Documentation on the NMO package

Version 1.0

7 January 2010

Randall E. Cone

Randall E. Cone Email: rcone@math.vt.edu

Address: Virginia Tech
Blacksburg, VA, 24060, USA

Acknowledgements

Contents

Chapter 1

NMO

1.1 Introduction

What follows is a description of the largely experimental project of providing arbitrary monomial orderings to the GBNP package. The addition of the orderings comes in the form of a library, and a patch to GBNP; the patching process being called at the GBNP user's discretion.

More precisely, after a user creates a monomial ordering via the NMO library functions, a routine is called which overwrites the two GBNP functions "LtNP" and "GtNP". In GBNP, these latter two functions are explicitly length-lexicographic monomial comparison functions, and are used in GBNP's Gröbner Basis routines. Therefore NMO allows for the creation of arbitrary monomial ordering comparison functions, which, after the patching process, will be used by GBNP in place of its native comparison functions.

NMO is an acronym for Noncommutative Monomial Orderings. Such orderings play a key role in research surrounding noncommutative Gröbner basis theory; see [3], [4]. This package is geared primarily toward the use and study of noncommutative (associative) free algebras with identity, over computational fields. We have done our best to write code that treats a more general class of algebras, but the routines have not been as extensively tested in those cases. Users of the package are encouraged to provide constructive feedback about this issue or any others; we have open ears to ways to better these research tools.

Flexibility in the creation and use of noncommutative monomial orderings has been our guiding principle in writing NMO. For example, two (or more) orderings can be chained together to form new orderings. It should be noted, however, that efficiency has also been considered in the design of NMO for commonly used monomial orderings for noncommutative rings (e.g. length left-lexicographic). That is to say, some monomial orderings that occur regularly in the study of noncommutative algebras have already been included in NMO.

Throughout this chapter, methods and functions are generally classed as *External* and *Internal* routines. *External* routines are methods and functions that will be most useful to the average user, and generally work directly with native GAP algebraic objects. *Internal* routines usually concern backend operations and mechanisms, and are often related to operations involving *NP representations* of GAP algebraic elements, or they are related to attributes of monomial orderings. Many examples of basic code use are provided; with some examples following the reference material for the functions or methods involved.

1.2   NMO Files within GBNP

Per the GAP package standard, NMO library code is read in via the file `gbnp/read.g`. The following gives brief descriptions of each of the files loaded by `gbnp/read.g`, all of which reside in the `gbnp/lib/nmo/` subdirectory:

- `ncalgebra.gd`

  Sets up some nice categories and filters in GAP.

- `ncordmachine.g*`

  Code for creating the new GAP family of noncommutative monomial orderings, as well as its attending (internal) machinery.

- `ncorderings.g*`

  Sets up actual noncommutative monomial orderings. This is where some specific example routines for monomial orderings are included. The less-than functions determining monomial orderings should be collected here, e.g. the length left-lexicographic ordering is here.

- `ncinterface.g*`

  These files provide the interface to comparison routines for determining equivalence, less-than, and greater-than comparison between two algebraic elements under a given NMO ordering.

- `ncutils.g*`

  Helpful utility routines, such as: patching GBNP for use with an NMO ordering, unpatching GBNP, as well as a 'String' routine for elements of an associative algebra not already covered in GAP.

There is a documentation directory in `gbnp/doc/nmo` wherein the GAPDoc source for this chapter may be found.

Finally, there is an examples directory in `gbnp/doc/examples/nmo` where the plain GAP source can be found for the examples in the Quickstart section of this chapter.

1.3   Quickstart

This Quickstart assumes you've already installed the GBNP package in its proper home. If that's yet to be done, please see the GBNP package manual for installation instructions.

If the user wishes, cutting and pasting the commands which directly follow the GAP prompt `gap>` is a good way to become familiar with NMO via the examples below. Alternatively, code for the following examples may be found in `gbnp/doc/examples/nmo/example0*.g`.

This Quickstart covers specific use of the NMO package's functionality as pertaining to computing noncommutative Gröbner bases for various examples. There are NMO user-level routines beyond these Gröbner basis applications that may be of interest, all of which are documented in later sections.

1.3.1   NMO Example 1

Example 1 is taken from Dr. Edward Green's paper "Noncommutative Gröbner Bases, and Projective Resolutions", and is referenced as "Example 2.7" there; please see [3] for more information.

Load the GBNP package with:

```
——————————————— Example ———————————————
gap> LoadPackage("gbnp");
true
```

Create a noncommutative free algebra on 4 generators over the Rationals in GAP:

```
——————————————— Example ———————————————
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");
<algebra-with-one over Rationals, with 4 generators>
```

Label the generators of the algebra:

```
——————————————— Example ———————————————
gap> a := A.a; b := A.b; c := A.c; d := A.d;
(1)*a
(1)*b
(1)*c
(1)*d
```

Set up our polynomials, and convert them to GBNP NP format:

```
——————————————— Example ———————————————
gap> polys := [c*d*a*b-c*b,b*c-d*a];
[ (-1)*c*b+(1)*c*d*a*b, (1)*b*c+(-1)*d*a ]
gap> reps := GP2NPList(polys);
[ [ [ [ 3, 4, 1, 2 ], [ 3, 2 ] ], [ 1, -1 ] ],
          [ [ [ 4, 1 ], [ 2, 3 ] ], [ -1, 1 ] ] ]
```

Compute the Gröbner basis via GBNP using its default (length left-lexicographic) ordering; that is, without patching GBNP with an NMO ordering:

```
——————————————— Example ———————————————
gap> gbreps := Grobner(reps);;
gap> gb := NP2GPList(gbreps,A);
[ (1)*d*a+(-1)*b*c, (1)*c*b*c*b+(-1)*c*b ]
```

Create length left-lexicographic ordering, with generators ordered: a < b < c < d. Note: this is the default ordering of generators by NMO, if none is provided:

```
——————————————— Example ———————————————
gap> ml := NCMonomialLeftLengthLexOrdering(A);
NCMonomialLeftLengthLexicographicOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
```

Patch GBNP with the ordering ml, and then run the same example. We should get the same answer as above:

```
——————————————— Example ———————————————
gap> PatchGBNP(ml);
LtNP patched.
GtNP patched.
gap> gbreps := Grobner(reps);;
gap> gb := NP2GPList(gbreps,A);
[ (1)*d*a+(-1)*b*c, (1)*c*b*c*b+(-1)*c*b ]
```

Create a Length-Lexicographic ordering on the generators such that d < c < b < a:

```
—————————————————— Example ——————————————————
gap> ml2 := NCMonomialLeftLengthLexOrdering(A,[4,3,2,1]);
NCMonomialLeftLengthLexicographicOrdering([ (1)*d, (1)*c, (1)*b, (1)*a ])
```

Compute the Gröbner basis with respect to this new ordering on the same algebra:

```
—————————————————— Example ——————————————————
gap> PatchGBNP(ml2);
LtNP patched.
GtNP patched.
gap> gbreps2 := SGrobner(reps);;
gap> gb2 := NP2GPList(gbreps2,A);
[ (1)*b*c+(-1)*d*a, (1)*c*d*a*b+(-1)*c*b, (1)*d*a*d*a*b+(-1)*d*a*b,
  (1)*c*d*a*d*a+(-1)*c*d*a, (1)*d*a*d*a*d*a+(-1)*d*a*d*a ]
```

1.3.2   NMO Example 2

This example is the same as Example 1 above, except that the length and left-lexicographic orderings are created independently and then chained to form the usual length left-lexicographic ordering. Hence, all results should be the same. Note: we assume from this point forward in all further examples that GBNP is loaded.

Create a noncommutative free algebra on 4 generators over the Rationals, label, and set up the example:

```
—————————————————— Example ——————————————————
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");;
gap> a := A.a;; b := A.b;; c := A.c;; d := A.d;;
gap> polys := [c*d*a*b-c*b,b*c-d*a];;
gap> reps := GP2NPList(polys);;
```

Create left-lexicographic ordering with a < b < c < d:

```
—————————————————— Example ——————————————————
gap> lexord := NCMonomialLeftLexicographicOrdering(A);
NCMonomialLeftLexicographicOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
```

Create a length ordering on monomials in *A*, with ties broken by the lexicographic order `lexord`:

```
—————————————————— Example ——————————————————
gap> lenlex := NCMonomialLengthOrdering(A,lexord);
NCMonomialLengthOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
```

Patch GBNP and proceed with our example:

```
—————————————————— Example ——————————————————
gap> PatchGBNP(lenlex);;
LtNP patched.
GtNP patched.
gap> gbreps := Grobner(reps);;
gap> gb := NP2GPList(gbreps,A);
[ (1)*d*a+(-1)*b*c, (1)*c*b*c*b+(-1)*c*b ]
```

Now, proceed similarly, with the lexicographic order such that d < c < b < a:

```
─────────── Example ───────────
gap> lexord2 := NCMonomialLeftLexicographicOrdering(A,[4,3,2,1]);
NCMonomialLeftLexicographicOrdering([ (1)*d, (1)*c, (1)*b, (1)*a ])
gap> lenlex2 := NCMonomialLengthOrdering(A,lexord2);
NCMonomialLengthOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
gap> PatchGBNP(lenlex2);;
LtNP patched.
GtNP patched.
gap> gbreps2 := Grobner(reps);;
gap> gb2 := NP2GPList(gbreps2,A);
[ (1)*b*c+(-1)*d*a, (1)*c*d*a*b+(-1)*c*b, (1)*d*a*d*a*b+(-1)*d*a*b,
  (1)*c*d*a*d*a+(-1)*c*d*a, (1)*d*a*d*a*d*a+(-1)*d*a*d*a ]
```

An important point can be made here. Notice that when the `lenlex2` length ordering is created, a lexicographic (generator) ordering table is assigned internally to the ordering since one was not provided to it. This is merely a convenience for lexicographically-dependent orderings, and in the case of the length order, it is not used. Only the lex table for `lexord2` is ever used. Some clarification may be provided in examining:

```
─────────── Example ───────────
gap> HasNextOrdering(lenlex2);
true
gap> NextOrdering(lenlex2);
NCMonomialLeftLexicographicOrdering([ (1)*d, (1)*c, (1)*b, (1)*a ])
gap> LexicographicTable(NextOrdering(lenlex2));
[ (1)*d, (1)*c, (1)*b, (1)*a ]
```

1.3.3 NMO Example 3

Example 3 is taken from the book "Ideals, Varieties, and Algorithms", ([2], Example 2, p. 93-94); it is a commutative example.

First, we set up the problem and find a Gröbner basis with respect to the length left-lexicographic ordering implicitly assumed in GBNP:

```
─────────── Example ───────────
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"x","y","z");;
gap> x := A.x;; y := A.y;; z := A.z;; id := One(A);;
gap> polys := [ x^2 + y^2 + z^2 - id, x^2 + z^2 - y, x-z,
>               x*y-y*x, x*z-z*x, y*z-z*y];;
gap> reps := GP2NPList(polys);;
gap> gb := Grobner(reps);;
gap> NP2GPList(gb,A);
[ (1)*z+(-1)*x, (1)*x^2+(-1/2)*y, (1)*y*x+(-1)*x*y,
  (1)*y^2+(2)*x^2+(-1)*<identity ...> ]
```

The example, as presented in the book, uses a left-lexicographic ordering with z < y < x. We create the ordering in NMO, patch GBNP, and get the result expected:

```
─────────── Example ───────────
gap> ml := NCMonomialLeftLexicographicOrdering(A,[3,2,1]);
NCMonomialLeftLexicographicOrdering([ (1)*z, (1)*y, (1)*x ])
gap> PatchGBNP(ml);
LtNP patched.
GtNP patched.
```

```
gap> gb := Grobner(reps);;
gap> NP2GPList(gb,A);
[ (1)*z^4+(1/2)*z^2+(-1/4)*<identity ...>, (1)*y+(-2)*z^2, (1)*x+(-1)*z ]
```

1.3.4 NMO Example 4

Example 4 was taken from page 339 of the book "Some Tapas of Computer Algebra" by A.M. Cohen, H. Cuypers, H. Sterk, [1]; it also appears as Example 6 in the GBNP example set.

A noncommutative free algebra on 6 generators over the Rationals is created in GAP, and the generators are labeled:

```
━━━━━━━━━━━━━━━━━━━━ Example ━━━━━━━━━━━━━━━━━━━━
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d","e","f");;
gap> a := A.a;; b := A.b;; c := A.c;; d := A.d;; e := A.e;; f := A.f;;
```

Set up list of noncommutative polynomials:

```
━━━━━━━━━━━━━━━━━━━━ Example ━━━━━━━━━━━━━━━━━━━━
gap> polys := [ e*a, a^3 + f*a, a^9 + c*a^3, a^81 + c*a^9 + d*a^3,
>               a^27 + d*a^81 + e*a^9 + f*a^3, b + c*a^27 + e*a^81 + f*a^9,
>               c*b + d*a^27 + f*a^81, a + d*b + e*a^27, c*a + e*b + f*a^27,
>               d*a + f*b, b^3 - b, a*b - b*a, a*c - c*a, a*d - d*a,
>               a*e - e*a, a*f - f*a, b*c - c*b, b*d - d*b, b*e - e*b,
>               b*f - f*b, c*d - d*c, c*e - e*c, c*f - f*c, d*e - e*d,
>               d*f - f*d, e*f - f*e
> ];;
gap> reps := GP2NPList(polys);;
```

Create a length left-lex ordering with the following (default) ordering on the generators a < b < c < d < e < f:

```
━━━━━━━━━━━━━━━━━━━━ Example ━━━━━━━━━━━━━━━━━━━━
gap> ml := NCMonomialLeftLengthLexOrdering(A);
NCMonomialLeftLengthLexicographicOrdering([ (1)*a, (1)*b, (1)*c, (1)*d,
    (1)*e, (1)*f ])
```

Patch GBNP and compute the Gröbner basis with respect to the ordering `ml`:

```
━━━━━━━━━━━━━━━━━━━━ Example ━━━━━━━━━━━━━━━━━━━━
gap> PatchGBNP(ml);
LtNP patched.
GtNP patched.
gap> gb := Grobner(reps);;
gap> NP2GPList(gb,A);
[ (1)*a, (1)*b, (1)*d*c+(-1)*c*d, (1)*e*c+(-1)*c*e,
  (1)*e*d+(-1)*d*e, (1)*f*c+(-1)*c*f,
  (1)*f*d+(-1)*d*f, (1)*f*e+(-1)*e*f ]
```

1.4 Orderings

This section describes the current orderings built into the GAP package NMO, and describes some of the internals of the machinery involved.

1.4.1 Internals

The orderings portion of NMO is divided codewise into the files `ncordmachine.gd`, `ncordmachine.gi` and `ncorderings.gd`, `ncorderings.gi`. The former file pair contains code to set up the machinery to create new monomial orderings on noncommutative algebras, whereas the latter sets up actual orderings. We will first describe the creation and use of length lexicographic ordering, afterward describing more of the details of the new GAP family 'NoncommutativeMonomialOrdering'.

The NMO package was built with the mindset of allowing great flexibility in creating new monomial orderings on noncommutative algebras. All that is required to install a new ordering is to create two GAP functions that determine less-than comparisons (one non-indexed, and one indexed) and then call `InstallNoncommutativeMonomialOrdering` with the comparison functions as arguments. The comparison functions should be written to compare simple lists of integers, these lists representing monomials as in GBNP's 'NP' format, or the letter representation format in GAP (see "The External Representation for Associative Words" in the GAP reference manual). An example follows the description of the function `InstallNoncommutativeMonomialOrdering`.

A bit of explanation is due here to address the added complexity introduced by requiring that two functions (`<function>`, `<function2>`) need be supplied to `InstallNoncommutativeMonomialOrdering` to create an ordering. The first function `<function>` should be responsible for comparing two given monomial list representations in their unadultered forms. The second, indexed, function `<function2>` should be capable of using a provided index list corresponding to an order on generators, based on a different lexicographic ordering. This accomplishes something worthwhile: two orderings with different lexicographic tables can be applied to the same algebra in GAP.

One more caveat: `InstallNoncommutativeMonomialOrdering` will create a default lexicographic table for all orderings, despite whether or not it will be used in the comparison function. It does this only out of convenience and ease of use.

For example, in the creation of the following left-lex ordering, which is installed via the `InstallNoncommutativeMonomialOrdering` function, a default ordering of a < b < c is created for `ml` even though an ordering on the generators is not provided:

```
———————————— Example ————————————
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c");
<algebra-with-one over Rationals, with 3 generators>
gap> lexord := NCMonomialLeftLexicographicOrdering(A);
NCMonomialLeftLexicographicOrdering([ (1)*a, (1)*b, (1)*c ])
```

Notice next that when an ordering on the generators is provided, it is utilized in the creation of the ordering:

```
———————————— Example ————————————
gap> lexord2 := NCMonomialLeftLexicographicOrdering(A,[2,3,1]);
NCMonomialLeftLexicographicOrdering([ (1)*b, (1)*c, (1)*a ])
```

1.4.2 Internal Routines

▷   `InstallNoncommutativeMonomialOrdering(<string>, <function>, <function2>)`
(function)

Given a name `<string>`, a direct comparison function `<function>`, and an indexed comparison function `<function2>`, `InstallNoncommutativeMonomialOrdering` will install a monomial ordering function to allow the creation of a monomial ordering based on the provided functions.

For example, we create a length ordering by setting up the two comparison functions, choosing a name for the ordering type and then calling `InstallNoncommutativeMonomialOrdering`.

```
———————————————————————————— Example ————————————————————————————
    gap> f1 := function(a,b,aux)
    >    return Length(a) < Length(b);
    > end;
    function( a, b, aux ) ... end
    gap> f2 := function(a,b,aux,idx)
    >    return Length(a) < Length(b);
    > end;
    function( a, b, aux, idx ) ... end

    DeclareGlobalFunction("lenOrdering");
    InstallNoncommutativeMonomialOrdering("lenOrdering",f1,f2);
```

Now we create an ordering based on this new function, and make some simple comparisons. (Note: we are passing in an empty `aux` table since it is not being used. Also, the comparison function is the non-indexed version since we determined no lex order on the generators):

```
———————————————————————————— Example ————————————————————————————
    gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c");
    <algebra-with-one over Rationals, with 3 generators>
    gap> ml := lenOrdering(A);
    lenOrdering([ (1)*a, (1)*b, (1)*c ])
    gap>
    gap> LtFunctionListRep(ml)([1,2],[1,1,1],[]);
    true
    gap> LtFunctionListRep(ml)([1,1],[],[]);
    false
```

▷ `IsNoncommutativeMonomialOrdering(<obj>)`   (Category)

A noncommutative monomial ordering is an object representing a monomial ordering on a noncommutative (associative) algebra. All NMO orderings are of this category.

▷ `LtFunctionListRep(<NoncommutativeMonomialOrdering>)`   (attribute)

Returns the low-level comparison function used by the given ordering. The function returned is a comparison function on the external representations (lists) for monomials in the algebra.

▷ `NextOrdering(<NoncommutativeMonomialOrdering>)`   (attribute)

Returns the next noncommutative monomial ordering chained to the given ordering, if one exists. It is usually called after a `true` determination has been made with a `HasNextOrdering` call.

▷ `ParentAlgebra(<NoncommutativeMonomialOrdering>)`   (attribute)

Returns the parent algebra used in the creation of the given ordering.

▷ `LexicographicTable(<NoncommutativeMonomialOrdering>)` (attribute)

Returns the ordering of the generators of the `ParentAlgebra`, as specified in the creation of the given ordering.

▷ `LexicographicIndexTable(<NoncommutativeMonomialOrdering>)` (attribute)

Returns the ordering of the generators of the `ParentAlgebra`, as specified in the creation of the given ordering.

An example here would be useful. We create a length left-lexicographic ordering on an algebra `A` with an order on the generators of $b < a < d < c$. Then in accessing the attributes via the atrributes above we see how the list given by `LexicographicIndexTable` indexes the ordered generators:

```
─────────────────────── Example ───────────────────────
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");
<algebra-with-one over Rationals, with 4 generators>
gap> ml := NCMonomialLeftLengthLexOrdering(A,2,4,1,3);
NCMonomialLeftLengthLexicographicOrdering([ (1)*b, (1)*d, (1)*a, (1)*c ])
gap>  LexicographicTable(ml);
[ (1)*b, (1)*d, (1)*a, (1)*c ]
gap> LexicographicIndexTable(ml);
[ 3, 1, 4, 2 ]
```

The index table shows that the generator a is the third in the generator ordering, b is the least generator in the ordering, c is the greatest and d the second least in order.

▷ `LexicographicPermutation(<NoncommutativeMonomialOrdering>)` (attribute)

Experimental permutation based on the information in `LexicographicTable`, could possibly be used to make indexed versions of comparison functions more efficient. Currently only used by the NMO built-in ordering `NCMonomialLLLTestOrdering`.

▷ `AuxilliaryTable(<NoncommutativeMonomialOrdering>)` (attribute)

An extra table carried by the given ordering which can be used for such things as weight vectors, etc.

▷ `OrderingLtFunctionListRep(<NoncommutativeMonomialOrdering>)` (operation)

▷ `OrderingGtFunctionListRep(<NoncommutativeMonomialOrdering>)` (operation)

Given a noncommutative monomial ordering, `OrderingLtFunctionListRep` and `OrderingLtFunctionListRep` return functions which compare the 'list' representations (NP representations) of two monomials from the ordering's associated parent algebra. These functions are not typically accessed by the user.

1.4.3   Provided Orderings

▷ `NCMonomialLeftLengthLexicographicOrdering(<algebra>, <list>)` (function)

Given a free algebra *A*, and an optional ordered (possibly partial) ordered list of generators for the algebra *A*, `NCMonomialLeftLengthLexicographicOrdering` returns a noncommutative length lexicographic ordering object. If an ordered list of generators is provided, its order is used in creation

of the ordering object. If a list is not provided, then the ordering object is created based on the order of the generators when the free algebra *A* was created.

Note: the synonym `NCMonomialLeftLengthLexOrdering` may also be used.

▷ `NCMonomialLengthOrdering(<algebra>)`   (function)

Given a free algebra *A*, `NCMonomialLengthOrdering` returns a noncommutative length ordering object. Only the lengths of the words of monomials in *A* are compared using this ordering.

▷ `NCMonomialLeftLexicographicOrdering(<algebra>, <list>)`   (function)

Given a free algebra *A*, and an optional ordered (possibly partial) ordered list of generators for the algebra *A*, `NCMonomialLeftLexicographicOrdering` returns a simple noncommutative left-lexicographic ordering object.

▷ `NCMonomialCommutativeLexicographicOrdering(<algebra>, <list>)`   (function)

Given a free algebra *A*, and an optional ordered (possibly partial) ordered list of generators for the algebra *A*, `NCMonomialCommutativeLexicographicOrdering` returns a commutative left-lexicographic ordering object. Under this ordering, monomials from *A* are compared using their respective commutative analogues.

▷ `NCMonomialWeightOrdering(<algebra>, <list>, <list2>)`   (function)

Given a free algebra *A*, an ordered (possibly partial) ordered `<list>` of generators for the algebra *A*, and a `<list2>` of respective weights for the generators, `NCMonomialWeightOrdering` returns a noncommutative weight ordering object.

1.4.4   Externals

All user-level interface routines in the descriptions following allow for the comparison of not only monomials from a given algebra with respect to a given ordering, but also compare general elements from an algebra by comparing their leading terms (again, with respect to the given ordering). These routines are located in the files `ncinterface.gd` and `ncinterface.gi`.

1.4.5   External Routines

▷ `NCLessThanByOrdering(<NoncommutativeMonomialOrdering>, <a>, <b>)`   (operation)

Given a `<NoncommutativeMonomialOrdering>` on an algebra *A* and $a, b \in A$, `NCLessThanByOrdering` returns the (boolean) result of $a < b$, where $<$ represents the comparison operator determined by `<NoncommutativeMonomialOrdering>`.

▷ `NCGreaterThanByOrdering(<NoncommutativeMonomialOrdering>, <a>, <b>)`   (operation)

Given a `<NoncommutativeMonomialOrdering>` on an algebra *A* and $a, b \in A$, `NCLessThanByOrdering` returns the (boolean) result of $a > b$, where $>$ represents the comparison operator determined by `<NoncommutativeMonomialOrdering>`.

▷ `NCEquivalentByOrdering(<NoncommutativeMonomialOrdering>, <a>, <b>)`   (operation)

Given a `<NoncommutativeMonomialOrdering>` on an algebra A and $a, b \in A$, `NCLessThanByOrdering` returns the (boolean) result of $a = b$, where $=$ represents the comparison operator determined by `<NoncommutativeMonomialOrdering>`.

Some examples of these methods in use:

```
———————————————————————— Example ————————————————————————
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"x","y","z");
<algebra-with-one over Rationals, with 3 generators>
gap> x := A.x;; y := A.y;; z := A.z;; id := One(A);;
gap> w1 := x*x*y;; w2 := x*y*x;; w3 := z*x;;

gap> ml := NCMonomialLeftLengthLexOrdering(A);
NCMonomialLeftLengthLexicographicOrdering([ (1)*x, (1)*y, (1)*z ])

gap> ml2 := NCMonomialLengthOrdering(A);
NCMonomialLengthOrdering([ (1)*x, (1)*y, (1)*z ])

gap> ml7 := NCMonomialWeightOrdering(A,[1,2,3],[1,1,2]);
NCMonomialWeightOrdering([ (1)*x, (1)*y, (1)*z ])

gap> ml8 := NCMonomialWeightOrdering(A,[2,3,1],[1,1,2]);
NCMonomialWeightOrdering([ (1)*y, (1)*z, (1)*x ])

gap> #  Left length-lex ordering, x<y<z:
gap> NCEquivalentByOrdering(ml,w1,w2);
false
gap> #  Length ordering:
gap> NCEquivalentByOrdering(ml2,w1,w2);
true
gap> NCEquivalentByOrdering(ml2,w3,w2);
false
gap> # Weight ordering ( z=2, x=y=1 ):
gap> NCEquivalentByOrdering(ml7,w1,w2);
true
gap> NCEquivalentByOrdering(ml7,w3,w2);
true
gap> # Weight ordering ( z=2, x=y=1 ), different lex:
gap> NCEquivalentByOrdering(ml8,w1,w2);
true
gap> NCEquivalentByOrdering(ml8,w3,w2);
true
```

▷ NCSortNP(*`<NoncommutativeMonomialOrdering>`, `<list>`, `<function>`*)   (operation)

Given a `<list>` of NP 'list' representations for monomials from a noncommutative algebra, and an NP comparison (ordering) function `<function>`, NCSortNP returns a sorted version of `<list>` (with respect to the NP comparison function `<function>`). The sort used here is an insertion sort, per the recommendation from [5].

1.4.6   Flexibility vs. Efficiency

We recall that `InstallNoncommutativeMonomialOrdering` completes a list of generators if only a partial one is provided. An example will provide clarity here. It is given in terms of length-lex,

but the generator list completion functionality is identical for any NMO ordering. Note: If at all possible, users are encouraged to use the default ordering on generators as it is more efficient than the indirection inherent in sorting via the indexed list `LexicographicIndexTable`. Here is the example showing the flexibility in requiring only a partial list of the ordering on generators:

```
———————————————————— Example ————————————————————
  gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");
  <algebra-with-one over Rationals, with 4 generators>
  gap> ml2 := NCMonomialLeftLengthLexOrdering(A,[3,1]);
  NCMonomialLeftLengthLexicographicOrdering([ (1)*c, (1)*a, (1)*b, (1)*d ])
  gap> LexicographicTable(ml2);
  [ (1)*c, (1)*a, (1)*b, (1)*d ]
```

1.5 Utility Routines

1.5.1 GBNP Patching Routines

▷ PatchGBNP(*<NoncommutativeMonomialOrdering>*)   (operation)

▷ UnpatchGBNP()   (function)

Let `<NoncommutativeMonomialOrdering>` be a monomial ordering (on an algebra *A*). `PatchGBNP` overwrites the GBNP Global functions `LtNP` and `GtNP` with the less-than and greater-than functions defined for `<NoncommutativeMonomialOrdering>`. The purpose of such a patching is to force GBNP to use `<NoncommutativeMonomialOrdering>` in its computation of a Gröbner basis. `UnpatchGBNP()` simply restores the `LtNP` and `GtNP` functions to their original state. The examples in Quickstart section are more illustrative, but here is an example of the use of the patching routines above:

```
———————————————————— Example ————————————————————
  gap> A := FreeAssociativeAlgebraWithOne(Rationals,"x","y","z");
  <algebra-with-one over Rationals, with 3 generators>
  gap> ml := NCMonomialLeftLexicographicOrdering(A,3,2,1);
  NCMonomialLeftLexicographicOrdering([ (1)*z, (1)*y, (1)*x ])
  gap> PatchGBNP(ml);
  LtNP patched.
  GtNP patched.
  gap> UnpatchGBNP();
  LtNP restored.
  GtNP restored.
```

1.5.2 Printing Routine

▷ String(*<obj>*)   (operation)

GAP seems to be currently lacking a method to convert an object from a free associative ring to a string version of the same object. This routine fills that gap.
    Example (after loading NMO via the GBNP package):

```
———————————————————— Example ————————————————————
  gap> A := FreeAssociativeAlgebraWithOne(ZmodpZ(19),"x","y");
  <algebra-with-one over GF(19), with 2 generators>
```

```
gap> x := A.x; y := A.y;
(Z(19)^0)*x
(Z(19)^0)*y
gap> IsString(String(x^2+x*y*x));
true
gap> String(x^2+x*y*x);
"(Z(19)^0)*x^2+(Z(19)^0)*x*y*x"
```

Example (before loading NMO via the GBNP package):

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━ Example ━━━━━━━━━━━━━━━━━━━━━━━━━━━
gap>  A := FreeAssociativeAlgebraWithOne(ZmodpZ(19),"x","y");
<lgebra-with-one over GF(19), with 2 generators>
gap> x := A.x; y := A.y;
(Z(19)^0)*x
(Z(19)^0)*y
gap>  String(x^2+x*y*x);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for 'String' on 1 arguments called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk>
```

References

[1] Arjeh M. Cohen, Hans Cuypers, and Hans Sterk. *Some Tapas of Computer Algebra*, volume 4 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Heidelberg, 1999. 9

[2] David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, second edition, 1997. An introduction to computational algebraic geometry and commutative algebra. 8

[3] Edward L. Green. Noncommutative Gröbner bases and projective resolutions. In *Computational Methods for Representations of Groups and Algebras*, pages 29–60. Birkhäuser, 1999. (Essen 1997). 4, 5

[4] Teo Mora. An introduction to commutative and noncommutative Gröbner bases. *Theoretical Computer Science*, 134(1):131–173, Nov 1994. 4

[5] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002. 14