ladders in the diagram of a partition are the lines connecting nodes of constant e–residue, having slope $e-1$ (see [JK]). A new partition can be obtained from μ by sliding all nodes up to the highest possible rungs on their ladders. `ETopLadder(`e, μ`)` returns the partition obtained in this way; it is automatically e–regular (this partition is denoted μ^R in [JK]).

```
gap> H:=Specht(4);;
gap> ETopLadder(H,1,1,1,1,1,1,1,1,1,1);
[ 4, 3, 3 ]
gap> ETopLadder(6,1,1,1,1,1,1,1,1,1,1);
[ 2, 2, 2, 2, 2 ]
```

This function requires the package "specht" (see 56.1).

1.50 Dominates

`Dominates(`μ, ν`)`

The dominance ordering is an important partial order in the representation theory of Hecke algebra because $d_{\mu\nu} = 0$ unless ν dominates μ. `Dominates(`μ, ν`)` returns `true` if either μ=ν or for all $i \geq 1$, $\sum_{j=1}^{i} \mu_j \geq \sum_{j=1}^{i} \nu_j$, and `false` otherwise.

```
gap> Dominates([5,4],[4,4,1]);
true
```

This function requires the package "specht" (see 56.1).

1.51 LengthLexicographic

`LengthLexicographic(`μ, ν`)`

`LengthLexicographic` returns `true` if the length of μ is less than the length of ν or if the length of μ equals the length of ν and `Lexicographic(`μ, ν`)`.

```
gap> p:=Partitions(6);;Sort(p,LengthLexicographic); p;
[ [ 6 ],[ 5, 1 ],[ 4, 2 ],[ 3, 3 ],[ 4, 1, 1 ],[ 3, 2, 1 ],[ 2, 2, 2 ],
  [ 3, 1, 1, 1 ],[ 2, 2, 1, 1 ],[ 2, 1, 1, 1, 1 ],[ 1, 1, 1, 1, 1, 1 ] ]
```

This function requires the package "specht" (see 56.1).

1.52 Lexicographic

`Lexicographic(`μ, ν`)`

`Lexicographic(`μ, ν`)` returns `true` if μ is lexicographically greater than or equal to ν.

```
gap> p:=Partitions(6);;Sort(p,Lexicographic); p;
[ [ 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 ] ]
```

This function requires the package "specht" (see 56.1).

1.53 ReverseDominance

`ReverseDominance(`μ`, `ν`)`

This is another total order on partitions which extends the dominance ordering (see 1.50). Here μ is greater than ν if for all $i > 0$

$$\sum_{j \geq i} \mu_j > \sum_{j \geq i} \nu_j.$$

```
gap> p:=Partitions(6);;Sort(p,ReverseDominance); p;
[ [ 6 ], [ 5, 1 ], [ 4, 2 ], [ 3, 3 ], [ 4, 1, 1 ], [ 3, 2, 1 ],
  [ 2, 2, 2 ], [ 3, 1, 1, 1 ], [ 2, 2, 1, 1 ], [ 2, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1 ] ]
```

This is the ordering used by James in the appendix of his Springer lecture notes book.

This function requires the package "specht" (see 56.1).

Miscellaneous functions on modules

This section contains some functions for looking at the partitions in a given module for the Hecke algebras. Most of them are used internally by SPECHT.

1.54 Specialized

`Specialized(x [,q]);`
`Specialized(d [,q]);`

Given an element of the Fock space x (see 1.1), or a crystallized decomposition matrix (see 1.4), `Specialized` returns the corresponding element of the Grothendieck ring or the corresponding decomposition matrix of the Hecke algebra respectively. By default the indeterminate `v` is specialized to 1; however `v` can be specialized to any (integer) q by supplying a second argument.

```
gap> H:=Specht(2);; x:=H.Pq(6,2);
S(6,2)+v*S(6,1,1)+v*S(5,3)+v^2*S(5,1,1,1)+v*S(4,3,1)+v^2*S(4,2,2)
+(v^3 + v)*S(4,2,1,1)+v^2*S(4,1,1,1,1)+v^2*S(3,3,1,1)+v^3*S(3,2,2,1)
+v^3*S(3,1,1,1,1,1)+v^3*S(2,2,2,1,1)+v^4*S(2,2,1,1,1,1)
gap> Specialized(x);
S(6,2)+S(6,1,1)+S(5,3)+S(5,1,1,1)+S(4,3,1)+S(4,2,2)
+2*S(4,2,1,1)+S(4,1,1,1,1)+S(3,3,1,1)+S(3,2,2,1)+S(3,1,1,1,1,1)
+S(2,2,2,1,1)+S(2,2,1,1,1,1)
gap> Specialized(x,2);
S(6,2)+2*S(6,1,1)+2*S(5,3)+4*S(5,1,1,1)+2*S(4,3,1)+4*S(4,2,2)+10*S(4,2,1,1)
+4*S(4,1,1,1,1)+4*S(3,3,1,1)+8*S(3,2,2,1)+8*S(3,1,1,1,1,1)+8*S(2,2,2,1,1)
+16*S(2,2,1,1,1,1)
```

An example of `Specialize` being applied to a crystallized decomposition matrix can be found in 1.4. This function requires the package "specht" (see 56.1).

1.55 ERegulars

`ERegulars(`x`)`
`ERegulars(`d`)`
`ListERegulars(`x`)`

`ERegulars(`x`)` prints a list of the e–regular partitions, together with multiplicities, which occur in the module x. `ListERegulars(`x`)` returns an actual list of these partitions rather than printing them.

```
gap> H:=Specht(8);;
gap> x:=H.S(InducedModule(H.P(8,5,3)) );
S(9,5,3)+S(8,6,3)+S(8,5,4)+S(8,5,3,1)+S(6,5,3,3)+S(5,5,4,3)+S(5,5,3,3,1)
gap> ERegulars(x);
[ 9, 5, 3 ]  [ 8, 6, 3 ]  [ 8, 5, 4 ]  [ 8, 5, 3, 1 ]
[ 6, 5, 3, 3 ]  [ 5, 5, 4, 3 ]  [ 5, 5, 3, 3, 1 ]
gap> H.P(x);
P(9,5,3)+P(8,6,3)+P(8,5,4)+P(8,5,3,1)
```

This example shows why these functions are useful: given a projective module x, as above, and the list of e–regular partitions in x we know the possible indecomposable direct summands of x.

Note that it is not necessary to specify what e is when calling this function because x "knows" the value of e.

The function `ERegulars` can also be applied to a decomposition matrix d; in this case it returns the unitriangular submatrix of d whose rows and columns are indexed by the e–regular partitions.

These function requires the package "specht" (see 56.1).

1.56 SplitECores

`SplitECores(`x`)`
`SplitECores(`x`, `μ`)`
`SplitECores(`x`, `y`)`

The function `SplitECores(`x`)` returns a list `[`b_1`,...,`b_k`]` where the Specht modules in each b_i all belong to the same block (i.e. they have the same e-core). Similarly, `SplitECores(`x`, `μ`)` returns the component of x which is in the same block as μ, and `SplitECores(`x`, `y`)` returns the component of x which is in the same block as y.

```
gap> H:=Specht(2);;
gap> SplitECores(InducedModule(H.S(5,3,1)));
[ S(6,3,1)+S(5,3,2)+S(5,3,1,1), S(5,4,1) ]
gap> InducedModule(H.S(5,3,1),0);
S(5,4,1)
gap> InducedModule(H.S(5,3,1),1);
S(6,3,1)+S(5,3,2)+S(5,3,1,1)
```

See also `ECore` 1.39, `InducedModule` 1.6, and `RestrictedModule` 1.8.

This function requires the package "specht" (see 56.1).

1.57 Coefficient

`Coefficient(`x`, `μ`)`

If x is a sum of Specht (resp. simple, or indecomposable) modules, then `Coefficient(`x`, `μ`)` returns the coefficient of `S(`μ`)` in x (resp. `D(`μ`)`, or `P(`μ`)`).

```
gap> H:=Specht(3);; x:=H.S(H.P(7,3));
S(7,3)+S(7,2,1)+S(6,2,1^2)+S(5^2)+S(5,2^2,1)+S(4^2,1^2)+S(4,3^2)+S(4,3,2,1)
gap> Coefficient(x,5,2,2,1);
1
```

This function requires the package "specht" (see 56.1).

1.58 InnerProduct

`InnerProduct(`x`, `y`)`

Here x and y are some modules of the Hecke algebra (i.e. Specht modules, PIMS, or simple modules). `InnerProduct(`x`, `y`)` computes the standard inner product of these elements. This is sometimes a convenient way to compute decomposition numbers (for example).

```
gap> InnerProduct(H.S(2,2,2,1), H.P(4,3));
1
gap> DecompositionNumber(H,[2,2,2,1],[4,3]);
1
```

This function requires the package "specht" (see 56.1).

1.59 SpechtPrettyPrint

`SpechtPrettyPrint(true)`
`SpechtPrettyPrint(false)`
`SpechtPrettyPrint()`

This function changes the way in which SPECHT prints modules. The first two forms turn pretty printing on and off respectively (by default it is off), and the third form toggles the printing format.

```
gap> H:=Specht(2);; x:=H.S(H.P(6));;
gap> SpechtPrettyPrint(true); x;
S(6)+S(5,1)+S(4,1^2)+S(3,1^3)+S(2,1^4)+S(1^6)
gap> SpechtPrettyPrint(false); x;
S(6)+S(5,1)+S(4,1,1)+S(3,1,1,1)+S(2,1,1,1,1)+S(1,1,1,1,1,1)
gap> SpechtPrettyPrint(); x;
S(6)+S(5,1)+S(4,1^2)+S(3,1^3)+S(2,1^4)+S(1^6)
```

This function requires the package "specht" (see 56.1).

Semi–standard and standard tableaux

These functions are not really part of SPECHT proper; however they are related and may well be of use to someone. Tableaux are represented as lists, where the first element of the list is the first row of the tableaux and so on.

1.60 SemiStandardTableaux

`SemiStandardTableaux(`μ`, `ν`)`

μ a partition, ν a composition.

Returns a list of the semistandard μ–tableaux of type ν [JK]. Tableaux are represented as lists of lists, with the first element of the list being the first row of the tableaux and so on.

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

See also `StandardTableaux` 1.61. This function requires the package "specht" (see 56.1).

1.61 StandardTableaux

`StandardTableaux(`μ`)`

μ a partition.

Returns a list of the standard μ–tableaux.

```
gap> StandardTableaux(4,2);
[ [ [ 1, 2, 3, 4 ], [ 5, 6 ] ], [ [ 1, 2, 3, 5 ], [ 4, 6 ] ],
  [ [ 1, 2, 3, 6 ], [ 4, 5 ] ], [ [ 1, 2, 4, 5 ], [ 3, 6 ] ],
  [ [ 1, 2, 4, 6 ], [ 3, 5 ] ], [ [ 1, 2, 5, 6 ], [ 3, 4 ] ],
  [ [ 1, 3, 4, 5 ], [ 2, 6 ] ], [ [ 1, 3, 4, 6 ], [ 2, 5 ] ],
  [ [ 1, 3, 5, 6 ], [ 2, 4 ] ] ]
```

See also `SemiStandardTableaux` 1.60. This function requires the package "specht" (see 56.1).

1.62 ConjugateTableau

`ConjugateTableau(`*tab*`)`

Returns the tableau obtained from *tab* by interchangings its rows and columns.

```
gap> ConjugateTableau([ [ 1, 3, 5, 6 ], [ 2, 4 ] ]);
[ [ 1, 2 ], [ 3, 4 ], [ 5 ], [ 6 ] ]
```

This function requires the package "specht" (see 56.1).

1.63 ShapeTableau

`ShapeTableau(`*tab*`)`

Given a tableau *tab* this function returns the partition (or composition).

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

This function requires the package "specht" (see 56.1).

1.64 TypeTableau

`TypeTableau(`*tab*`)`

Returns the type of the (semistandard) tableau *tab*; that is, the composition $\sigma = (\sigma_1, \sigma_2, \ldots)$ where σ_i is the number of entries in *tab* which are equal to i.

```
gap> List(SemiStandardTableaux([5,4,2],[4,3,0,1,3]),TypeTableau);
[ [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ],
  [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ] ]
```

This function requires the package "specht" (see 56.1).

Installing SPECHT

When you unpack SPECHT you will find the following files in a directory called specht-2.4:

```
README      -this file
doc/        -Specht documentation (see below)
gap/        -GAP source
init.g      -initialization file
lib/        -Specht library files
```

Ideally, SPECHT should be installed in the GAP packages directory (in a subdirectory called "specht"); however, it can be installed anywhere. If SPECHT is not installed in the GAP packages directory then include a line of the form

```
Add(PKGNAME, "/path/to/directory/containing/specht/");
PKGNAME:=Reversed(PKGNAME);
```

in your .gaprc in your home directory (create such a file with this line if you do not already have one). SPECHT is now ready to use.

```
gap> RequirePackage("specht");
gap> H:=Specht(3);
Specht(e=3, S(), P(), D(), Pq())
```

Installing SPECHTs online documentation

The documentation for Specht can be found in the subdirectory `doc`. The more important files in this directory are:

```
specht.tex  -LaTeX source for the manual
specht.html -an HTML version of the manual
manual.tex  -header file for specht.tex (a LaTeX2e file)
manual.dvi  -a dvi file for SPECHT
install.tex -these installation notes
```

To install the **online** documentation for SPECHT proceed as follows:

```
1. Copy specht.tex into the GAP doc/ directory,
2. Change directory to GAPs <doc/> directory and add the line
```

Include{*specht*} `to` *manual.tex*`.`

`3.` LaTeX *manual.tex*`.`

`4. Run makeindex (if available) and repeat 3.`

A copy of just the SPECHT manual can be obtained by printing the dvi file *manual.dvi* (or LaTeX2eing the file *manual.tex*) in SPECHTS *doc/* directory.

Index

AddIndecomposable, 22
AddRimHook, 31
AdjustmentMatrix, 19

BetaSet, 33

CalculateDecompositionMatrix, 20
Canonical basis, 8
Coefficient, 37
CombineEQuotientECore, 32
ConjugatePartition, 33
ConjugateTableau, 38
CrystalDecompositionMatrix, 11

Decomposition numbers of Hecke algebras
    of type A, 1
DecompositionMatrix, 10
    TeX, 11
DecompositionMatrixMatrix, 21
DecompositionNumber, 12
Dominates, 34

EAbacus, 31
ECore, 31
EQuotient, 32
ERegularPartitions, 33
ERegulars, 36
EResidueDiagram, 29
ETopLadder, 34
EWeight, 32

Fock space, 8

GoodNodeLatticePath, 28
GoodNodes, 26
GoodNodeSequence, 27
GoodNodeSequences, 27

H.Ordering
    Specht, 11

Hecke
    Specht, 4
Hecke algebras
    crystal decomposition matrix, 11
    decomposition matrix, 10
HookLengthDiagram, 30

InducedDecompositionMatrix, 17
InducedModule, 13
InnerProduct, 37
InverseLittlewoodRichardsonRule, 29
InvertDecompositionMatrix, 19
IsECore, 32
IsERegular, 33
IsNewIndecomposable, 17
IsSimpleModule, 24

LengthLexicographic, 34
Lexicographic, 34
ListERegulars, 36
LittlewoodRichardsonRule, 28

MatrixDecompositionMatrix, 21
MissingIndecomposables, 22
MullineuxMap, 25
MullineuxSymbol, 26

NormalNodes, 27

PartitionBetaSet, 33
PartitionGoodNodeSequence, 27
PartitionMullineuxSymbol, 26

RemoveIndecomposable, 22
RemoveRimHook, 30
RestrictedModule, 15
ReverseDominance, 35

SaveDecompositionMatrix, 20
Schaper, 24

Schur, 9
SemiStandardTableaux, 38
ShapeTableau, 38
SimpleDimension, 23
SInducedModule, 15
Specht, 4
SpechtDimension, 23
SpechtDirectory, 10, 20
SpechtPrettyPrint, 37
Specialized, 35
SplitECores, 36
SRestrictedModule, 16
StandardTableaux, 38

TeX
    DecompositionMatrix, 11
TypeTableau, 39