CREP Manual
Part II

Peter Dräxler
Rainer Nörenberg
(editors)

Contents

Introduction **5**

1 Additions to the CREP package **7**
1.1 Additions to the basic package of crep 7
1.2 wp - test a unit form for weak positivity 7
1.3 wnn - test a unit form for weak non-negativity 9

2 Additions to the heralg package **11**
2.1 New features of the heralg package 11
2.2 heralg[perpgen] - the projective generator of the right perpendicular category of a module 11
2.3 heralg[perp_cm] - the Cartan matrix of the algebra corresponding to the right perpendicular category of a module 12
2.4 heralg[perp_quiver] - the quiver of the algebra corresponding to the right perpendicular category of a module 13
2.5 heralg[star] - the hereditary star algebra of type $(p_1, p_2, \dots)$ given in the form of a quiverdata structure 14

3 The xpreproj interface **15**
3.1 xpreproj - frontend for preproj 15

4 The tubular package **17**
4.1 TUBULAR - Program Package for treatment of tubular canonical algebras . . . 17
4.2 tubular/datatypes 18
4.3 tubular/Lp 19
4.4 tubular/globalvars 19
4.5 The Binary Operator &+ 20
4.6 The Prefix Operator &- 20
4.7 The Binary Operator &m 20
4.8 The Binary Operator &= 21
4.9 The Binary Operator &<> 21
4.10 The Binary Operator &<= 22
4.11 The Binary Operator &>= 22
4.12 The Binary Operator &< 22
4.13 The Binary Operator &> 23
4.14 NF - normal form for an element of Lp; 23
4.15 delta - evaluate the delta-function on Lp. 24
4.16 read_in - enter a valid weight sequence 24
4.17 get_class - conversion from "tub_enc" to K_0X 25
4.18 get_tub - conversion from K_0X to "tub_enc" 26
4.19 type Lp - implementing the Lp-grading group. 27
4.20 type quiver - chosen for "pretty printing" classes in K_0X 28
4.21 type grovect_enter - designed for entering arbitrary coordinates 28
4.22 type tub_enc - encoding an indecomposable sheaf with respect to its position within the tubular families. 28
4.23 Cartan - compute Cartan matrix 29
4.24 R - linear transformation corresponding to mutation R 30

4.25 S - linear transformation corresponding to mutation S 30
4.26 show - "pretty"-prints type quiver 31
4.27 enter - direct input of classes in K_0X 32
4.28 change - change of K_0X -basis for objects of type quiver. 33
4.29 telescope 34
4.30 tau - Auslander-Reiten-shift on K_0X 34
4.31 tauinv - inverse Auslander-Reiten-shift 35
4.32 Lp_shift - shift of a K_0-class 36
4.33 rank_ - compute the rank of a K_0X-class 36
4.34 degree_ - compute the degree of a K_0X-class 37
4.35 slope_ - compute the slope of a K_0X-class 37
4.36 eulf - the euler-characteristic 38
4.37 The Binary Operator &++ 39
4.38 The Prefix Operator &-- 40
4.39 The Binary Operator &== 41
4.40 &mm - scalar product of a K_0X class with an integer. 41
4.41 quaf - the quadratic form for a K_0X-class 42

5 The repetit package **44**
5.1 The CREP-package repetit for repetitive algebras and iteratedly tilted algebras 44
5.2 repetit/datatypes 44
5.3 it_tilt_alg - representatives of frames of iteratedly tilted algebras of type E. . . . 45
5.4 it_tilted_num - the numbers of frames of iteratedly tilted algebras of type E. . . 46
5.5 it_tilt_test - test for a iteratedly tilted algebra of type E 46
5.6 repetitive - the repetitive algebra of a bound poset algebra 47
5.7 repetitive_alg - representatives of classes of repetitive algebras of type E 47
5.8 rep_alg_num - the numbers of classes of repetitive algebras of type E 48
5.9 repetit_addrel - include relations for a repetitive algebra 48

6 The tpe package **50**
6.1 The CREP-package tpe for completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras. 50
6.2 tpe_alg - representatives of frames of completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras. 50
6.3 tpe_num - the numbers of frames of completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras. 51
6.4 tpe_test - test for completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras. 51

Introduction

With this second volume of the Crep manual we are able to present new packages of the Crep system as well as some additional functions that have been added to existing packages since the publication of the version 1.0 described in the first volume of the manual [2].

Due to the modular concept of Crep, most changes in the system take the shape of new packages to be used together with the ones already existing. However there are two exceptions where the contents of a package already existing have been changed.

First, the new programs of Hans von Höhne for testing a unit form for weak positivity respectively weak non-negativity, using edge reduction, bring an amazing improvement of speed compared to the older programs of the main package of Crep. Nevertheless, we also maintain the old functions, since the side information (which may be of interest for particular questions) provided there is different.

Next, we have decided to include the new functions Stefan Lache has provided to deal with perpendicular categories of exceptional modules of a hereditary algebra directly into his package heralg for hereditary algebras, rather than making them a package of their own.

The most prominent addition to Crep presented in this manual is the 'tubular'-package by Piotr Dowbor and Thomas Hübner. This well structured package written entirely in Maple contains a collection of functions for working with canonical, especially tubular canonical algebras. While its main concern originally was the implementation and evaluation of so called telescoping functors, it also contains implementations for working in the Grothendieck group of a canonical algebra. It will not go unnoticed, that the functions of this package are not listed in alphabetical order, but rather in such a way that function and operators sharing a common subject are grouped together, reflecting the inner structure of the package.

Crep's data base is enlarged by another two packages.

First there is the 'repetit'-package by Bernd Roggon. This package gives access to the classes of repetitive algebras of type E_6, E_7, E_8 and iteratedly tilted algebras of type E_6, E_7, E_8, as arising form the classification in the thesis of Bernd Roggon (cf [4],[5]).

Then there is the 'tpe'-package containing the classes of tame completely separating poset algebras, as classified in [1].

While in general all communication with the Crep system is established via Maple's user interface, this is not the case for the function preproj of Crep's main package. This function for construction preprojective components requires some additional interactive input by the user. The way this interaction is handled by the preproj-function as has been considered somewhat awkward by some users. The xpreproj-function written by Dietmar Guhe now provides an interface for the preproj function using the possibilities of a modern graphical user interface to handle the interactions with the underlying program and to display the results. This may prove a model for handling more complex data in the future.

Note that information about the mathematical background of most of the packages in this manual and their underlying algorithms will appear in the proceedings of the Euroconference "Computational Methods for Representations of Groups and Algebras" in Essen, April 1996 [3].

Our Thanks go to the authors of the new functions and packages, who have not only agreed to have their code made part of Crep, but in most cases also have provided the descriptions of the resulting functions.

References

[1] P. Dräxler, R. Nörenberg, *Classification problems in the combinatorial representation theory of finite-dimensional algebras*, Ergänzungsreihe 97-008, Sonderforschugsbereich 343, Diskrete Strukturen in der Mathematik, Universität Bielefeld

[2] P. Dräxler, R. Nörenberg, *The Crep Manual*, Ergänzungsreihe 96-002, Sonderforschugsbereich 343, Diskrete Strukturen in der Mathematik, Universität Bielefeld, 1996

[3] P. Dräxler, G. Michler, C. M. Ringel (editors), *Proceedings of the Euroconference "Computational Methods for Representations of Groups and Algebras", Essen, April 1996*, to appear

[4] B.Roggon, *Selbstinjektive Algebren und iterierte Kippalgebren der Klassen* E_6, E_7, E_8, Diplomarbeit, Bielefeld, 1994

[5] B.Roggon, *Selfinjective and Iterarted Tilted Algebras of Type* E_6, E_7, E_8, Ergänzungsreihe 95-008, Sonderforschugsbereich 343, Diskrete Strukturen in der Mathematik, Universität Bielefeld, 1995

Crep can be obtained via `ftp` from the server
`ftp.uni-bielefeld.de`
where it can be found in the directory
`pub/math/f-d-alg/crep`

Running Crep will require the Maple system to be present, preferably MapleV release 3 or 4. For installation of some packages of Crep, a Pascal compiler or a C compiler will be needed. In particular, the tpe-package and the main package (including the new functions) require Pascal, the xpreproj interface requires C, and the repetit-package requires both pascal and C for a complete installation.

For questions or remarks on Crep, one may use the e-mail address
`fdowner@mathematik.uni-bielefeld.de` or contact

Peter Dräxler,
Fakultät für Mathematik,
Universität Bielefeld,
Postfach 100 131,
D-33501 Bielefeld,
e-mail: `draexler@mathematik.uni-bielefeld.de`.

1 Additions to the CREP package

Hans von Höhne (Berlin)

1.1 Additions to the basic package of crep

Synopsis:

- In addition to the functions described in the CREP Manual (Part I) The following functions have been added to the basic package of crep:

  wp wnn

- For more information on a particular function see `<function>`.

1.2 wp - test a unit form for weak positivity

Calling Sequence:

`wp(`*q*`)`
`wp(`*q,rt*`)`
`wp(`*q,rt,r*`)`

Parameters:

q – a unit form
rt – an optional parameter evaluating to a name
r – an optional parameter evaluating to 'a' or 'A'

Synopsis:

- `wp` uses a program of Hans von Hoehne to test whether the unit form q given as first argument is weakly positive.
- In case the function is called with exactly one argument (which then has to be the unit form q), `wp` will return the value `false` if q is not weakly positive, and, if q is weakly positive, will return a pair $[s,t]$ of non-negative integers, s being the number of all positive 1-roots of q, t the number of all sincere 1-roots
- In case the function is called with more than one argument, it will return the value `true` if its first argument, the unit form q, is weakly positive and the value `false` otherwise.
- In case the unit form q is weakly positive and a second argument rt is given, this argument will be used to return a table of positive 1-roots of the unit form q. If no third argument r is given, or if the third argument r evalutes to 'A', this table will consist of all sincere positive 1-roots of q. If the argument r evalutes to 'a', rt will be assigned a list of all positive 1-roots of q.

Examples:

```
> a:=[4,[[1,2,3,4]]];

                              a := [4, [[1, 2, 3, 4]]]
> q:=titsform(a);
```

```
                      [  2  -1  -1  -1 ]
                      [                ]
                      [ -1   2   0   0 ]
                      [                ]
                      [ -1   0   2   0 ]
                      [                ]
                      [ -1   0   0   2 ]
> wp(q);

                          [12, 2]
> wp(q,'rt');

                           true
> print(rt);

                     table([
                         1 = [2, 1, 1, 1]
                         2 = [1, 1, 1, 1]
                     ])
> wp(q,'rt','a');

                           true
> print(rt);

                     table([
                         1 = [1, 0, 0, 0]
                         2 = [0, 1, 0, 0]
                         3 = [0, 0, 1, 0]
                         4 = [0, 0, 0, 1]
                         5 = [1, 1, 0, 0]
                         6 = [1, 0, 1, 0]
                         7 = [1, 0, 0, 1]
                         8 = [1, 0, 1, 1]
                         9 = [2, 1, 1, 1]
                         10 = [1, 1, 1, 0]
                         11 = [1, 1, 0, 1]
                         12 = [1, 1, 1, 1]
                     ])
> wp(q,'rt','A');

                           true
> print(rt);

                     table([
                         1 = [2, 1, 1, 1]
                         2 = [1, 1, 1, 1]
                     ])
```

SEE ALSO: weakly_positive, wnn

1.3 wnn - test a unit form for weak non-negativity

Calling Sequence:

wnn(q)

wnn(q,rt)

Parameters:

q – a unit form

rt – an optional parameter evaluating to a name

Synopsis:

- wnn uses a program of Hans von Hoehne to test whether the unit form q given as first argument is weakly non-negative.
- If q is weakly non-negative, the function will return the value true, otherwise the value false.
- If q is weakly non-negative, the second argument rt, if given, will be assigned a list of the critical vectors of q.
- If q is not weakly non-negative, the second argument rt, if given, will be assigned a single positive vector on which the form q is negative.

Examples:

```
> a:=[5,[[1,2,3,4,5]]];

                          a := [5, [[1, 2, 3, 4, 5]]]
> q:=titsform(a);

                            [  2  -1  -1  -1  -1 ]
                            [                    ]
                            [ -1   2   0   0   0 ]
                            [                    ]
                            [ -1   0   2   0   0 ]
                            [                    ]
                            [ -1   0   0   2   0 ]
                            [                    ]
                            [ -1   0   0   0   2 ]
> wnn(q,'rt');

                                     true
> print(rt);

                            table([
                                1 = [2, 1, 1, 1, 1]
                            ])
----------------------------------------------------------------------------
> a:=hypercritical_algd(4,1);
```

```
                [6, {[3, 1], [4, 1], [2], [5, 4], [6], [1, 2, 6]}]
> q:=titsform(a);

                        [  2  -1  -1  -1   0  -1 ]
                        [                        ]
                        [ -1   2   0   0   0   0 ]
                        [                        ]
                        [ -1   0   2   0   0   0 ]
                        [                        ]
                        [ -1   0   0   2  -1   0 ]
                        [                        ]
                        [  0   0   0  -1   2   0 ]
                        [                        ]
                        [ -1   0   0   0   0   2 ]
> wnn(q,'rt');

                                  false
> print(rt);

                           [6, 3, 2, 3, 1, 3]
> 1/2*innerprod(rt,q,rt);

                                   -1
```

SEE ALSO: `wp, weakly_nonneg`

2 Additions to the heralg package

Stefan Lache (Düsseldorf)

The heralg package is a supplement for CREP dealing with hereditary algebras. It is written wholly in Maple and organised as a Maple library. It can thus also be used independently of the CREP system.

The package has already been part of CREP distribution 1.0 and 1.1 and described in the CREP Manual (version 1.0). The new features of the package mainly are concerned with the right perpendicular category of an indecomposable module without self extensions.

2.1 New features of the heralg package

Synopsis:

- In addition to the functions described in the CREP Manual (version 1.0) The following functions have been added to the heralg package:

  perpgen perp_cm perp_quiver star

- To use a heralg function, either define that function alone using the command `with(heralg, <function>)`, or define all **heralg** functions using the command `with(heralg)`. Alternatively, invoke the function using the long form `heralg[<function>]`.

- For more information on a particular function see `heralg[<function>]`.

2.2 heralg[perpgen] - the projective generator of the right perpendicular category of a module

Calling Sequence:

`perpgen`(x, Q)

Parameters:

x – a vector

Q – a quiver given by a quiverdata structure

Synopsis:

- The function `perpgen`(x, Q) computes the dimension vectors of the indecomposable projective objects in the right perpendicular category of the module of dimension vector x over the path algebra of the quiver Q. This module must be an exceptional module, i.e. an indecomposable module without self extensions.

- The algorithm uses a simple heuristic which will fail if the module considered is a tau-sincere regular module (or too far away from a non sincere, projective or injective module).

- This function is part of the `heralg` package, and so can be used in the form `perpgen(..)` only after performing the command `with(heralg)` or `with(heralg, perpgen)`. The function can always be accessed in the long form `heralg[perpgen](..)`.

Examples:

```
> with(heralg):
> perpgen(array([4,5,0]), [3,[[2,1,2],[3,2]]]);

                    [[ 5, 6, 0 ], [ 30, 36, 1 ]]
```

```
> Q:= [ 4, [[2,1], [3,1], [3,2], [4,3]]];

                  Q := [4, [[2, 1], [3, 1], [3, 2], [4, 3]]]

> x := array( [ 9,6,6,2 ] );

                            x := [ 9, 6, 6, 2 ]

> perpgen(x,Q);

              [[ 6, 4, 4, 1 ], [ 8, 6, 5, 2 ], [ 2, 1, 1, 1 ]]
```

SEE ALSO: heralg/quiverdata, heralg[perp_quiver]

2.3 heralg[perp_cm] - the Cartan matrix of the algebra corresponding to the right perpendicular category of a module

Calling Sequence:
`perp_cm`(x, Q)
Parameters:
x – a vector
Q – a quiver given by a quiverdata structure
Synopsis:

- The function `perp_cm`(x, Q) uses the function `perpgen` (also part of the `heralg` package) to compute the Cartan matrix of the quiver algebra corresponding to the right perpendicular category of the module of dimension vector x over the path algebra of the quiver Q. This module must be an exceptional module, i.e. an indecomposable module without self extensions.
- The algorithm uses a simple heuristic which will fail if the module considered is a tau-sincere regular module (or too far away from a non sincere, projective or injective module).
- This function is part of the `heralg` package, and so can be used in the form `perp_cm(..)` only after performing the command `with(heralg)` or `with(heralg, perp_cm)`. The function can always be accessed in the long form `heralg[perp_cm](..)`.

Examples:

```
> with(heralg):
> perp_cm(array([4,5,0]), [3,[[2,1,2],[3,2]]]);

                                 [ 1  6 ]
                                 [      ]
                                 [ 0  1 ]

> Q:= [ 4, [[2,1], [3,1], [3,2], [4,3]]];

                  Q := [4, [[2, 1], [3, 1], [3, 2], [4, 3]]]
```

```
> x := array( [ 9,6,6,2 ] );

                              x := [ 9, 6, 6, 2 ]

> perp_cm(x,Q);

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

SEE ALSO: heralg/quiverdata, heralg[perpgen], heralg[perp_quiver]

2.4 heralg[perp_quiver] - the quiver of the algebra corresponding to the right perpendicular category of a module

Calling Sequence:

perp_quiver(x, Q)

Parameters:

x – a vector

Q – a quiver given by a quiverdata structure

Synopsis:

- The function perp_quiver(x, Q) uses the function perp_cm (also part of the heralg package) to compute the quiver of the quiver algebra corresponding to the right perpendicular category of the module of dimension vector x over the path algebra of the quiver Q. This module must be an exceptional module, i.e. an indecomposable module without self extensions.
- The algorithm uses a simple heuristic which will fail if the module considered is a tau-sincere regular module (or too far away from a non sincere, projective or injective module).
- This function is part of the heralg package, and so can be used in the form perp_quiver(..) only after performing the command with(heralg) or with(heralg, perp_quiver). The function can always be accessed in the long form heralg[perp_quiver](..).

Examples:

```
> with(heralg):
> perp_quiver(array([4,5,0]), [3,[[2,1,2],[3,2]]]);

                              [2, [[2, 1, 6]]]

> Q:= [ 4, [[2,1], [3,1], [3,2], [4,3]]];

                Q := [4, [[2, 1], [3, 1], [3, 2], [4, 3]]]

> x := array( [ 9,6,6,2 ] );

                              x := [ 9, 6, 6, 2 ]
```

```
> perp_quiver(x,Q);
```

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

SEE ALSO: `heralg/quiverdata`, `heralg[perpgen]`, `heralg[perp_cm]`

2.5 heralg[star] - the hereditary star algebra of type $(p_1, p_2, \ldots)$ given in the form of a quiverdata structure

Calling Sequence:
`star`$(p_1,\ p_2,\ \ldots)$
Parameters:
$p_1,\ p_2,\ \ldots$ – positive integers
Synopsis:

- The function `star`$(p_1, p_2, \ldots)$ computes a quiverdata structure for the hereditary star algebra of type $(p_1, p_2, \ldots)$. The quiver is given subspace orientation, i.e. the central vertex (vertex 1) is the unique sink of the quiver.
- This function is part of the `heralg` package, and so can be used in the form `star(..)` only after performing the command `with(heralg)` or `with(heralg, star)`. The function can always be accessed in the long form `heralg[star](..)`.

Examples:

```
> with(heralg):
> star(2,2,2,2);
```

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

```
> star(2,3,7);
```

$$[9, [[2, 1], [3, 1], [4, 3], [5, 1], [6, 5], [7, 6], [8, 7], [9, 8]]]$$

SEE ALSO: `heralg/quiverdata`

3 The xpreproj interface

Dietmar Guhe (Bielefeld)

The `xpreproj` function provides a graphical front end for the function `xpreproj` of the main package of CREP. It thus can only be used together with CREP's main package.

To make the function available, read in the file `xpreproj.src` located in the CREP home directory using the command `read ‘xpreproj.src‘`.

3.1 xpreproj - frontend for preproj

Calling Sequence:
`xpreproj(`p`)`
Parameters:
p – a bound poset algebra
Synopsis:

- `xpreproj` passes the algebra p to CREP-function `preproj`, which then tries to compute the beginning of a preprojective component of the Auslander-Reiten quiver of p.
- By executing `xpreproj` you get a window which is divided into three areas:
  - The first one, on the upper left of the window, encompasses the buttons 'apply' and 'end'. Pressing the 'apply' button executes the preproj algorithm with the arguments stated in the second area and then the output is given to the third area.
  - The second area gives you the possibility to rearrange the output of the preproj algorithm. After the label 'Which output?' you can vary the output of the dimension-vectors (see also `help preproj`). Following that you can insert a number of tau-translations by moving the mouse pointer into the rectangle after the label and then typing in the desired amount. By pressing the 'reordering' button you receive a window in which you can type in the length of the section (see also 'help preproj'). Then a window named 'Reordering' is available. Here you insert the number which stands after the label 'Next:' into the table by clicking with the left mouse button in the upper row, these represent the orbit numbers. Each time you repeat this action the number after 'Next:' increases. Pressing the 'plus' and 'minus' button increases or decreases this number, respectively. After moving the mouse-pointer into the second line of the table behind the insertion label you can enter the number of the orbits to be inserted, these orbits are seperated by pressing 'return'. The 'cursor up', 'cursor down' and 'backspace' keys give you the possibility to edit your input. With 'done' you complete your reordering. The 'visualisation' button opens a window. This window enables you to define the width and the height of the table which appears once having pressed 'done'. In this table you place the points of your quiver (see `help preproj`). To put a point into the table just press the left mouse button. The number which is thus fitted in is the number visible after the 'next' label, it increases each time the mouse button is depressed. With 'plus' and 'minus' you can control which number is inserted.
  - Hint: zero is an entry which leaves an empty space later in the output as well as in the table, so leave some columns empty to seperate the calculated entries of the dimension-vector.

– The third area contains the output of the function `preproj`.

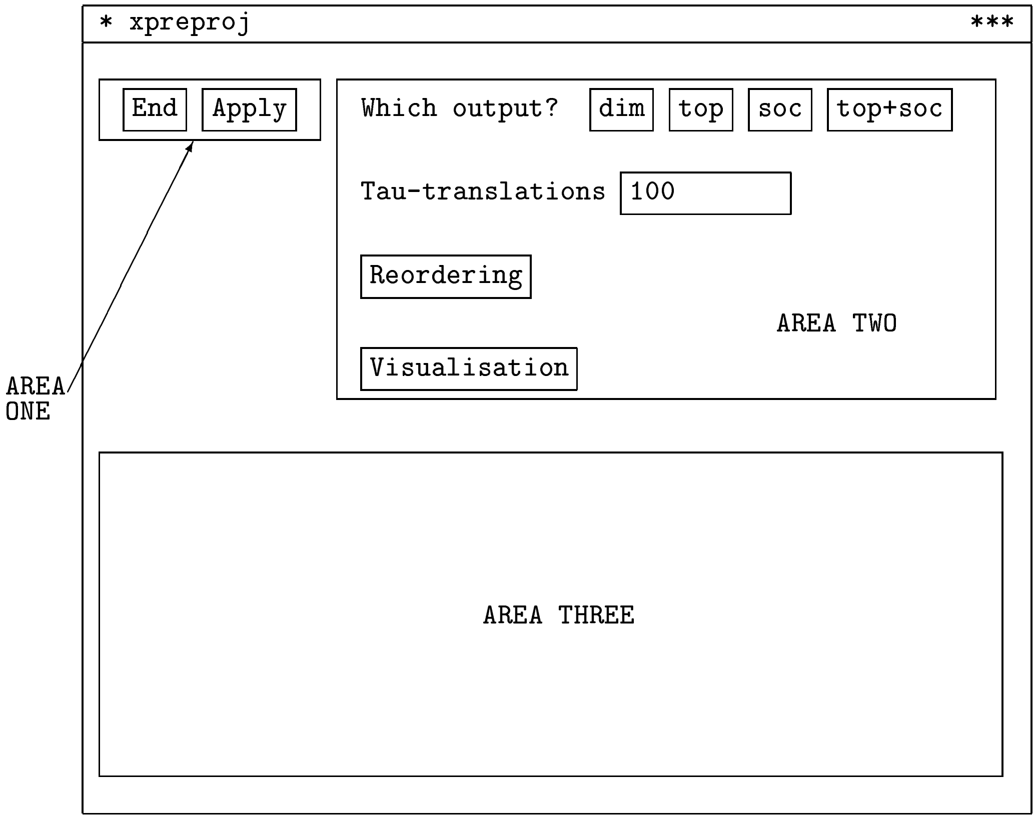
Examples:

```
'> read`xpreproj`;

> a:=[4,[[1,2,3,4]]];

                        a := [4, [[1, 2, 3, 4]]]

> xpreproj(a);
# then you receive a window:
```



SEE ALSO: `preproj`

4 The tubular package

Piotr Dowbor (Toruń) and Thomas Hübner (Paderborn)

The tubular - package contains functions dealing with K-theoretic methods for canonical algebras and weighted projective lines with special emphasis on tubular canonical algebras and the evaluation of 'telescoping functors'.

The package is written wholly in Maple and organised as a Maple library. It can thus also be used independently of the CREP system.

To make the package available read in the file `tubular.src` located in the CREP home directory using the command `read ‘tubular.src‘`. This should be followed by the command `with(tubular)` to make the functions of the package directly accessible.

4.1 TUBULAR - Program Package for treatment of tubular canonical algebras

Calling Sequence:

<function>(*args*)

Synopsis:

- TUBULAR provides functions and data related to the representation theory of (tubular) canonical algebras.
- The functions available are:
- for Lp-calculus:

```
&+        &-        &m        &=
&<>       &<=       &<        &>=
&>        NF        delta
```

- for K_0X -calculus:

```
rank_     degree_   slope_    Lp_shift
tau       tauinv    read_in   show
eulf      quaf      change    Cartan
&++       &==       &--       &mm
enter
```

- for tubular-calculus:

```
telescope   get_class   get_tub
R           S
```

- All of these functions are collected in a program package called "tubular". Therefore, prior to using any of those, please type the command "with(tubular)".
- For more information on a particular function see <function>.
- Some data are made available by means of global variables rather than via functions. See tubular/globalvars for further explaination.

- The general philosophy of the tubular program package is to set up an environment depending mainly on the chosen weight-sequence. Therefore, ahead of invoking any specific calculation, a w e i g h t s e q u e n c e will have to be entered using the procedure read_in. A chosen weight sequence remains in force until another weight-sequence has been entered (invoking read_in again). In particular, only one weight sequence can be valid at a time.

SEE ALSO: `read_in, tubular/datatypes, tubular/globalvars, tubular/Lp`

4.2 tubular/datatypes

Synopsis:

- In dealing with graded sheaf theory TUBULAR uses several specific data formats:
- elements of the grading group Lp are encoded as lists of t+1 integers (t=number of weights); with operations of Lp defined.
- Elements of K_0X are internally coded as vectors of Z^N (where N = ‘N$‘ equals the Z-rank of K_0X.) - For the convenience of the user however, the datatype "quiver" has been introduced: Using "show" instead of "print" allows to visualize classes in a "canonical algebra" shape (Cf. to help page for "show"). Several operations such as grade shifts, coxeter transformation, euler characteristic are user-accessible with arguments as well as results being entered using datatype "quiver". The user has the option to encode each class with respect to three different "canonical" bases: (Λ denotes the canonical algebra corresponding to the entered weight sequence.)
- the basis ‘@O‘ of indec. projective Λ-modules:
- $O, O(x_1), O(2x_1), \ldots, O((p_1-1)x_1), \ldots, \ldots, \ldots, O(x_t), \ldots, O((p_t-1)x_t), O(c)$
- the basis ‘@E‘ of simple Λ-modules:
- chosen subject to the same order as for the indec. projective Λ-modules above, where each indec. projective module is replaced by the corresponding(!) simple module.
- the "simple sheaves" basis ‘@S‘:
- $O, S_{1,1}, S_{1,2}, \ldots, S_{1,p_1-1}, \ldots, \ldots, \ldots, S_{t,1}, \ldots, S_{t,p_t-1}, S$
- where $S_{i,j}$ denotes the cokernel of $O(jx_i) \longrightarrow O((j+1)x_i)$.
- For the t u b u l a r canonical algebras there is a special datatype "tub_enc" encoding objects with respect to the tubular family and their position within it; one of the central aspects of the TUBULAR package is the conversion between this data type and ordinary class representation "quiver" described above.

SEE ALSO: `tubular/Lp, show, change, enter, get_class, get_tub`

4.3 tubular/Lp

Synopsis:

- Elements of the grading group Lp are encoded as lists of t+1 integers (t=number of weights, where the t+1 entry refers to the canonical element.) - For further details, refer to the help page for type/Lp .
- The following operations are defined (help is available for all of these):

`&+`	`&-`	`&m`	`&=`
`&<>`	`&<=`	`&<`	`&>=`
`&>`	`NF`	`delta`	

- Whenever arguments of Lp are entered, they need not be in normal form. However, if a result of an operation is of type Lp, it will usually be returned in normal form.

SEE ALSO: `type/Lp, tubular/datatypes, tubular/globalvars, Lp_shift`

4.4 tubular/globalvars

Synopsis:

- The tubular program package uses a variety of internal global variables in order to prepare the proper environment for calculation (this depending mainly on the current weight sequence entered by the user).
- Most internal global variables used by the tubular program package are string-variables of type 'name$'. This has been done in order to avoid interference with any variable names created during a maple session.
- Most of these variables are only considered for internal purposes and therefore bear no relevance for the user.
- The following predefined variables might be of interest also for passive usage, but should u n d e r n o c i r c u m s t a n c e s be directly changed by the user:

```
`t$`  -  the number of weights entered
`p$`  -  the current weight sequence (always sorted) as a Maple list
`pp$` -  the least common multiple of all weights
`N$`  -  the Z-rank of the Grothendieck group corresponding to coh X
`omega$`  -  the dualizing element of the grading group Lp, of type Lp
               (a type defined within this program package; cf. tubular/Lp)
`delta_omega$`  -  the degree of the dualizing element
```

SEE ALSO: `tubular, tubular/datatypes`

4.5 The Binary Operator &+

Synopsis:

- &+ is a binary operator with two operands x and y being of type Lp. Their sum is returned in normal form.

Example:

```
> read_in(3,3,4);
> x := [2,0,2,4];  y := [-1,0,0,-2];
> x &+ y;

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

SEE ALSO: `tubular/Lp, NF, &-, &m`

4.6 The Prefix Operator &-

Synopsis:

- &- is a unary operator with an operand x being of type Lp. The additive inverse of x is returned in normal form. Note that you must n o t use the ordinary minus sign instead!

Example:

```
> read_in(3,3,4);
> x := [2,0,-2,4];
> &- x;

                              [1,0,2,-6]
```

SEE ALSO: `tubular/Lp, NF, &+, &m`

4.7 The Binary Operator &m

Synopsis:

- &m is a binary operator, whose first operand is an integer z, and whose second operand is an element x of type Lp. "The product" z times x is returned in normal form.

Examples:

```
> read_in(3,3,4);
> x := [2,0,2,4];
> 8 &m x;

                              [1,0,0,41]

> (-8) &m x;

                              [2,0,0,-42]
```

```
> &-( 8 &m x);

                              [2,0,0,-42]
```

SEE ALSO: `tubular/Lp, NF, &+, &m`

4.8 The Binary Operator &=

Synopsis:

- &= is a binary operator with two operands x and y being of type Lp. It returns true, if x and y are equal, and false otherwise.
- This operation seems trivial, but please note that the equality of x and y may not be obvious at the first sight if at least one of x and y is not in normal form.

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> x &= y;

                              true

> x&= z;

                              false
```

SEE ALSO: `tubular/Lp, NF, &<, &>, &<=, &>=, &<>`

4.9 The Binary Operator &<>

Synopsis:

- &<> is a binary operator with two operands x and y being of type Lp. It returns true, if x and y are n o t equal, and false otherwise.
- This operation seems trivial, but please note that the equality of x and y may not be obvious at the first sight if at least one of x and y is not in normal form.

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> x &<> y;

                              false

> x &<> z;

                              true
```

SEE ALSO: `tubular/Lp, NF, &<, &>, &<=, &>=, &<>`

4.10 The Binary Operator &<=

Synopsis:

- &<= is a binary operator with two operands x and y being of type Lp. It returns true, if x is less than, or equal to y, and false otherwise.

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> x &<= y;

                              true

> x &<= z;

                              true
```

SEE ALSO: `tubular/Lp, NF, &<, &>, &<=, &>=, &<>`

4.11 The Binary Operator &>=

Synopsis:

- &>= is a binary operator with two operands x and y being of type Lp. It returns true, if x is greater than, or equal to y, and false otherwise.

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> x &>= y;

                              true

> x &>= z;

                              false
```

SEE ALSO: `tubular/Lp, NF, &<, &>, &<=, &>=, &<>`

4.12 The Binary Operator &<

Synopsis:

- &< is a binary operator with two operands x and y being of type Lp. It returns true, if x is less than, but not equal to y, and false otherwise.

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> x &< y;

                              false

> x &< z;

                              true
```

SEE ALSO: `tubular/Lp, NF, &<, &>, &<=, &>=, &<>`

4.13 The Binary Operator &>

Synopsis:

- &> is a binary operator with two operands x and y being of type Lp. It returns true, if x is greater than, but not equal to y, and false otherwise.

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> x &> y;

                              false

> x &> z;

                              false
```

SEE ALSO: `tubular/Lp, NF, &<, &>, &<=, &>=, &<>`

4.14 NF - normal form for an element of Lp;

Calling Sequence:
`NF(`x`)`
Parameters:
x – an arbitrary element of Lp; encoded as a list of t+1 integers.
Synopsis:

- `NF` computes the normal form of an arbitrary element of Lp; such that the i-th entry (i=1,..,t) l_i satisfies $0 \leq l_i < p_i$.

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> NF(x);
```

```
                    [0,0,2,-5]

> NF(z);

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

SEE ALSO: `tubular/datatypes, tubular/Lp`

4.15 delta - evaluate the delta-function on Lp.

Calling Sequence:
`delta(`x`)`
Parameters:
x – an arbitrary element of Lp; encoded as a list of t+1 integers.
Synopsis:

- The function $\delta : Lp \longrightarrow Z$ is a surjective morphism of groups defined by $\delta(x_i) = p/p_i$ and $\delta(c) = p$. (Here p denotes the lcm of the weights $p_1, \ldots, p_t$. For further details on the grading group Lp, refer to help pages for tubular/Lp and type/Lp .)

Examples:

```
> read_in(3,3,4);
> x := [-3,0,2,-4];  y := [0,0,2,-5];  z := [0,1,1,3];
> delta(x);

                    -54

> delta(z);

                    43
```

SEE ALSO: `tubular/datatypes, tubular/Lp, NF`

4.16 read_in - enter a valid weight sequence

Calling Sequence:
`read_in(`*p1,...,pt*`)`
Parameters:
p1,...,pt – positive integers (= weights)
Synopsis:

- read_in(p1,...,pt) invokes a variety of internal routines; setting internal variables (mostly matrices, but also resetting remember tables etc.) for various computations. "read_in" formally returns NULL. In case one of the four tubular weight sequences has been entered, additional variables concerning the use of telescope functors will be set.

- The weights will automatically be ordered s.t. $p_1 \leq p_2 \leq \ldots \leq p_t$.

- Note that a chosen weight sequence remains valid until another weight sequence has been entered (invoking read_in again); in particular, only one weight sequence can be valid at a time.

Examples:

```
> read_in(3,3,3);

> read_in(2,3,5);
```

SEE ALSO: `tubular/datatypes`

4.17 get_class - conversion from "tub_enc" to K_0X

Calling Sequence:

`get_class(`*[q,i,s,l]*`)`

Parameters:

q – the "slope" (a rational, or infinity)
i – number of an exceptional tube (i=1,..,t)
s – socle index (an integer, which is taken modulo p_i)
l – the quasi-length (height in corresponding tube)

Synopsis:

- Computes the K_0X -class from a given dataset of type "tub_enc" which encodes an indecomposable sheaf by a set of data relating to its position within the tubular family. (Cf. help page for type/tub_enc.)
- is only defined for one of the tubular weight-sequences, i.e. (2,3,6), (3,3,3), (2,2,2,2) and (2,4,4).
- get_class([q,i,s,l]) returns the Grothendieck-class of the coherent sheaf corresponding to the given data. The "inverse" procedure to get_class is get_tub.

Examples:

```
> read_in(3,3,3);

> get_class([7/4,1,2,5]);  show(");

                              [ @S               ]
                              [                  ]
                              [     -7   -13     ]
                              [                  ]
                              [     -6   -13     ]
                              [                  ]
                              [ 20  -6   -13  31 ]

> read_in(2,3,6);

> get_class([-41/3,3,0,4]);  show(");
```

```
[ @S                               ]
[                                  ]
[      -6                          ]
[                                  ]
[      -4   -8                     ]
[                                  ]
[ 12   -1   -3   -5   -7  -10  -16 ]
```

SEE ALSO: `show, telescope, get_tub, type/tub_enc`

4.18 get_tub - conversion from K_0X to "tub_enc"

Calling Sequence:
`get_tub(`*f*`)`

Parameters:
f – the K_0X -class of an object (of datatype quiver)

Synopsis:

- Computes the dataset (cf. help page for type/tub_enc) for a given K_0X -class of type quiver, representing an indecomposable sheaf.
- is only defined for one of the tubular weight-sequences, i.e. (2,3,6), (3,3,3), (2,2,2,2) and (2,4,4).
- get_tub(f) returns the dataset encoding the relevant tubular data for the given Grothendieck-class, in terms of the type "tub_enc". In a certain sense it is the "inverse" to the procedure "get_class".
- However, it is also possible to apply "get_tub" to some K_0X class of type quiver which has been entered directly using "enter" (cf. one of the examples below).
- Note that any class in K_0X representing an indecomposable sheaf must necessarily be a positive root of the quadratic form induced by the Euler-characteristic, or lie in the radical.
- If f is a positive root, then its encoding with respect to the tubular structure will be computed, and the result of type "tub_enc" will be returned.
- If f is positive lying in the radical, then a continuous family of sheaves (parametrized over X) will be mapped to the class f. In that case (cf. to the second example given below), in the encoding corresponding to the type "tub_ent" the expression "cont" will replace the second entry (normally indicating the tube-number), and the third entry (quasi-socle-index) will be set to zero. The quasi-length l given in the fourth entry will refer to a sheaf (with K_0 class f) lying in a string tube. Please note that in this case in each exceptional tube of rank p_i there will be p_i indecomposable sheaves of quasi-length $p_i \cdot l$ which are also represented by the class f in K_0X.

Examples:

```
> read_in(3,3,3);

> f := get_class([7/4,1,2,5]);  get_tub(f);
```

```
                                    [7/4, 1, 2, 5]
> g := get_class([8/7,1,0,6];   get_tub(g);

                                    [8/7,cont,0,6]

> h := enter([0,[0,0],[1,0],[0,0],0],`@S`):  show(h):

                                  [ @S           ]
                                  [              ]
                                  [     0   0    ]
                                  [              ]
                                  [     1   0    ]
                                  [              ]
                                  [  0   0   0   0 ]

> get_tub(h);

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

SEE ALSO: `show`, `enter`, `telescope`, `get_class`, `type/tub_enc`

4.19 type Lp - implementing the Lp-grading group.

Synopsis:

- Given a weight sequence $(p_1, \ldots, p_t)$, the Lp-group is the rank one abelian group generated by $x_1, \ldots, x_t$ subject to the relations
- $p_1 x_1 = p_2 x_2 = ... = p_t x_t =: c$.
- c is called the "canonical element" in Lp.
- Each element x in Lp has a unique presentation in "normal form":
- $x = l_1 x_1 + \ldots + l_t x_t + lc$,
- with integers $l_1, \ldots, l_t, l$ such that $0 \leq l_i < p_i$.
- These elements are encoded as lists with t+1 entries: $[l_1, \ldots, l_t, l]$. They need not be entered in normal form; so when entering such an element you are allowed to enter a list consisting of t+1 arbitrary integers:
- So, for instance, if the weight sequence entered is (2,3,5), then [1,-1,8,2] would encode the element $x_1 - x_2 + 8x_3 + 2c$. Note, however, that even if you wish to enter an element where you specify only the multiplicities of the generators - for instance $2x_1 - 7x_2 + 14x_3$ - you have to key in a list of t+1 (in our example t+1=4) integers, then setting the t+1 entry to 0 (so [2,-7,14,0] for our example; not(!) [2,-7,14]).
- The command "NF" computes the normal form corresponding to this element. - All operations defined on Lp (cf. to tubular/Lp for a complete list of these) return their results in normal form.

SEE ALSO: `tubular/Lp`, `Lp_shift`

4.20 type quiver - chosen for "pretty printing" classes in K_0X

Synopsis:

- Internally, classes in K_0X are handled as vectors with N coordinates. These vectors are changed to Maple-tables, which in turn are "pretty-printed" when invoking the "show" procedure."
- For further details, cf. to the help page for "show".

SEE ALSO: `show, enter, tubular/datatypes`

4.21 type grovect_enter - designed for entering arbitrary coordinates

Synopsis:

- For the users convenience, arbitrary coordinates (with respect to one of the three standard bases '@O', '@S' or '@E') are entered in a way imitating the branch structure of the canonical algebra.
- This type has been created to support the procedure "enter", cf. to the help page for "enter" for details. Actual operations in K_0X should be performed using the type "quiver".

SEE ALSO: `enter, show, get_tub, type/quiver`

4.22 type tub_enc - encoding an indecomposable sheaf with respect to its position within the tubular families.

Synopsis:

- For X being of tubular representation type, each indecomposable sheaf F lies in an AR-component which is a a homogeneous tube in the tubular family C_q (where q = slope(F)). Those sheaves lying in one of the exceptional tubes of rank p_i can be encoded by a discrete set of data. The list of these data is defined to be the type "tub_enc"; namely:

  ```
  [q,i,s,l]
  ```

  where:

  ```
  q - the "slope" (a rational, or infinity)
  i - number of an exceptional tube (i=1,..,t)
  s - quasi-socle index (an integer, which is taken modulo p_i)
  l - the quasi-length (height in corresponding tube)
  ```

- One of the central tasks of the tubular-routines (and the main application of the telescoping functors) is to relate this encoding to the presentation of a sheaf given by its class in the Grothendieck-group K_0X. (Cf. to "get_class" and "get_tub".)

Examples:

```
Assume  p = (2,3,6).

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

encodes the sheaf lying in C_{-4/3}, lying in the 2nd tube of rank
p_2 = 3, having quasi-socle indexed by 1, and quasi-length 3,
so its corresponding class in K_0 X lies in the radical.

        [infinity,3,0,1]

encodes the sheaf in C_{infinity}, in the 3rd tube of rank
p_3 = 6, having quasi-socle indexed by 0 and quasi-length 1,
so being quasi-simple.
```

SEE ALSO: `show`, `telescope`, `get_class`, `get_tub`

4.23 Cartan - compute Cartan matrix

Calling Sequence:
`Cartan()`

Parameters:

Synopsis:

- Renders the Cartan matrix for the canonical algebra corresponding to the currently entered weight sequence.
- works for any weight type (not only tubular).

Example:

```
> read_in(2,3,4);  Cartan();

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

SEE ALSO: read_in, tubular/globalvars

4.24 R - linear transformation corresponding to mutation R

Calling Sequence:
R()
Parameters:

Synopsis:

- Renders matrix encoding the linear tranformation induced by mutation R (with respect to standard basis '@S')
- works only for t u b u l a r weight type!

Example:

```
> read_in(2,3,6);  R();

                    [  1   3   2   2   1   1   1   1   1   6 ]
                    [                                        ]
                    [ -1  -2  -1  -1  -1   0  -1   0  -1  -3 ]
                    [                                        ]
                    [  0  -1  -1   0  -1   0   0  -1   0  -2 ]
                    [                                        ]
                    [ -1  -2  -2  -1  -1  -1   0  -1  -1  -4 ]
                    [                                        ]
                    [  0  -1  -1   0   0   0   0   0   0  -1 ]
                    [                                        ]
                    [  0  -1  -1  -1  -1   0   0   0   0  -2 ]
                    [                                        ]
                    [  0  -2  -1  -1  -1  -1   0   0   0  -3 ]
                    [                                        ]
                    [  0  -2  -2  -1  -1  -1  -1   0   0  -4 ]
                    [                                        ]
                    [ -1  -3  -2  -2  -1  -1  -1  -1   0  -5 ]
                    [                                        ]
                    [  1   4   3   2   2   1   1   1   1   7 ]
```

SEE ALSO: read_in, tubular/globalvars, S

4.25 S - linear transformation corresponding to mutation S

Calling Sequence:
S()
Parameters:

Synopsis:

- Renders matrix encoding the linear tranformation induced by mutation S (with respect to standard basis '@S')

- works only for t u b u l a r weight type!

Example:

```
> read_in(2,3,6);  S();

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

SEE ALSO: `read_in, tubular/globalvars, R`

4.26 show - "pretty"-prints type quiver

Calling Sequence:
`show(`*f*`)`

Parameters:
f – the K_0X -class of an object (of datatype quiver)

Synopsis:

- `show` pretty-prints the coordinates of a Grothendieck-class of type quiver; in the upper left corner, the corresponding basis is indicated (cf. procedure "change").

- The canonical configuration

```
          O(x1) --> O(2 x1) --> ... --> O((p1-1) x1)
          O(x2) --> O(2 x2) --> ... --> O((p2-1) x2)
O          ...        ...       ...          ...             O(c)
          O(xt) --> O(2 xt) --> ... --> O((pt-1) xt)
```

  consists of those vector bundle which are projective when considered as Λ-modules. The presentation of a class in K_0X using "show" is intended to somehow imitate this shape.

- In particular, using the basis '@E' of simple Λ-modules, "show" visualizes the dimension vector of a module.

Example:

```
> get_class([7/4,1,2,5]);  show(");

                              [ @S            ]
                              [               ]
                              [    -7  -13    ]
                              [               ]
                              [    -6  -13    ]
                              [               ]
                              [ 20 -6  -13 31 ]
```

SEE ALSO: `tubular/datatypes, change, enter`

4.27 enter - direct input of classes in K_0X

Calling Sequence:
`> enter(`*L*,*B_ind*`)`
Parameters:
L – a list (cf. below)
B_ind – the "Basis indicator", i.e. one of '@O', '@S' or '@E'.
Synopsis:

- In order to make entering a coordinate vector as simple as possible coordinates are entered as a list, where entries corresponding to branches (e.g. of the canonical configuration) are in turn entered as lists. - Thus the first parameter of "enter", the coordinate vector, is entered as a list of t+2 items, where the first and last items are integers, while items 2,..,t+1 are lists of integers of length $p_1 - 1, \dots, p_t - 1$.
- The second parameter specifies the chosen basis to which the coordinate entries shall relate. - Please observe the ordering of each basis as indicated below, meaning that you have to replace each member of the chosen basis by its corresponding coordinate.
- As the result, the corresponding K_0X class of type "quiver" is returned.
- for basis '@O': (i.e. the indec. projective Λ-modules)
- $[O, [O(x_1), O(2x_1), \dots, O((p_1-1)x_1)], \dots, \dots, \dots, [O(x_t), \dots, O((p_t-1)x_t)], O(c)]$
- for basis '@E': (i.e. the simple Λ-modules) choose the same order as for the indec. projective Λ-modules, indicated above, where each indec. projective module is replaced by the corresponding(!) simple module.
- for basis '@S': (i.e. "simple sheaves")
- $[O, [S_{1,1}, S_{1,2}, \dots, S_{1,p_1-1}], \dots, \dots, \dots, [S_{t,1}, \dots, S_{t,p_t-1}], S]$
- where $S_{i,j}$ denotes the cokernel of $O(jx_i) \longrightarrow O((j+1)x_i)$.

Example:

```
> read_in(2,3,6);

# The following coordinates correspond to the indec. projective O(3 x3)

>  L := [0,[0],[0,0],[0,0,1,0,0],0];  f := enter(L,‘@O‘);  show(f);

                    [ @O                  ]
                    [                     ]
                    [     0               ]
                    [                     ]
                    [     0   0           ]
                    [                     ]
                    [   0   0   0   1   0   0   0 ]
```

SEE ALSO: `tubular/datatypes, show, change`

4.28 change - change of K_0X -basis for objects of type quiver.

Calling Sequence:
`change`(*f*,*B_ind*)

Parameters:
f – the K_0X -class of an object (of datatype quiver)
B_ind – a basis-indicator, i.e. one of ‘@O‘,‘@S‘,‘@E‘

Example:

```
> show(f);

                    [ @S               ]
                    [                  ]
                    [     -2   -5      ]
                    [                  ]
                    [     -2   -5      ]
                    [                  ]
                    [  8  -3   -6  105 ]

> fE := change(f,‘@E‘); show(fE)

                    [ @E               ]
                    [                  ]
                    [     94   92      ]
                    [                  ]
                    [     94   92      ]
                    [                  ]
                    [ 97  95   92   89 ]
```

SEE ALSO: `tubular/datatypes, show, enter`

4.29 telescope

Calling Sequence:
`telescope(`q`)`
Parameters:
q – an arbitrary fraction
Synopsis:

- Computes the K_0X -isomorphism corresponding to the telescoping functor $C_\infty \longrightarrow C_q$, where q is an arbitrary fraction.
- Originally, this procedure is not designed for immediate use but invoked within get_tub and get_class.

Example:

```
> read_in(3,3,3);
> q := -71/4;  M := telescope(q);  print(M);

                    [  1    4    4    4    4    4    4    12 ]
                    [                                        ]
                    [  0   -1   -1   -2   -1   -2   -1   -4 ]
                    [                                        ]
                    [ -1   -3   -2   -3   -3   -3   -3   -8 ]
                    [                                        ]
                    [  0   -2   -1   -1   -1   -2   -1   -4 ]
                    [                                        ]
                    [ -1   -3   -3   -3   -2   -3   -3   -8 ]
                    [                                        ]
                    [  0   -2   -1   -2   -1   -1   -1   -4 ]
                    [                                        ]
                    [ -1   -3   -3   -3   -3   -3   -2   -8 ]
                    [                                        ]
                    [ -5  -19  -20  -19  -20  -19  -20  -59 ]
```

SEE ALSO: `show, get_tub, get_class`

4.30 tau - Auslander-Reiten-shift on K_0X

Calling Sequence:
`tau(`f`)`
Parameters:
f – an arbitrary class of type quiver
Synopsis:

- Computes the Auslander-Reiten shift of any object in K_0X of type "quiver".

Example:

```
> show(f1);
```

```
[ @S              ]
[                 ]
[     -2   -5     ]
[                 ]
[     -2   -4     ]
[                 ]
[  6  -2   -4   9 ]
```

```
> f2 := tau(f1);  show(f2);
```

```
[ @S              ]
[                 ]
[     -3   -4     ]
[                 ]
[     -2   -4     ]
[                 ]
[  6  -2   -4   9 ]
```

SEE ALSO: `tauinv, Lp_shift`

4.31 tauinv - inverse Auslander-Reiten-shift

Calling Sequence:
`tauinv(`*f*`)`
Parameters:
f – an arbitrary class of type quiver
Synopsis:

- Computes the inverse of the Auslander-Reiten shift of any object in K_0X of type "quiver".

Example:

```
> show(f2);

> f1 := tauinv(f2);  show(f1)
```

```
[ @S              ]
[                 ]
[     -3   -4     ]
[                 ]
[     -2   -4     ]
[                 ]
[  6  -2   -4   9 ]

[ @S              ]
[                 ]
[     -2   -5     ]
[                 ]
[     -2   -4     ]
[                 ]
[  6  -2   -4   9 ]
```

SEE ALSO: `tau, Lp_shift`

4.32 Lp_shift - shift of a K_0-class

Calling Sequence:

`Lp_shift(`f,x`)`

Parameters:

f – an arbitrary class of type quiver

x – an arbitrary element of type Lp.

Synopsis:

- Computes the shift of a K_0-class of type quiver by an arbitrary element of the grading group Lp.

Examples:

```
> read_in(3,3,3);
x := [2,0,1,-4];  f := get_class([4/3,2,0,1]);  show(f);

                              [ @S            ]
                              [               ]
                              [     -1  -2    ]
                              [               ]
                              [     -2  -3    ]
                              [               ]
                              [  3  -1  -2  5 ]

> f1 := Lp_shift(f,x);  show(f1);

                              [ @S             ]
                              [                ]
                              [     -1  -2     ]
                              [                ]
                              [     -2  -3     ]
                              [                ]
                              [  3  -1  -2  -4 ]
```

SEE ALSO: `tau, tauinv, tubular/datatypes, tubular/Lp, type/Lp`

4.33 rank_ - compute the rank of a K_0X-class

Calling Sequence:

`rank_(`f`)`

Parameters:

f – an arbitrary class of type quiver

Synopsis:

- Computes the rank of any K_0X -class of type quiver.

Example:

```
> read_in(3,3,3);  f := [-7/3,3,0,2]);  show(f);
```

```
                    [ @S              ]
                    [                 ]
                    [     -2   -4     ]
                    [                 ]
                    [     -2   -4     ]
                    [                 ]
                    [  6  -1   -4   1 ]
```

```
> rank_(f);

                    6
```

SEE ALSO: `degree_`, `slope_`, `get_class`, `tubular/datatypes`

4.34 degree_ - compute the degree of a K_0X-class

Calling Sequence:
`degree_(`*f*`)`
Parameters:
f – an arbitrary class of type quiver
Synopsis:

- Computes the degree of any K_0X -class of type quiver.

Example:

```
> read_in(3,3,3);  f := [-7/3,3,0,2]);  show(f);

                    [ @S              ]
                    [                 ]
                    [     -2   -4     ]
                    [                 ]
                    [     -2   -4     ]
                    [                 ]
                    [  6  -1   -4   1 ]

> degree_(f);

                    -14
```

SEE ALSO: `rank_`, `slope_`, `get_class`, `tubular/datatypes`

4.35 slope_ - compute the slope of a K_0X-class

Calling Sequence:
`slope_(`*f*`)`
Parameters:
f – an arbitrary class of type quiver
Synopsis:

- Computes the slope of any K_0X -class of type quiver.

Example:

```
> read_in(3,3,3);  f := [-7/3,3,0,2]);  show(f);

                          [ @S           ]
                          [              ]
                          [     -2  -4   ]
                          [              ]
                          [     -2  -4   ]
                          [              ]
                          [  6  -1  -4  1 ]

> slope_(f);

                           -7/3
```

SEE ALSO: rank_, degree_, get_class, tubular_datatypes

4.36 eulf - the euler-characteristic

Calling Sequence:
eulf(f_1, f_2)
Parameters:
f_1, f_2 – two classes of type quiver
Synopsis:

- Computes the euler-characteristic for any pair of K_0X -classes of type quiver.

Example:

```
> read_in(3,3,3);
> f1 := get_class([4/3,3,0,2]);  f2 := get_class([4/3,3,1,7]);
> show(f1);  show(f2);

                          [ @S           ]
                          [              ]
                          [     -2  -4   ]
                          [              ]
                          [     -2  -4   ]
                          [              ]
                          [  6  -1  -3  8 ]

                         [ @S              ]
                         [                 ]
                         [     -7  -14     ]
                         [                 ]
                         [     -7  -14     ]
                         [                 ]
                         [ 21  -7  -13  30 ]
```

```
> eulf(f1,f2);
                                        1

> eulf(f2,f1);
                                        0
```

SEE ALSO: quaf, get_class, tubular/datatypes

4.37 The Binary Operator &++

Synopsis:

- &++ is a binary operator with two operands of type quiver. It returns their sum in $K_0 X$. The use of this operator is not limited to tubular weight types.

Example:

```
> read_in(3,3,3);
> # assume f1, f2 defined
> show(f1);

                          [ @S           ]
                          [              ]
                          [     -2  -4   ]
                          [              ]
                          [     -2  -4   ]
                          [              ]
                          [  6  -1  -3  8 ]

> show(f2);

                         [ @S              ]
                         [                 ]
                         [     -7  -14     ]
                         [                 ]
                         [     -7  -14     ]
                         [                 ]
                         [ 21  -7  -13  30 ]


> f := f1 &++ f2;  show(f);

                         [ @S              ]
                         [                 ]
                         [     -9  -18     ]
                         [                 ]
                         [     -9  -18     ]
                         [                 ]
                         [ 27  -8  -16  38 ]
```

SEE ALSO: `&==`, `&--`, `&mm`, `tubular/datatypes`

4.38 The Prefix Operator &--

Synopsis:

- &-- is a prefix operator with an operand of type quiver. It returns the negative inverse in K_0X. The use of this operator is not limited to tubular weight types.

Example:

```
> read_in(3,3,3);
> # assume f1, f2 defined
> show(f1);
                               [ @S            ]
                               [               ]
                               [     -1   -1   ]
                               [               ]
                               [     -1   -2   ]
                               [               ]
                               [  2  -1   -2  3 ]
> show(f2);
                               [ @S            ]
                               [               ]
                               [     -2   -3   ]
                               [               ]
                               [     -2   -4   ]
                               [               ]
                               [  6  -2   -4  3 ]

> f3 := &-- f1;  show(f3);
                                [ @S           ]
                                [              ]
                                [     1   1    ]
                                [              ]
                                [     1   2    ]
                                [              ]
                                [ -2  1   2  -3 ]

> show(f1 &++ (&-- f2));
                                [ @S          ]
                                [             ]
                                [     1   2   ]
                                [             ]
                                [     1   2   ]
                                [             ]
                                [ -4  1   2  0 ]
```

SEE ALSO: `&==`, `&++`, `&mm`, `tubular/datatypes`

4.39 The Binary Operator &==

Synopsis:

- &== is a binary operator with two operands f1 and f2 of type quiver. It returns true, if and only if f1 and f2 are equal in K_0X. The use of this operator is not limited to tubular weight types.
- Note that the equality of two given classes f1 and f2 may not be visible with the naked eye if f1 and f2 are encoded with respect to different bases.

Example:

```
> read_in(3,3,3);
> # assume f1 defined
> show(f1);
```

```
                              [ @S             ]
                              [                ]
                              [     -1   -1    ]
                              [                ]
                              [     -1   -2    ]
                              [                ]
                              [  2  -1   -2  3 ]
```

```
> f2 := change(f1,`@E`);  show(f2);
```

```
                              [ @E              ]
                              [                 ]
                              [     -1   -2     ]
                              [                 ]
                              [      0   -1     ]
                              [                 ]
                              [  0   0   -1  -2 ]
```

```
> f1 &== f2;
```

```
                                    true
```

SEE ALSO: `&++, &--, &mm, change, tubular/datatypes`

4.40 &mm - scalar product of a K_0X class with an integer.

Calling Sequence:
`&mm`(*r*,*f*)
Parameters:
r – an integer
f – a K_0 class of type quiver
Synopsis:

- `&mm`(r, f) computes the scalar product of the K_0X class f with the integer r.

Example:

```
> read_in(3,3,3);
> # assume f1 defined
> show(f1);
```

```
                    [ @S             ]
                    [                ]
                    [     -1  -1     ]
                    [                ]
                    [     -1  -2     ]
                    [                ]
                    [  2  -1  -2  3  ]
```

```
> f2 := &mm(-8,f1);  show(f2);
```

```
                    [  @S               ]
                    [                   ]
                    [       8   8       ]
                    [                   ]
                    [       8  16       ]
                    [                   ]
                    [ -16   8  16  -24  ]
```

SEE ALSO: `&++, &==, &--, tubular/datatypes`

4.41 quaf - the quadratic form for a K_0X-class

Calling Sequence:
`quaf` (*f*)
Parameters:
f – a K_0X-class of type quiver
Synopsis:

- Computes the quadratic form for a K_0X -class of type quiver.

Example:

```
> read_in(3,3,3);
> f1 := get_class([4/3,3,0,2]);  f2 := get_class([4/3,3,0,3]);
> show(f1);
```

```
                    [ @S             ]
                    [                ]
                    [     -2  -4     ]
                    [                ]
                    [     -2  -4     ]
                    [                ]
                    [  6  -1  -3  8  ]
```

```
> quaf(f1);
                                    1

> show(f2);

                        [ @S              ]
                        [                 ]
                        [     -3   -6     ]
                        [                 ]
                        [     -3   -6     ]
                        [                 ]
                        [  9  -3   -6  13 ]

> quaf(f2);
                                    0
```

SEE ALSO: `eulf`, `get_class`, `tubular/datatypes`

5 The repetit package
Bernd Roggon (Bielefeld)

The repetit package is a supplement to CREP containing functions and data about repetitive algebras and iteratedly tilted algebras of type E_6, E_7, and E_8.

The package can be used only together with the CREP system. To use the package, you thus first have to start CREP. Reading in the file `repetit.src` will then make the functions of the repetit package available (provided the package is installed in the CREP home directory).

5.1 The CREP-package repetit for repetitive algebras and iteratedly tilted algebras

Calling Sequence:

`<function>`(*args*)

Synopsis:

- The CREP-package repetit gives access to data about repetitive algebras and iteratedly tilted algebras of type E_6, E_7 and E_8.
- The functions available are:

it_tilt_alg	it_tilted_num	it_tilt_test
repetitive	repetitive_alg	rep_alg_num
repetit_addrel		

- For more information on a particular function see `<function>`.
- See `repetit/datatypes` for information on the data formats used for iteratedly tilted algebras and repetitive algebras respectively.

5.2 repetit/datatypes

Synopsis:

- The Format for iteratedly tilted algebras: Iteratedly tilted algebras are represented using CREP's standard format for bound poset algebras
- The Format for repetitive algebras:

  A repetitive algebra R of the form

$$\begin{bmatrix} \ddots & & & & \\ \ddots & A & & & \\ & DA & A & & \\ & & DA & A & \\ & & & \ddots & \ddots \end{bmatrix}$$

  is represented by a triple $[a, b, c]$, where:

- a describes the quiver of a subalgebra B of R of the form

$$\begin{bmatrix} A & \\ DA & A \end{bmatrix}$$

  using CREP's data format for posets.
  The relations of this algebra then are the following:
  * All possible commutativity relations.
  * Zero relations for all paths that cannot be extended to a path leading from a vertex of the 'upper' algebra to the corresponding vertex of the 'lower' algebra.
- b is a set marking the vertices of B belonging to the 'lower' copy of A contained in B.
- c is a list of integers used as labels for the vertices of B such that corresponding vertices in the 'upper' and in the 'lower' copy of A in B get the same label.

(That way, the data $r = [a, b, c]$ can be displayed, using the function call

`drwpst`($r[1], r[2], r[2]$);)

Examples:

```
> r:=repetitive_alg(6,2);
[

[12, {[2, 1], [3, 2], [4, 1], [5, 1], [6, 4], [7, 3, 4, 5], [8, 7], [9, 8],

         [10, 7, 6], [11, 7], [12, 10]}                       ],

{7, 8, 9, 10, 11, 12}, [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]]

> drwpst(r[1],r[2],r[2]);
```

SEE ALSO: `repetitive_alg, drwpst`

5.3 it_tilt_alg - representatives of frames of iteratedly tilted algebras of type E.

Calling Sequence:
`it_tilt_alg`(n,m)
Parameters:
n – 6,7 or 8
m – a positive integer less than or equal to `it_tilted_num`(n)
Synopsis:

- `it_tilt_alg`(n, m) returns an algebra which belongs to the m-th frame of iteratedly tilted algebras of type E_n.
- the number of frames available may by obtained by calling the function `it_tilted_num`.

Examples:

```
> it_tilt_alg(6,2);
             [6, {[4, 3], [5, 2], [6, 1, 2, 3]}]
```

SEE ALSO: `it_tilted_num`, `it_tilt_test`

5.4 it_tilted_num - the numbers of frames of iteratedly tilted algebras of type E.

Calling Sequence:
`it_tilted_num(`n`)`
Parameters:
n – 6,7 or 8
Synopsis:

- `it_tilted_num`(n) returns the numbers of frames of iteratedly tilted algebras of type E_n.

Examples:

```
> it_tilted_num(8);

                                  21948
```

SEE ALSO: `it_tilt_alg`

5.5 it_tilt_test - test for a iteratedly tilted algebra of type E

Calling Sequence:
`it_tilt_test(`p`)`
Parameters:
p – a bound poset algebra
Synopsis:

- `it_tilt_test`(p) tests whether the bound poset algebra p is iteratedly tilted of type E_n. If so, the function will return the value `true` otherwise the value `false`.

Examples:

```
> p:=[6, {[3, 2], [4, 3], [5, 2], [6, 1, 2]}, {[4, 2]}];

           p := [6, {[3, 2], [4, 3], [5, 2], [6, 1, 2]}, {[4, 2]}]

> it_tilt_test(p);
                                   true
```

SEE ALSO: `it_tilt_alg`

5.6 repetitive - the repetitive algebra of a bound poset algebra

Calling Sequence:
`repetitive(`p`)`
Parameters:
p – a bound poset algebra
Synopsis:

- `repetitive(`p`)` returns the repetitive algebra of the bound poset algebra p.

Examples:

```
> a:=[3,{[1,2],[2,3]},{[1,3]}];

                      a := [3, {[2, 3], [1, 2]}, {[1, 3]}]

> repetitive(a);
     [[6, [[1, 2], [2, 3, 4], [3, 5], [4, 5], [5, 6], [6]]], {1, 2, 3},

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

> ra:=repetitive(a);
     [[6, [[1, 2], [2, 3, 4], [3, 5], [4, 5], [5, 6], [6]]], {1, 2, 3},

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

> drwpst(ra[1],ra[2],ra[3]);

> ra:=repetit_addrel(ra);
    [[6, {[4, 5], [3, 5], [5, 6], [2, 3, 4], [1, 2]}, {[1, 3], [4, 6]}],

        {1, 2, 3}, [1, 2, 3, 1, 2, 3]]

> drwpst(ra[1],ra[2],ra[3]);
```

SEE ALSO: `repetit/datatypes`, `repetit_addrel`

5.7 repetitive_alg - representatives of classes of repetitive algebras of type E

Calling Sequence:
`repetitive_alg(`n,m`)`
Parameters:
n – 6,7 or 8
m – a positive integer less than or equal to `rep_alg_num`(n)
Synopsis:

- `repetitive_alg(`n, m`)` returns an algebra belonging to the m-th class of repetitive algebras of type E_n.
- the number of classes available may by obtained by calling the function `rep_alg_num`.

Examples:

```
> r:=repetitive_alg(6,1);
[

[12, {[7, 6], [4, 2, 3], [5, 3], [6, 1, 3], [8, 4], [9, 4, 5, 6], [10, 8, 9],

        [11, 9], [12, 7, 9]}                          ],

{7, 8, 9, 10, 11, 12}, [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]]

> drwpst(r[1],r[2],r[3]);
```

SEE ALSO: `rep_alg_num`, `repetit/datatypes`, `drwpst`

5.8 rep_alg_num - the numbers of classes of repetitive algebras of type E

Calling Sequence:
`rep_alg_num` (n)

Parameters:
n – 6,7 or 8

Synopsis:

- `rep_alg_num`(n) returns the numbers of classes of repetitive algebras of type E_n.

Examples:

```
> rep_alg_num(8);

                                    598
```

SEE ALSO: `repetitive_alg`

5.9 repetit_addrel - include relations for a repetitive algebra

Calling Sequence:
`repetit_addrel`(p)

Parameters:
p – a repetitive algebra

Synopsis:

- `repetit_addrel` includes zero-relations into the representation $p = [a, b, c]$ used to describe a repetitive algebra. (See `repetit/datatypes` for details on this representation.) Thus `repetit_addrel`(p) returns a triple $[a', b, c]$, where a is a bound poset algebra, endowned with the relations described in `repetit/datatypes`.

Examples:

```
> repetitive_alg(6,1);

[ [12, {[4, 2, 3], [5, 3], [6, 1, 3], [7, 6], [8, 4], [9, 4, 5, 6],

    [10, 8, 9], [11, 9], [12, 7, 9]}   ],

{7, 8, 9, 10, 11, 12}, [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]]

> repetit_addrel(");

[ [12, {[4, 2, 3], [5, 3], [6, 1, 3], [7, 6], [8, 4], [9, 4, 5, 6],

    [10, 8, 9], [11, 9], [12, 7, 9]},

{[9, 1], [9, 2], [7, 3], [8, 3], [11, 4], [12, 4], [10, 5], [12, 5], [10, 6],

     [11, 6]}   ],

{7, 8, 9, 10, 11, 12}, [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]]
```

SEE ALSO: `repetit/datatypes`, `repetitive_alg`

6 The tpe package

Rainer Nörenberg (Bielefeld)

The tpe package is a supplement to CREP containing functions and data about completely separating tame poset algebras containing algebras of type $\tilde{E}_6$, $\tilde{E}_7$ or $\tilde{E}_8$.

The package can be used only together with the CREP system. To use the package, you thus first have to start CREP. Reading in the file `tpe.src` will then make the functions of the tpe package available (provided the package is installed in the CREP home directory).

6.1 The CREP-package tpe for completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras.

Calling Sequence:

`<function>`(*args*)

Synopsis:

- The CREP-package tpe gives access to data about completely separating tame poset algebras containing a full convex proper subalgebra tame concealed of type $\tilde{\mathrm{E}}_6$, $\tilde{\mathrm{E}}_7$ or $\tilde{\mathrm{E}}_8$.
- The functions available are:

  tpe_alg   tpe_num   tpe_test
- For more information on a particular function see `<function>`.

6.2 tpe_alg - representatives of frames of completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras.

Calling Sequence:

`tpe_alg`(n,m)

Parameters:

n – 8,9,10,11,12,13,14, or 15

m – a positive integer less than or equal to `tpe_num`(n)

Synopsis:

- `tpe_alg`(n, m) returns an algebra which belongs to the m-th frame of completely separating tame poset algebras with n vertices containing a full convex proper subalgebra tame concealed of type $\tilde{\mathrm{E}}_6$, $\tilde{\mathrm{E}}_7$ or $\tilde{\mathrm{E}}_8$.
- The number of frames available may by obtained by calling the function `tpe_num`.

Examples:

```
> tpe_alg(8,3);
  [8, {[1, 2, 3], [2, 4, 6], [3, 5, 6], [4, 7], [5, 7], [6, 7, 8], [7], [8]}]
```

SEE ALSO: `tpe_num, tpe_test`

6.3 tpe_num - the numbers of frames of completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras.

Calling Sequence:
`tpe_num(`n`)`
Parameters:
n – 8,9,10,11,12,13,14, or 15
Synopsis:

- `tpe_num(`n`)` returns the number of frames of completely separating tame poset algebras containing a full convex proper subalgebra tame concealed of type $\tilde{\mathrm{E}}_6$, $\tilde{\mathrm{E}}_7$ or $\tilde{\mathrm{E}}_8$.

Examples:

```
> tpe_num(8);

                                  28
```

SEE ALSO: `tpe_alg`

6.4 tpe_test - test for completely separating tame poset algebras containing $\tilde{\mathrm{E}}_6$-, $\tilde{\mathrm{E}}_7$- or $\tilde{\mathrm{E}}_8$-algebras.

Calling Sequence:
`tpe_test(`p`)`
Parameters:
p – a bound poset algebra
Synopsis:

- `tpe_test(`p`)` tests whether the bound poset algebra p is a completely separating tame poset algebras containing a full convex proper subalgebra tame concealed of type $\tilde{\mathrm{E}}_6$, $\tilde{\mathrm{E}}_7$ or $\tilde{\mathrm{E}}_8$. If so, the function will return a pair $[n, m]$ such that p and the algebra given by the call of `tpe_alg(`n, m`)` have the same frame. Otherwise the value `false` will be returned.

Examples:

```
> p:=[8,{[1,2,3],[2,4,6],[3,5,6],[4,7],[5,7],[6,7,8],[7],[8]}];

p :=

    [8, {[1, 2, 3], [2, 4, 6], [3, 5, 6], [4, 7], [5, 7], [6, 7, 8], [7], [8]}]

> tpe_test(p);
                                [8, 3]
```

SEE ALSO: `tpe_alg`