Translating MAGMA code to GAP

Alexander Hulpke
(Colorado State University)

hulpke@math.colostate.edu

Introduction

In a pattern that is common to mathematical software, programs for computational group theory have gone through several generations of design. Starting over 50 years ago with single-purpose software that might even require recompilation to work on a different problem, programs evolved to accept general input, became parts of program packages in which one program could work on the output of another, and finally became routines in a larger system such as Cayley [6] or GAP 2 [9] that provided a general environment both in the form of a programming language and in the provision of many basic mathematical routines. Based on experienced shortcomings in the initial designs, these systems evolved to their current incarnations MAGMA [4] and (via the intermediate GAP 3 [11]) the system GAP 4 [8].

Each of these design steps often led to a fundamental re-implementation of most algorithms. Doing so not only permitted these algorithms to work on more general input (say, going from rationals to finite fields and algebraic extensions), but also often incorporated newer programming paradigms and thus resulted in better code.

This iterative process seems to have reached a plateau, as witnessed by the fact that the current systems, having seen first versions in the late '90s, are now reaching an age where a human would be considered as "grown up." This stabilization is a reflection of the maturity of software design in general (and thus the diminishing return of changes), as well as of the enormous cost that a re-implementation would impose. It therefore seems likely that the fundamental engines of the current systems will be with us for the foreseeable future.

Sophisticated algorithm packages

The availability of a broad set of functions for groups in multiple representations made it possible to implement more sophisticated construction and classification algorithms. For example, consider the following results of the last two decades:

- Construction of groups of small (or innocuous) order following [2].
- Constructions of groups of order p^7 based on [10].
- Lists of maximal subgroups of classical groups based on [5].
- Composition tree for matrix groups as described in [1].

These algorithms build on the classical routines of computational group theory and share many characteristics that distinguish them from more basic routines:

- The algorithms work inherently with groups in different representations and thus cannot be implemented stand-alone, but only in a system.
- They need to distinguish many cases, and when re-implementing from scratch or translating ad-hoc it would be easy to make accidental mistakes. Indeed this has happened in the past.
- The time-critical parts of the calculation sit mostly – far beyond even the traditional 80/20% flip of code size versus time spent – in library routines called by the algorithm.
- The functionality provided would be useful to have (say to check other results) in multiple systems, even if it was based on the same code.
- The time required for a re-implementation is prohibitive, in particular considering that such work currently would not be judged as an "original" result in a CV.

Because of these obstacles to a re-implementation, C. LEEDHAM-GREEN encouraged [7] the community to work on a translation of the matrix group composition tree routines of [1] to GAP. This has been the motivation for the work described here.

The purpose of this note is to introduce a translation tool, the *converter*, that aids in the translation of code from MAGMA to GAP. Besides its immediate use for conversion of code, it might be of interest more generally to indicate possibilities and obstacles to such an approach.

The program is available from `github.com/hulpke/mgmconvert`. It consists of a single GAP file and provides the command

`MagmaConvert(`*infile*`[,` *outfile*`])`

(with default output to the screen) that reads in a MAGMA file and outputs the translation.

The Code Conversion

The purpose of the converter is as a tool to ease the translation of code. It produces output in valid GAP syntax that a human can modify with moderate effort into actual runnable code, but it does not aim to produce output that will run immediately. Attempting to do so would need to deal with different argument requirements or return value conventions in library functions (or the unavailability of some functions), different paradigms for constructing objects (for example, a permutation in MAGMA has to belong to a particular group, while in GAP it happily exists on its own), and different paradigms for doing certain tasks (e.g. in MAGMA one can test a matrix group for irreducibility, in GAP this would be the property of its natural module).

In this setup the converter is fundamentally different from "classical" compilers (such as FORTRAN to C). It also is a reflection of the fact that it would be futile to translate a highly optimized time-critical implementation (such as lattice reduction) one-to-one.

Instead, it still will be necessary for a competent (in GAP as well as with understanding of the underlying algorithm) programmer to go through and edit the produced code. In examples tried between 5 and 40% of lines required such an edit. Crucially, however, these edits do not involve the control structure or actual calculations, but easily identified and isolated instances of calling e.g. library or constructor functions. Compared with re-implementing the routine from a theoretical description, this will be a magnitude faster. (The examples below indicate the amount of work required.)

An underlying assumption is that the code to be converted is itself not time-critical (but that all criticality is in fact in the operations called by the code). It therefore is plausible to translate control structures one-to-one without considerations that were made to satisfy conditions of internal data structures (e.g. the internal way a matrix is stored implies the proper way to nest row/column loops), or of particular functions being available. If this was the case a more substantial rewrite of the resulting code would become necessary.

The conversion process also only translates functions to functions and will not try to install methods or even declare operations – such actions require global system knowledge and are left to the human programmer.

The converter itself is written in GAP, a choice as as a high-level language (providing e.g. string and list functions) the author is most familiar with. Clearly a language such as Python would have been equally appropriate, but neither would there have been a substantial benefit in choosing an inherently third language for this purpose.

Running the converter is a one-time task and therefore not time critical, nor is memory use an issue.

Basic Setup

While the syntax of GAP and MAGMA seems to be sufficiently similar on first view to allow for translation by simple string replacement, a look at more complicated code quickly indicates that such a strategy will only go so far. Instead, similar to what a compiler would do, the converter reads in the MAGMA code, and translates it.

Following standard compilation techniques, in the first step the MAGMA program is read in and fed to a tokenizer that recognizes keywords and separates the text of comments (which are translated into special comment tokens).

The resulting token sequence then is read by a parser and translated into a parse tree. This is the most complicated part of the process as the MAGMA language seems to allow the programmer a certain amount of latitude. (Examples: declaring functions alternately as `myfct:=function(...)` or `function myfct(...)`; An `error if` construct that duplicates a normal if-not condition-then error; Using the keyword `rec` also as an identifier name; A `where` construct that assigns temporary variables within another construction.) Indeed, the author has been unable to find a description of the MAGMA language in BNF or a similar format; this also prevented an approach using standard compiler generators.

The current version of the parser can still be stumped by very complicated constructs (such as a list selection with two simultaneous variables). In such cases some lines need to be treated separately by hand. This however is only a minor issue in practice.

Every node in the parse tree is stored simply as a GAP record with components indicating type, parameters, and links to children nodes. While this is a gratuitous use of memory for the benefit of easy processing, this is on modern computers not a concern for any plausible input. (Similar design decisions of trading memory for ease of coding were made e.g. when parsing expressions by order of precedence.)

Finally a further routine takes this parse tree and creates functionally equivalent GAP code from it. This routine can use the whole tree (or even multiple trees for files in a packages) to identify, for example, local variables for declaration (MAGMA seems to declare these variables implicitly, but GAP will by default treat

them as global), or identify package-local variables. The MAGMA language constructs are translated to equivalent (combinations of) GAP constructs, in some cases with added comment warnings indicating that this is not an exact equivalent. Calls to functions are translated straightforwardly to a function call with the same arguments, with only certain library function names being replaced (such as `Modexp` to `PowerMod`). It remains the human operator's responsibility to either change arguments and return values appropriately, or to provide (see below) syntactic equivalents in GAP for particular MAGMA functions.

We now describe a number of common issues and their treatment in the following sections:

Lists-based data types

While GAP presents to the user a uniform syntax of list-based objects that can represent homogeneous and inhomogeneous lists, sets, ranges, as well as vectors and matrices, MAGMA utilizes different constructs for different categories of objects. (Translating in the other direction thus would be substantially harder as the compiler would need to deduce the appropriate list type.) This is easily translated (namely all to the same type of GAP-list), but it can be necessary to inform GAP, by a separate function call, about particular properties or storage formats such an object is supposed to possess (such as being a set, or a finite field matrix being stored in compact form). The general format of a list data type also requires in GAP to make certain objects *immutable* (for example lists within a larger list so that being sorted can be stored) for efficiency reasons. In either of these cases appropriate conversion commands (either to make immutable or to replace an object with a copy that can be changed) will need to be added by hand.

Object constructors and attributes

The biggest difference between systems occurs when building new internal objects, or accessing attributes of internal objects, as internal data types, access to such information, and the parameters of constructors tend to differ. In some frequent situations – for example the construction of matrices from a description of entries or by composing blocks – this merits the addition of utility functions that provide a GAP constructor using the MAGMA syntax. (Here again the assumption is used that the code is not time-critical, as such a constructor emulation will take time.)

Since MAGMA requires every object to lie in a particular domain, a frequent construct is a cast `D!obj`. This is translated to `obj*FORCEOne(D)` – the `FORCE` indicating that it is a cast that requires editing, while a multiplication with an appropriate one would be the GAP way of accomplishing such a cast.

A dual issue arises with substructures, whose category (group, ring, vector space, ...) MAGMA will deduce from the parent structure. It also will allow generation from any collection of structures and elements, while GAP distinguishes e.g. `Ring`, `Group`, `Subgroup` and `ClosureSubgroup`. GAP also is less flexible in combining, say, 3 subgroups and 5 elements as generating entities, but requires different functions for different constructions. The `<...>` construct therefore is translated into a generic `SubStructure` with a comment `TODO CLOSURE` indicating that this might need to be converted to a generation or a closure operation.

Language-specific constructs

MAGMA has a number of language constructs that have no direct equivalent in GAP and therefore are difficult to translate:

`where` allows to implicitly declare variables, as in:

```
[x,x^2] where x:=SquareRoot(y)
  where y=PrimitiveElement(GF(q));
```

The parser simply puts such implicit assignment *after* the command with an comment indicating that it should be placed before.

`select` has a value depending on a condition:

```
a:= x=5 select 1 else 2;
```

This currently gets translated to a function call

```
a:=SELECT(x=5,1,2);
```

(an appropriate `SELECT` function would be easily written). Such a construct requires the evaluation of the *if* and the *else* branch in every case which could potentially come at significant cost, however. It therefore is best replaced by hand with a proper `if...then` construct.

Infix operators: Infix operators in MAGMA (such as `cat`, `diff`, or `meet`) are translated to appropriate function calls in GAP. This can lead to parentheses becoming obsolete.

Multi-argument returns: A MAGMA function may return multiple objects, while GAP code always returns only one object. This is translated into the convention that a multi-object return happens in form of a record with components `val1` etc. (Returned objects to be ignored would be assigned to the _ variable, such assignments then simply can be deleted.)

In some cases (such as `IsSquare`, which returns a truth value and a square root) equivalent functionality in GAP is implemented by treating `fail` as an exceptional return value. This needs to be translated by hand.

Implicit Assignments: The MAGMA `exists` construct returns a truth value, thus is equivalent to `ForAny` in GAP. But at the same time it

can also assign the existing value to a variable. This is translated by making the boolean condition become an assignment to the variable: `if x:=ForAny(...)`, indicating the need for manual intervention.

Call-by-reference: GAP holds any composite object (an object that is neither a small integer, a finite field element, or a boolean value) only in form of a pointer to the actual object, i.e. universally uses call-by-reference. In MAGMA instead, a function that is to change a list needs to be handed a pointer to the list, in the form of the '$\sim$' operator. This operator is translated (to be cautious with translating paradigm changes) by prepending the word `TILDE`, resulting in translated constructs such as `Append(TILDElist,a)` that can be quickly processed (e.g. in this example be changed to `Add(list,a)`) by a text editor's search-and-replace facilities.

Function Declarations: A function in MAGMA may be declared with typed input and output. This is not required in GAP and the type part is ignored (actually the output type is moved into a comment). Likewise, GAP does not require a declaration of a record format, hence such a declaration is replaced by a dummy string.

Similarly, forward declarations or particular function installations (such as `intrinsic`) are ignored, respectively translated to simple identifier assignments.

Package translation

When translating a package of files, often one file defines a function that is used within another file, but still is only local to the package. The command `Project(`*`directory`*`)` assumes that a subfolder `magma` contains source code that is to be translated. It reads in all files that end in a suffix `.m`, uses the information to identify which functions are provided by the package for use within other package files, and writes translated versions of output into a subfolder `translation`. Global identifiers get a symbol `@` appended to allow for use of GAP's name space functionality. The routine also provides a list of file-dependencies that makes it easier to determine which (of many) files would be required for a particular functionality.

A library of basic utility functions

In some cases (for example when creating objects or accessing their attributes) the data structures used by both systems are so different that existing code would need to be completely rewritten from scratch. A typical case is MAGMA's construction of matrices that allows, for example, to provide a list of entry positions and values. If such constructions are used frequently is makes sense to provide GAP wrapper functions that essentially provide an argument/data structure conversion. Included with the MAGMA converter is a growing file `util.g` that provides such routines, if function names differ this is translated automatically by the converter.

Examples and Results

As with any utility program, the proof is in the pudding: The author has so far used the converter to produce raw-translates for several contributed MAGMA packages, containing in total about 80 source files and 50000 lines of code. It turned out that in about 1 in 1000 lines of source code constructs existed that were not parseable (typical reasons were complicated constructions with iterated `where`; `exists` statements or list constructors indexed over multiple variables simultaneously; or abuse of keywords such as `rec` as a variable). All of these could be handled by minor edits of the source code, or by an immediate hand translation. While not (yet) universal, this shows that the parsing process is good enough to be used in practice.

As for the GAP code produced, Example 1 shows a straightforward case where only minor human rearrangement is needed, while Example 2 shows the case where more substantial programmer interaction is necessary, and displays both the auto-translated and hand-edited results.

Further Developments

It is expected that practical use of the converter will indicate further areas that merit improvements.

The step that would buy most impact in reducing the amount of final hand-translation work would be to offer more and better substitutes for MAGMA function calls, for example allowing as well for different arguments or by providing more utility functions. Doing so would however increase the risk of accidentally overlooking subtle differences in the declarations, respectively add much wrapper code for the sake of avoiding minor reformatting of a handful of source code lines.

The author would be interested in any feedback on using the converter and observed shortcomings.

This work was supported by the Simons' Foundation under Collaboration Grant 244502, whose support is gratefully acknowledged.

Literatur

[1] H. Bäärnhielm, D. Holt, C.R. Leedham-Green, E.A. O'Brien. A practical model for computation with matrix groups. *J. Symbolic Comput.*, 68(part 1):27–60, 2015.

[2] H.U. Besche, B. Eick, E.A. O'Brien. A millennium project: constructing small groups. *Internat. J. Algebra Comput.*, 12(5):623–644, 2002.

[3] W. Bosma, J. Cannon. Discovering Mathematics with Magma. Springer, 2006.

MAGMA code:

```
intrinsic CSOPlus(d:: RngIntElt, q:: RngIntElt) -> GrpMat
{Conformal special orthogonal group of plus type}
  local W, X, Y, Z, gens, hd;
  require IsEven(d) : "Argument 1 must be even";
  require IsPrimePower(q) : "Argument 2 is not a prime power";
  if IsEven(q) then
    if GCD(d,q-1) ne 1 then
      return S where S := sub< SL(d,q) | SOPlus(d,q),
        ScalarMatrix(d,w^p) >
  where w := PrimitiveElement(GF(q))
    where p := (q-1) div GCD(d,q-1);
      else return S where S := SOPlus(d,q);
    end if;
  end if;

  Z := ScalarMatrix(GF(q),d,w) where w:=PrimitiveElement(GF(q));
  hd := d div 2;
  X := GOMinusSO(d,q,1);
  Y := NormGOMinusGO(d,q,1);
  //Normaliser in SL is generated by SO together with elements
  //X^x Y^y Z^z with x(q-1)/2 + yd/2 + zd = 0 mod q-1
  W := Matrix(Integers(),4,1,[(q-1) div 2, hd, d, q-1]);
  N := Nullspace(W);
  gens := [ X^n[1] * Y^n[2] * Z^n[3] : n in Generators(N) ];
  return sub< SL(d,q) | SOPlus(d,q), gens >;
end intrinsic;
```

Translated GAP code, converter result:

```
CSOPlus:=function(d,q)
#  -> ,GrpMat  Conformal special orthogonal group of plus type
local N,S,W,varX,Y,varZ,gens,hd,p,w;
  if not IsEvenInt(d) then Error("Argument 1 must be even"); fi;
  if not IsPrimePower(q) then Error("Argument 2 is not a prime power"); fi;
  if IsEvenInt(q) then
    if Gcd(d,q-1)<>1 then
      return S;
      #  #T WHERE -- MOVE BEFORE PREVIOUS LINE
      S:=SubStructure(SL(d,q),SOPlus(d,q),#TODO CLOSURE
        ScalarMatrix(d,w^p));
      #  #T WHERE -- MOVE BEFORE PREVIOUS LINE
      w:=PrimitiveElement(GF(q));
      #  #T WHERE -- MOVE BEFORE PREVIOUS LINE
      p:=QuoInt((q-1),Gcd(d,q-1));
    else
      return S;
      #  #T WHERE -- MOVE BEFORE PREVIOUS LINE
      S:=SOPlus(d,q);
    fi;
  fi;
  varZ:=ScalarMatrix(GF(q),d,w);
  #  #T WHERE -- MOVE BEFORE PREVIOUS LINE
  w:=PrimitiveElement(GF(q));
  hd:=QuoInt(d,2);
  varX:=GOMinusSO(d,q,1);
  Y:=NormGOMinusGO(d,q,1);
  #  Normaliser in SL is generated by SO together with elements
  #  X^x Y^y Z^z with x(q-1)/2 + yd/2 + zd = 0 mod q-1
  W:=MatrixByEntries(Integers(),4,1,[QuoInt((q-1),2),hd,d,q-1]);
  N:=Nullspace(W);
  gens:=List(Generators(N),n->varX^n[1]*Y^n[2]*varZ^n[3]);
  return SubStructure(SL(d,q),SOPlus(d,q),#TODO CLOSURE
    gens);
end;
```

Example 1: *A* MAGMA *function from the* `Group/GrpMat/Classical` *package, file* `conformal.m` *(code attributed to Don Taylor) and the result of the automatic translation. The only human work needed is to re-arrange the (iterated)* `where` *constructs and to replace* `SubStructure` *by an appropriate* `ClosureGroup` *call (and to delete the corresponding comment warnings).* `MatrixByEntries` *is a utility function that emulates the* MAGMA *syntax for matrix construction in* GAP. *The identifier names* `X` *and* `Z` *have a special function in* GAP *and thus got replaced. The translation from the infix* `div` *to the function* `QuoInt` *caused superfluous parentheses.*

[4] W. Bosma, J. Cannon, C. Playoust. The MAGMA algebra system I: The user language. *J. Symbolic Comput.*, 24(3/4):235–265, 1997.

[5] J.N. Bray, D.F. Holt, C.M. Roney-Dougal. *The maximal subgroups of the low-dimensional finite classical groups*, volume 407 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 2013. With a foreword by Martin Liebeck.

[6] J.J. Cannon. An introduction to the group theory language, Cayley. In Michael D. Atkinson, editor, *Computational group theory (Durham, 1982)*, pages 145–183. Academic press, 1984.

[7] J. Carlson, B. Eick, A. Hulpke, E. O'Brien. Algorithms for linear groups. final report of the 2014 workshop. `http://www.birs.ca/workshops/2014/14w5031/report14w5031.pdf`, 2014.

[8] The GAP Group, `http://www.gap-system.org`. *GAP – Groups, Algorithms, and Programming, Version 4.7.4*, 2014.

[9] A. Niemeyer, W. Nickel, M. Schönert, GAP - Getting started and Reference Manual. RWTH Aachen, 1988

[10] E.A. O'Brien, M.R. Vaughan-Lee. The groups with order p^7 for odd prime p. *J. Algebra*, 292(1):243–258, 2005.

[11] M. Schönert et al. *GAP 3.4, patchlevel 4*. Lehrstuhl D für Mathematik, Rheinisch-Westfälische Technische Hochschule, Aachen, 1997.

MAGMA code:

```
EulerPhiInverse:=function(m)
  mfact := Factorization(m);
  if IsEven(m) then
    twopows := {@ 2^i : i in [0..mfact[1][2]] @};
  else
    if m gt 1 then return [ ]; end if;
    twopows := {@ 1 @};
  end if;
  D := Divisors(mfact); P := [ ];
  for d in D do
    if d eq 1 then
      continue ;
    end if;
    if IsPrime(d+1) then
      Append(~P, d+1);
    end if;
  end for;

  S := [ <SeqFact([ ]), m> ];
  for p in Reverse(P) do
    for s in S do
      if s[2] eq 1 then
        continue;
      end if;
      k := 1;
      d, mmod := Quotrem(s[2], p-1);
      while mmod eq 0 do
        if IsEven(d) or d eq 1 then
          Append(~S, <SeqFact([<p, k>])*s[1], d>);
        end if;
        k +:= 1;
        d, mmod := Quotrem(d, p);
      end while;
    end for;
  end for;

  R := { };
  for s in S do
    j := Index(twopows, s[2]);
    if j gt 0 then
      Include(~R, SeqFact([<2, j>] cat s[1]));
      if j eq 1 then
        Include(~R, s[1]);
      end if;
    end if;
  end for;
  return Sort([ Facint(nf) : nf in R]);
end function;
```

Example 2: *A function to determine the inverse of the Euler function, $\phi^{-1}(m)$, for any $m \geq 1$. This function is given as an example on p. 16 of [3]. Since a more substantial programmer interaction is needed in this case, also the final resulting code after human intervention is shown, with hand-edited changes in red. Most of these changes – apart from the translation of the "factor sequence" data type – are almost mechanical and can be done in seconds.*
The example demonstrates the translation of different list types (though the translation of the "indexed set" into a plain list could cause a performance penalty with huge numbers); the case of multi-argument return values (which in this particular case is in form of a list; and the treatment of the '∼' operation. As GAP *has no "factor sequence" data type, this was hand-translated to a list of base/exponent pairs. The function* `Index` *has different meanings in other contexts and is therefore not auto-translated to* `Position`*. (Also the "failure" value returned is not* `0`*, but* `fail`*.)*

Translated GAP code, converter result:

```
EulerPhiInverse:=function(m)
local D,P,R,S,d,j,k,mfact,mmod,p,s,twopows;
  mfact:=Factorization(m);
  if IsEvenInt(m) then
    twopows:=List( # {@-list:
      [0..mfact[1][2]],i->2^i);
  else
    if m > 1 then
      return [];
    fi;
    twopows:=# {@-list:
    [1];
  fi;
  D:=DivisorsInt(mfact);
  P:=[];
  for d in D do
    if d=1 then continue; fi;
    if IsPrime(d+1) then
      Append(TILDEP,d+1);
    fi;
  od;
  S:=[[SeqFact([]),m]];
  for p in Reversed(P) do
    for s in S do
      if s[2]=1 then continue; fi;
      k:=1;
      # =v= MULTIASSIGN =v=
      mmod:=QuotientRemainder(s[2],p-1);
      d:=mmod.val1;
      mmod:=mmod.val2;
      # =^= MULTIASSIGN =^=
      while mmod=0 do
        if IsEvenInt(d) or d=1 then
          Append(TILDES,[SeqFact([[p,k]])*s[1],d]);
        fi;
        k:=k+1;
        # =v= MULTIASSIGN =v=
        mmod:=QuotientRemainder(d,p);
        d:=mmod.val1;
        mmod:=mmod.val2;
        # =^= MULTIASSIGN =^=
      od;
    od;
  od;
  R:=Set([]);
  for s in S do
    j:=Index(twopows,s[2]);
    if j > 0 then
      UniteSet(TILDER,SeqFact(Concatenation([[2,j]],s[1])));
      if j=1 then
        UniteSet(TILDER,s[1]);
      fi;
    fi;
  od;
  return Sort(List(R,nf->Facint(nf)));
end;
```

Resulting working GAP code:

```
EulerPhiInverse:=function(m)
local D,P,R,S,d,j,k,mfact,mmod,p,s,twopows;
  mfact:=Collected(Factors(m));
  if IsEvenInt(m) then
    twopows:=List([0..mfact[1][2]],i->2^i);
  else
    if m > 1 then return []; fi;
    twopows:=[1];
  fi;
  D:=DivisorsInt(m);
  P:=[];
  for d in D do
    if d=1 then continue; fi;
    if IsPrime(d+1) then
      Add(P,d+1);
    fi;
  od;
  S:=[[[],m]];
  for p in Reversed(P) do
    for s in S do
      if s[2]=1 then continue; fi;
      k:=1;
      d:=QuotientRemainder(s[2],p-1);
      mmod:=d[2]; d:=d[1];
      while mmod=0 do
        if IsEvenInt(d) or d=1 then
          Add(S,[Concatenation([[p,k]],s[1]),d]);
        fi;
        k:=k+1;
        d:=QuotientRemainder(d,p);
        mmod:=d[2]; d:=d[1];
      od;
    od;
  od;
  R:=Set([]);
  for s in S do
    j:=Position(twopows,s[2]);
    if j <> fail then
      UniteSet(R,[Concatenation([[2,j]],s[1])]);
      if j=1 then
        UniteSet(R,[s[1]]);
      fi;
    fi;
  od;
  R:=List(R,nf->Product(List(nf,x->x[1]^x[2])));
  Sort(R);
  return R;
end;
```