GAP利用者ガイド

あなたも GAP を使ってみよう!

脇 克志

~~弘前大学 理工学部~~ 山形大学 理学部

~~slwaki@cc.hirosaki-u.ac.jp~~ waki@sci.kj.yamagata-u.ac.jp

平成 14 年 8 月 16 日

これは、7 月 22 日から 7 月 26 日まで千葉大学で行なわれた集中講義の講義ノートです。内容は、代数構造計算システム GAP(version 4.2) を使った有限群論の計算方法を具体的な例を入れながら解説して行きます。いくつかの章は、GAP のマニュアルを翻訳して載せています。また、G. Butler の著書 [1] を基に書いている部分もあります。計算機を使った置換群の計算についての最近の本としては、Cameron [2] が参考になると思います。

Time-stamp: <02/08/16 02:49:58 slwaki>

目 次

第 1 章　GAP の概説 1
1.1 GAP の概要 1
1.1.1 GAP の守備範囲 (出来ること出来ないこと) 1
1.1.2 GAP の基本操作 2
1.1.3 GAP の基本データ構造 (List, Record) 2
1.1.4 データベースとしての GAP 4
1.1.5 計算の保存法 5
1.1.6 GAP のヘルプ機能と補完・編集機能 6
1.2 GAP を使った代数構造の定義 7
1.2.1 有限体の定義 7
1.2.2 ベクトル空間の定義 7
1.2.3 置換群の定義 8
1.2.4 行列群の定義 8
1.2.5 生成元と関係式での群 (Fp-群) の定義 8
1.2.6 Power-Commutator 表現による群 (Pc-群) の定義 . . . 9
1.2.7 GAP の群ライブラリから群を呼び出す 9

第 2 章　GAP のプログラミングと群の計算 13
2.1 GAP でのプログラミング 13
2.1.1 関数の定義方法 13
2.1.2 IF 文の利用 14
2.1.3 For 文の利用 14
2.1.4 局所変数 14
2.1.5 再帰的な関数の定義 15
2.1.6 不変オブジェクト (定数）と可変オブジェクト 16
2.1.7 構造化言語としての GAP 17
2.1.8 プログラムがうまく動かないとき 17
2.1.9 置換 サッカーゲーム 18
2.2 GAP での置換群の計算 20
2.2.1 置換群を定義する 20
2.2.2 置換群の性質を調べる 21
2.2.3 置換群の元に対する計算 22

2.2.4 部分群の構成 22
2.2.5 群の作用 24
2.2.6 固定部分群 26

第 3 章 置換群を計算するための基礎 31

3.1 Strong Generating Sets と Base 31
3.1.1 Schreier Tree の定義 32
3.1.2 Strong Generators と Bases を使う 33
3.2 Backtrack Search アルゴリズム 34
3.2.1 置換群の元に順序を導入する 35
3.2.2 GAP を使った木構造の定義 36
3.2.3 Backtrack Search アルゴリズムの基本原理 38

第1章　GAPの概説

この章はGAPの全体像を捕えてもらうことを目標にしています。GAPには、いろいろな代数構造の計算が可能ですが、ここでは、特に有限群と関係が深い内容を選びました。

1.1　GAPの概要

GAPは、ドイツで産まれた有限代数構造の計算を行なうフリーソフトです。UNIX, Windows, MacOS の3つOS上で動かすことが出来ますが、主にUNIX上でもっとも良く動いているソフトです。この講義ノートでも基本的にUNIX版のGAPを念頭に置いて話を進めて行きますが、ほとんどの計算はどのOSにおいても実行可能であるはずです。GAPは、そのVersionが3から4に移行する段階で、各代数構造のデータ構造をかなり変更しました。そのため、 Version 3で作ったプログラムはVersion 4で使う場合に、部分的または根本的な変更をする必要が出てきます。

最新版のGAPは、`http://www-history.mcs.st-and.ac.uk/~gap/` より入手できます。このサイトを参照すればGAPについての必要にして充分な情報が得られます。

1.1.1　GAPの守備範囲(出来ること出来ないこと)

GAPは、**有限の**メモリーを備えた計算機の中で動く代数構造の計算を得意とする計算システムです。(プログラム言語と呼ぶことも出来ます) よって、**有限の**代数構造の構成、分解、解析をその主なる目的としています。有理数体など一部有限でない代数構造も定義されていますが、私には、これらの有限でない代数構造はあくまで**有限の**代数構造やその表現を実現するのに必要なため、無理してとっつけたと思っています。つまり、有理数体はそれ自身を解析するためではなく、これを土台としたべつの有限代数構造を定義したり、表現したりするために存在してると思っています。逆に有限集合上で構成される代数構造なら利用者が自由にその演算を定義できる環境が調えられていると思います。GAPにとって一番の得意分野は、(私のひいき目かも知れませんが) 有限群とその表現 (指標とモジュラー表現）であると思います。ただし、実際の計算能力の点で、有償のソフトウェアであるMAGAMAに及

ばない部分がいくつかあるのも事実です。しかし、中味がすべて公開されていて改良もできる GAP の方が教育的にも、自分の研究道具としてもより私にとってありがたいソフトウェアであると思っています。

1.1.2　GAP の基本操作

GAP がきちんとインストールされた状態では、“gap” とタイプすることで GAP が起動します。この場合デカデカと GAP のタイトルが現れますが、このタイトルは “gap -b” と打ち込んだ場合は見ないで済ませる事が出来ます。また、GAP を終了させるためには、quit; と打ち込むことになります。GAP に入力する命令は 1 つの命令語とセミコロン”;”を続けて打ち込む必要があります。このセミコロンを押さない場合、たとえ改行キーを押したとしてもまだ命令は完結していないと解釈して GAP は、行の頭に> を表示して命令の続きの入力を待ちます。また命令の後にセミコロンを 2 回続けて打ち込んだ場合、実行された命令は計算結果を画面に表示しないようにします。

```
gap> x:=[1,0,0,0];
[ 1, 0, 0, 0 ]
gap> y:=[ [0,1,0,0],
>         [0,0,1,0],
>         [0,0,0,1],
>         [1,0,0,0]];
[ [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ], [ 1, 0, 0, 0 ] ]
gap> z:=x*y;;
gap> z;
[ 0, 1, 0, 0 ]
```

1.1.3　GAP の基本データ構造 (List, Record)

GAP の基本データ構造は、リスト (List) とレコード (Record) ですが、既存のデータ構造はこれら基本データ構造にいろいろな付加情報を追加しているため、一見すると単純なリストやレコードには、見えないものばかりになってしまいました。GAP が Version 4 になってからの大きな変化で今でも Version 4 ではなく Version 3 を使っている利用者がいるのは、データ構造が単純な Version 3 の方が理解しやすいと考えるプログラマーが多いせいかも知れません。しかし個人的には構造化言語の仕組を採り入れた Version 4 の方がゆくゆく分かりやすく効率的なプログラムを実現してもらえそうだと感じています。

List

では、リストの説明から始めます。リストとは、GAP のオブジェクトをカンマ”,” で区切りながら連ねて”[”と”]”で囲ったものです。例えば、1 か

ら 5 までの数字のリストは、[1,2,3,4,5] と表せます。GAP の命令として、L:=[x,y,z]; と入力すれば、長さが 3 で 3 つの要素 x, y そして z を持つリスト L を定義したことになります。 このリスト L から 2 番目の要素 (つまり、y) を取りだしたいときは、L[2] とすれば良いことになります。また、要素 0 が 100 個並んだリストを作りたいときは、ListWithIdenticalEntries(100,0) とすると作れます。

リストは、変域を表すときにも使われます。たとえば、1 から 100 までの数のリストが [1..100] として表されて、$\{x \in \mathbb{Z} | 1 \leq x \leq 100\}$ を意味する変域となります。10 以下の正の偶数のリストなら、[2,4..10] と表すことが出来ます。

リストを扱うときもっとも頻繁に利用する関数が List です。関数 List は、リストからリストを作る関数で、リストと関数を 1 つ与えるとリストの各成分を関数の入力に代入しその結果をリストにしたものを出力します。関数については、2.1.1 を参照して下さい。例えば、1 から 5 までの数字のリストから、2 から 10 までの偶数のリストを作るときは、List([1..5],i->2*i) とします。 次の例では、まず 3 次から 7 次までの交代群のリストを作った後に、それぞれの交代群に対して関数 Size を適用して各群の位数を計算しています。

```
gap> As:=List([3..7],i->AlternatingGroup(i));;
gap> List(As,Size);
[ 3, 12, 60, 360, 2520 ]
```

List を扱うときに利用するもう 1 つの関数は Filtered です。関数 Filtered もリストからリストを作る関数ですが、これはリストの各要素から条件に合うものだけを抽出する関数です。次の例では、1 から 50 の数のリストから 1 引いた値が 4 で割り切れる素数の要素だけを選び出しています。

```
gap> Filtered([1..50],i-> (i-1) mod 4 = 0 and IsPrime(i) );
[ 5, 13, 17, 29, 37, 41 ]
```

さらに、ForAll や、ForAny を使ってリストの持つ性質を確認することも出来ます。

また、リストに新しい要素を加えたり 2 つのリストを 1 つにまとめる関数として Add, Append, Concatenation などが用意されています。

```
gap> ForAll([1..4],i->IsPrime(2^(2^i)+1));
true
gap> ForAll([1..5],i->IsPrime(2^(2^i)+1));
false
gap> ForAny([1..10],i->IsPrime(2^i-1));
true
gap> ForAny([11..20],i->IsPrime(2^i-1));
true
gap> ForAny([21..30],i->IsPrime(2^i-1));
false
gap> MersenneNr:=[];;
gap> for i in [1..30] do
>      if IsPrime(2^i-1) then
>        Add(MersenneNr,2^i-1);
>      fi;
```

```
>    od;
gap> MersenneNr;
[ 3, 7, 31, 127, 8191, 131071, 524287 ]
gap> MersenneNrPlus:=Concatenation(MersenneNr,[2^31-1,2^61-1]);
[ 3, 7, 31, 127, 8191, 131071, 524287, 2147483647, 2305843009213693951 ]
```

Record

次に、レコードですが、Version 4 になってから、レコードの利用はとても少なく (というより、あまり表面に出なく) なりました。というのも Version 4 の中で各オブジェクトをレコードではなく構造化言語としてのクラスと呼ばれる概念で扱うことが一般的になったからです。ただし、利用者がプログラムする上では、レコードは今でも手軽な構造変数としてたいへん便利なデータ構造だと思います。レコードは、名前を持つ成分を集めて構成された構造変数です。次の例では、変数 `man` を身長 `height`, 体重 `weight`, 名前 `name` という 3 つの成分から構成される構造変数として定義しています。各要素の呼び出し方は "変数名. 要素名" となります。`Record` に登録された成分を、見るためには、関数 `RecNames` を使います。

```
gap> man:=rec( height:=160, weight:=65, name:="K.Waki");
rec( height := 160, weight := 65, name := "K.Waki" )
gap> man.name;
"K.Waki"
gap> man.height;
160
gap> man.weight;
65
gap> RecNames(man);
[ "height", "weight", "name" ]
```

1.1.4　データベースとしての GAP

GAP の持つもう 1 つの大きな特長は、今まで計算されて来た代数構造の情報をまとめた代数構造や指標表のデータベースの存在を上げることが出来ます。これらのデータベースをうまく利用することで、自分が建てた予想の確認や反例の作成に役立ちます。ここでは、どんな群のデータベースがあるのかだけを紹介します。実際のデータベースからの群の呼び出し方法は、1.2.7 を見て下さい。

基本的な群

基本的な群として、自明な群 `TrivialGroup`、巡回群 `CyclicGroup`、可換群 `AbelianGroup`、基本可換群 `ElementaryAbelianGroup`、2 面体群 `DihedralGroup`、Extraspecial`ExtraspecialGroup`、対称群 `SymmetricGroup` と交代群 `AlternatingGroup`、Mathieu 群 `MathieuGroup`、鈴木群 `SuzukiGroup` があります。

古典群

線形 GL、シンプレクティック SP、ユニタリー GU、直交群 GO があります。

選択関数

いくつかの選択関数と呼ばれる、有限群のデータベースから条件を満たす群を呼び出す仕組も用意されています。位数の小さい群を呼び出す選択関数 AllSmallGroups、位数の小さい完全群を呼び出す選択関数 PerfectGroup、次数の小さい可移群を呼び出す選択関数 AllTransitiveGroups、次数の小さい原始的置換群を呼び出す選択関数 AllPrimitiveGroups などが、あります。

1.1.5 計算の保存法

GAP で行なった計算を残す方法には、2種類あると思われます。まず1つ目は、計算を終えた後の GAP の状況 (計算結果を入れた変数やプログラミングした関数の定義内容) をそのままファイルに保存する方法で、SaveWorkspace(filename) という命令を使います。ここで、ファイル名 filename は、ある文字列 (アルファベットと数字を" で囲ったものです。) となります。この命令で filename というファイルにこの命令を実行した直前の状態がそのまま保存されます。GAP を起動するときに gap -L filename と打ち込めば保存した状態が復元されます。この命令で作られるファイルは、非常に大きく中味は、直接覗くことは出来ません。また、別の種類の計算機にファイルを転送した場合、転送先の計算機の種類によってはこのファイルを読みこんで前回の計算状態を再現することが出来ない場合もあります。もう1つの方法は、計算を始める前に InputLogTo(filename) を実行して、その後の入力した命令をすべて filename に記録しておく方法です。この場合、filename には、その後に打ち込んだ命令がテキスト形式で保存されるため、あとから、記録内容を確認することは容易な上、Read(filename) を使って別の計算機で filename の内容を実行して、計算結果を得ることも出来ます。また、ファイルの大きさもとても小さくなります。ただし、すべての計算を、GAP がやり直すため、前回の現状が復活するまで、同じ性能の計算機では、以前の計算時間と同じだけの時間を必要としてしまいます。　もし、計算結果をそのまま保存するだけなら、LogTo(filename) で、画面の出力をそのまま、保存できます。また、PrintTo や AppendTo を使って自分の都合の良い形で、計算結果を出力することも可能です。次の例は、GAP の行列を TeX 型式にして、出力しています。関数 Exec は、GAP から UNIX の命令を実行するための命令です。

```
gap> MakeTeXArray:=function(fname,mtx)
>   local x_len, y_len,i,j;
>   x_len:=Length(mtx[1]);
>   y_len:=Length(mtx);
>   PrintTo(fname,"%\n%  This file is generated by MakeTeXArray\n%\n");
>   AppendTo(fname,"\\[\n\\left(\n");
>   AppendTo(fname,"\\begin{array}{");
>     for i in [1..x_len] do
>       AppendTo(fname,"r");
```

```
>     od;
>   AppendTo(fname,"}\n");
> #
>   for j in [1..y_len] do
>     for i in [1..x_len] do
>       if i > 1  then AppendTo(fname," & "); fi;
>       AppendTo(fname,mtx[j][i]);
>     od;
>     AppendTo(fname," \\\\\n");
>   od;
>   AppendTo(fname,"\\end{array}\n");
>   AppendTo(fname,"\\right)\n\\]\n");
>   return true;
> end;
function( fname, mtx ) ... end
gap> mtx:=[[1,2,3,4],[5,6,7,8],[9,10,11,12]];;
gap> Display(mtx);
[ [   1,   2,   3,   4 ],
  [   5,   6,   7,   8 ],
  [   9,  10,  11,  12 ] ]
gap> MakeTeXArray("array_test.tex",mtx);
true
gap> Exec("cat array_test.tex");
%
%  This file is generated by MakeTeXArray
%
\[
\left(
\begin{array}{rrrr}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\end{array}
\right)
\]
```

1.1.6　GAP のヘルプ機能と補完・編集機能

GAP は、オンラインヘルプ機能を持っています。ある**命令**についてのくわしい説明を知りたい場合は、?**命令**と打ち込みます。そして、見ているオンラインヘルプの次の頁を見たい場合は ?>、前の頁が見たい場合は ?<、 を入力するすることで、前後の関連したオンラインヘルプを読むことが出来ます。

GAP の入力を助けてくれる機能として、命令の補完機能があります。これは、命令を途中まで打ち込んだ段階で、tab キーを押すことにより GAP に定義されている命令の中から、今まで入力した文字で始まるものを探して当てはまるものが 1 つの場合は、入力した文字を補完してくれます。もし、複数の侯補がある場合は、その侯補が共通に持つ文字列のところまで、補完します。さらに、複数の侯補がある場合は、 tab キーを続けて 2 度打ち込むこと

で、複数候補の一覧を表示します。

入力を助けるもう 1 つの機能として、いままで入力した命令を上矢印を押すことで、もう一度表示できる機能です。似た命令を複数行なうときは、この機能により、以前打ち込んだ命令を表示して、左右矢印と Delete キーを使って命令を編集して使うことが出来ます。上矢印を打つ前に、少し文字列を打ち込んでおくと以前同じように文字列を入力した命令を導き出して表示してくれますので、さらに便利です。

1.2 GAP を使った代数構造の定義

この節では、GAP を使ってどのように代数構造を定義するかを紹介していきます。

1.2.1 有限体の定義

有限体の定義は、`GF(p^n)` の形で行ないます。(p はある素数) 例えば、2^3 個の元からなる有限体を `k` としたい場合は、`k:=GF(8);` と打ち込めば良いわけです。また、有限体 `GF(p^n)` の元は、有限体から 0 を除いて構成される巡回乗法群の生成元 `Z(p^n)` の巾として作ることが出来ます。また、`E(n)` は、1 の原始 n 乗根を表します。このため、GAP では、`Z` と `E` を利用者が作るプログラムの中で変数として利用することは出来ません。また、有限体を加法群としてみたいときは、 `AsVectorSpace` を使って有限体を素体上のベクトル空間として定義し直して、各元をこのベクトル空間のベクトルとして表すことも出来ます。有限体 `k` の零元や乗法の単位元は、`Zero(k)` や `One(k)` で定義できます。

```
gap> k:=GF(8);
GF(2^3)
gap> a:=Z(8);
Z(2^3)
gap> List([1..7],i->a^i);
[ Z(2^3), Z(2^3)^2, Z(2^3)^3, Z(2^3)^4, Z(2^3)^5, Z(2^3)^6, Z(2)^0 ]
gap> V:=AsVectorSpace(GF(2),k);
GF(2^3)
gap> Dimension(V);
3
gap> BasisVectors(Basis(V));
[ Z(2)^0, Z(2^3), Z(2^3)^2 ]
gap> Coefficients(Basis(V),a^5);
[ Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> a^5=a^0+a+a^2;
true
```

1.2.2 ベクトル空間の定義

一般ベクトル空間は、有限体上の n 次元ベクトル空間の部分空間として存在します。この n 次元ベクトル空間は、有限体 `GF(q)` に対して、`GF(q)^n` で

定義できます有限体上の一般ベクトル空間を作るためには、まず有限体上のベクトルを作る必要が出てくるでしょう。 ベクトルは各成分が同一の有限体の要素で構成されたリストとなります。一般ベクトル空間は生成元となるベクトルのリストから関数 VectorSpace を使ってきます。また、有理数体を表す Rationals を使えば有理数体上のベクトル空間も構成できます。

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );;
gap> W:= Subspace( V, [ [ 0, 1, 2 ] ] );
<vector space over Rationals, with 1 generators>
```

1.2.3　置換群の定義

置換群は、交代群や対象の部分群として存在しています。やはり置換群の生成元となる置換のリストから、関数 Group を使って構成できます。置換群の計算については、第 2.2 節で詳しく紹介します。

1.2.4　行列群の定義

行列群は、一般線形群の部分群として、行列群の生成元となる行列のリストから、やはり関数 Group を使って構成できます。行列の定義は長さの揃った同一の体の元で構成されたベクトルたちのリストとなります。もちろん行列群の生成元となるものは、正方行列である必要があります。

```
gap> m1 := [ [ Z(3)^0, Z(3)^0,   Z(3) ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ 0*Z(3),   Z(3), 0*Z(3) ] ];;
gap> m2 := [ [   Z(3),   Z(3), Z(3)^0 ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ Z(3)^0, 0*Z(3),   Z(3) ] ];;
gap> m := Group( [m1, m2] );
Group(
[
 [ [ Z(3)^0,Z(3)^0,Z(3) ], [ Z(3),0*Z(3),Z(3) ], [ 0*Z(3),Z(3),0*Z(3) ] ],
 [ [ Z(3),Z(3),Z(3)^0 ],   [ Z(3),0*Z(3),Z(3) ], [ Z(3)^0,0*Z(3),Z(3) ] ]
])
gap> Size(m);
864
```

1.2.5　生成元と関係式での群 (Fp-群) の定義

生成元に対する関係式から、群を定義する場合は、まず自由群 f を関数 FreeGroup を使って定義して、その生成元を使って関係式を表します。一般に群 f について、その n 番目の生成元は、f.n と表すことが出来ます。最後に自由群をその関係式で割ることで、新しい群が構成できます。以下の例では、群 $2.A_8$ を定義してその位数を求めています。

```
gap>   f := FreeGroup( "m1","m2","m3","m4", "m5","m6");
<free group on the generators [ m1, m2, m3, m4, m5, m6 ]>
```

```
gap>   m1:=f.1;; m2:=f.2;; m3:=f.3;; m4:=f.4;;
gap>   m5:=f.5;; m6:=f.6;;
gap>   rel:=[
>   m1^6, m2^2*m1^3, m3^2*m1^3, m4^2*m1^3, m5^2*m1^3, m6^2*m1^3,
>   (m1*m2)^3*m1^3,
>   (m1*m3)^2*m1^3, (m1*m4)^2*m1^3, (m1*m5)^2*m1^3, (m1*m6)^2*m1^3,
>   (m2*m3)^3*m1^3, (m2*m4)^2*m1^3, (m2*m5)^2*m1^3, (m2*m6)^2*m1^3,
>                   (m3*m4)^3*m1^3, (m3*m5)^2*m1^3, (m3*m6)^2*m1^3,
>                                   (m4*m5)^3*m1^3, (m4*m6)^2*m1^3,
>                                                   (m5*m6)^3*m1^3 ];;
gap> g:=f/rel;
<fp group on the generators [ m1, m2, m3, m4, m5, m6 ]>
gap> Size(g);
40320
```

1.2.6 Power-Commutator 表現による群 (Pc-群) の定義

可解群を生成元と関係式で定義する場合、以下に上げる Power - Commutator 表現を利用する事により、その群に対する計算がとてもスムーズに出来ます。Power - Commutator 表現とは、関係式の表し方で、左辺が必ずある 1 つの元 a_i の巾か 2 つの元 a_i, a_j $(i \leqq j)$ の交換子になっているもので、そのとき対応する右辺は、$a_{i+1}^{\alpha_{i+1}} a_{i+2}^{\alpha_{i+2}} \cdots a_n^{\alpha_n}$ の形で与えられるものです。

```
gap> F := FreeGroup("a","b","c","d");;
gap> a := F.1;; b := F.2;; c := F.3;; d := F.4;;
gap> rels := [a^2, b^3, c^2, d^2, Comm(b,a)/b, Comm(c,a)/d, Comm(d,a),
>             Comm(c,b)/(c*d), Comm(d,b)/c, Comm(d,c)];;
gap> G := F / rels;
<fp group on the generators [a,b,c,d]>
gap> H := PcGroupFpGroup( G );
<pc group of size 24 with 4 generators>
```

1.2.7 GAP の群ライブラリから群を呼び出す

群ライブラリから群を呼び出して定義する場合、Filter と呼ばれる属性を指定することが出来ます。主な Filter は、`IsPermGroup`, `IsMatrixGroup`, `IsPcGroup` などです。Filter を指定しないで、群ライブラリからの呼び出しを行なった場合は、それぞれの関数に設定されているデフォルトの Filter が適用されます。ただし、ライブラリの種類によって、使えない Filter もあります。

基本的な群

基本的な群を定義する関数としては、`TrivialGroup`, `CyclicGroup`, `AbelianGroup`, `ElementaryAbelianGroup`, `DihedralGroup`, `ExtraspecialGroup`, `AlternatingGroup`, `SymmetricGroup`, `MathieuGroup`, `SuzukiGroup` (`Sz`) などが上げられます。Filter

として IsMatrixGroup を指定した場合は、さらに行列の成分が含まれる有限体も指定することが出来ます。

```
gap> CyclicGroup(12);
<pc group of size 12 with 3 generators>
gap> CyclicGroup(IsPermGroup,12);
Group( [ ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12) ] )
gap> matgrp1:= CyclicGroup( IsMatrixGroup, 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp1 );
Rationals
gap> matgrp2:= CyclicGroup( IsMatrixGroup, GF(2), 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp2 );
GF(2)
```

古典群

次のようにして、一般線形群、シンプレスティク群そしてユニタリ群を定義出来ます。また関数 Action と作用に関するオプション OnLines を利用することで、線形群から射影線形群を作ることも出来ます。

```
gap> Gm:=GL(4,2);
SL(4,2)
gap> DisplayCompositionSeries(Gm);
G (size 20160)
 | A(8) ~ A(3,2) = L(4,2) ~ D(3,2) = O+(6,2)
1 (size 1)
gap> g:=GL(4,3);;Size(g);
24261120
gap> pgl:=Action(g,Orbit(g,Z(3)^0*[1,0,0,0],OnLines),OnLines);;
gap> Size(pgl);
12130560
```

選択関数

選択関数を使うことで、GAP の持つ群のライブラリから与えられた条件を満たすものだけを、選択して抽出することが出来ます。ここで、条件は、関数とその出力値で、与えられます。最初の例では、関数 Size とリスト [4,8..60] (1.1.3 参照) で、位数が 60 以下の 4 の倍数となるものを条件として設定しています。更に、関数 f を使って、位数 2 の元の中心化群が位数 4 の基本可換群となる群だけを選ぶように設定しています。２番目の例は、次数が 4 から 10 までの原始的な置換群について、関数 f の条件を満たすものを選び出しています。計算結果に５次の交代群が５次、６次そして１０次の原始的な置換群として、登場しているのが分かります。

```
gap> f:=function(G)
> local cc,cc2,x,C;
> cc:=List(ConjugacyClasses(G),Representative);
> cc2:=Filtered(cc,x->Order(x)=2);
> for x in cc2 do
>   C:=Centralizer(G,x);
>   if Order(C)=4 and IsElementaryAbelian(C) then
```

```
>     return true;
>   fi;
> od;
> return false;
> end;
gap> ag:=AllGroups(Size,[4,8..60],f,true);;
gap> List(ag,Size);
[  4,  8, 12, 12, 16, 16, 20, 24, 24, 24, 28, 32, 32, 36, 36,
  36, 40, 40, 44, 48,  48, 48, 48, 48, 52, 56, 56, 60, 60, 60 ]
gap> Filtered(ag,IsSimple);
[ Alt( [ 1 .. 5 ] ) ]
gap> at:=AllPrimitiveGroups(NrMovedPoints,[4..10],f,true);
[ Alt( [ 1 .. 4 ] ), Sym( [ 1 .. 4 ] ),
  Alt( [ 1 .. 5 ] ), PSL(2,5),A(5) ]
gap> List(at,NrMovedPoints);
[ 4, 4, 5, 6, 10 ]
```

第2章　GAPのプログラミングと群の計算

2.1　GAPでのプログラミング

この節では、GAP のマニュアルの [Tutorial:Functions] の部分を基にして GAP でのプログラミングを解説して行きます。GAP におけるプログラミングは、他の言語と同様に関数を定義して行くことで進んで行きます。

2.1.1　関数の定義方法

では、まずプログラミングの本では、必ず最初の例として使われるプログラムの例を使って関数の定義方法を見て行きましょう。以下実行例では、sayhello と言う名前の関数を定義しています。 \n は、改行記号を入力するときに使います。また、バックスラッシュ\ を出力したい場合は、\\ と打ち込みます。あまり、きちんと確認していませんが、私の環境では、日本語の出力も問題無く行なわれました。

```
gap> sayhello:= function()
> Print("hello, world.\n");
> end;
function (  ) ... end
gap> sayhello();
hello, world.
```

もっと、簡易的な関数の定義方法もあります。次の例では矢印記号-> を利用して、入力値を 2 乗する関数を定義しています。

```
gap> sq:=x->x^2;
function( x ) ... end
gap> List([1..10],i->sq(i));
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

複数の引数 x, y を持つ関数も function(x,y) の形で定義できます。次のように定義できます。関数の計算の結果は return を使って出力します。

```
gap> f:=function(x,y)
> return x^2-y^2;
> end;
function( x, y ) ... end
gap> List([1..10],i->f(i+1,i));
[ 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 ]
```

また、定義した関数は Print 命令を使って表示することも可能です。

```
gap> Print(f,"\n");
function ( x, y )
    return x ^ 2 - y ^ 2;
end
```

今までに定義した変数や関数の名前は、`NamesUserGVars();` を使って表示することが出来ます。

```
gap> NamesUserGVars();
[ "f", "sq" ]
```

2.1.2 IF 文の利用

IF 文は、条件分岐に使われます。使い方は、

`if` 評価 `then` 評価が真の時の実行命令 `else` 評価が偽の時の実行命令 `fi;`

です。例えば次の関数は、IF 文を使って符合を表示しています。

```
gap> ShowPM:=function(x)
> if x > 0 then Print("This number is plus.\n");
>          else Print("This number is *not* plus.\n");
> fi;
> end;
function( x ) ... end
gap> ShowPM(10);
This number is plus.
gap> ShowPM(-5);
This number is *not* plus.
```

2.1.3 For 文の利用

For 文は、基本的な繰り返しの命令でです。次の例では、変数 `ListOfWords` に、入っている文字列を 1 つづつ、変数 `word` に代入して各々の文字列の長さを表示しています。

```
gap> ListOfWords:=["Have","a","great","time"];
[ "Have", "a", "great", "time" ]
gap> for word in ListOfWords do
> Print("The Size of WORD [", word,"] is ",Size(word),"\n");
> od;
The Size of WORD [Have] is 4
The Size of WORD [a] is 1
The Size of WORD [great] is 5
The Size of WORD [time] is 4
```

2.1.4 局所変数

関数の定義の途中で、**その関数の中だけで**必要となる変数を局所変数と呼びます。GAP では、局所変数を関数定義の最初のところで `local` を使って定義する必要があ

ります。次の例では、局所変数 g と i を使って $PSL(n,q)$ の位数を計算する関数 L を定義しています。

```
gap> L:=function(n,q)
>  local g,i;
>  g:=q^( n*(n-1)/2 );
>  for i in [2..n] do
>   g:=g*(q^i-1);
>  od;
>  return g/Gcd(q-1,n);
> end;
function( n, q ) ... end
gap> L(2,3);
12
gap> L(3,2);
168
```

変数に関連してもう 1 つお話します。関数の入力値は、その種類により関数内の計算に影響を受ける場合と受けない場合があるようです。

```
gap> f:=function(I)
>   I:=I+1;
> end;
function( I ) ... end
gap> g:=function(L)
>   L[1]:=L[1]+1;
> end;
function( L ) ... end
gap> I:=1;; L:=[1];;
gap> f(I); g(L);
gap> I; L;
1
[ 2 ]
```

上の例では、数値 I の値は関数 f を適用した後も変化していませんが、リスト L の値は、関数 g を適用した後で増えています。この辺の違いは C 言語などでプログラムを書いたことのある人は、なんとなく理解できるのではないかと思います。

2.1.5 再帰的な関数の定義

再帰的な定義とは、関数を定義している途中に現在定義している関数自身を使う定義方法です。上手に使うとシンプルな関数定義ができますが、注意しないと計算自身が終了しないものになる危険性があります。次の例では、導来群 (交換子群) を計算する関数 DerivedSubgroup を使って、交換子列を計算する関数 D を定義しています。

```
gap> D:=function(g)
> local d;
> if IsAbelian(g) then
>   return [g];
```

```
> else
>   d:=DerivedSubgroup(g);
>   if d=g then
>     return [g];
>   else
>     return Concatenation([g],D(d));
>   fi;
> fi;
> end;
function( g ) ... end
gap> g:=Group([(3,4)(5,6), (1,4,6,2,3,5), (3,5)(4,6)]);
Group([ (3,4)(5,6), (1,4,6,2,3,5), (3,5)(4,6) ])
gap> D(g);
[ Group([ (3,4)(5,6), (1,4,6,2,3,5), (3,5)(4,6) ]),
  Group([ (1,2)(3,4), (1,5,3)(2,6,4) ]),
  Group([ (3,4)(5,6), (1,2)(3,4)  ]) ]
gap> List(D(g),Size);
[ 48, 12, 4 ]
```

2.1.6 不変オブジェクト (定数）と可変オブジェクト

GAP のオブジェクトには、不変オブジェクト (Immutable) と可変オブジェクト (Mutalbe) が あります。例えば、ベクトル空間 V やその基底などは、不変オブジェクト (Immutable) で、勝手に修正を加えることが出来ない様になってます。次の例では、２元体 `k` と３次元ベクトル空間 `V` を定義して、その自明な基底を `bs` としています。この `bs` は、ベクトルのリストとなっていますが、ベクトルたちが不変オブジェクト (Immutable) であるため、2 つ目の 基底ベクトルの第一成分 `bs[2][1]` を変更しようとしてもエラーが出ています。この例では、`ShallowCopy` というオブジェクトの複製を作る命令を利用して、`bs` から可変オブジェクトになっているベクトルのリスト `bs0` を作っています。新しく作られた `bs0` では、2 つ目のベクトルの第一成分を変更できるようになっているのが分かります。最後に `Display` 命令を使ってベクトルたちがほんとうに変更されたことを確認しています。GAP においてエラーが発生した場合、GAP は、計算を中止して `brk>`というプロンプトを出します。この状態から通常状態に戻るためには `quit;` と入力する必要があります。(エラーが発生した場合の対応は 2.1.8 を参考にして下さい。)

```
gap> k:=GF(2);
GF(2)
gap> V:=k^3;
( GF(2)^3 )
gap> bs:=Basis(V);
CanonicalBasis( ( GF(2)^3 ) )
gap> bs[2][1]:=Z(2);
Lists Assignment: <list> must be a mutable list
not in any function
Entering break read-eval-print loop, you can 'quit;' to quit to outer
loop,
or you can return and ignore the assignment to continue
brk> quit;
gap> IsMutable(bs);
false
gap> bs0:=List(bs,v->ShallowCopy(v));
```

```
[ <a GF2 vector of length 3>, <a GF2 vector of length 3>,
  <a GF2 vector of length 3> ]
gap> IsMutable(bs0);
true
gap> bs0[2][1]:=Z(2);
Z(2)^0
gap> Display(bs);
 1 . .
 . 1 .
 . . 1
gap> Display(bs0);
 1 . .
 1 1 .
 . . 1
```

2.1.7　構造化言語としての GAP

GAP の各オブジェクトは、カテゴリ (Category) と呼ばれる集合たちの要素として特徴づけられています。GAP では、2 つのオブジェクトが与えられてその積を計算せよと命令された場合、それぞれのオブジェクトがどんなカテゴリに属しているかを確認して、そのカテゴリのために用意された積の作法に従って計算を進めて行きます。使われているオブジェクトがどんなカテゴリに属しているかを見るためには、`CategoriesOfObject` を使います。次の例では、置換 `(1,2,3)` と 3 次の対称群がどんなカテゴリに属しているかを表示しています。

```
gap> CategoriesOfObject((1,2,3));
[ "IS_PERM", "IsExtLElement", "IsExtRElement",
"IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne",
"IsMultiplicativeElementWithInverse",
  "IsAssociativeElement", "IsFiniteOrderElement" ]
gap> CategoriesOfObject(SymmetricGroup(3));
[ ``IsListOrCollection'', ``IsCollection'', ``IsExtLElement'',
  ``CategoryCollections(IsExtLElement)``, ``IsExtRElement'',
  ``CategoryCollections(IsExtRElement)``,
  ``CategoryCollections(IsMultiplicativeElement)``,
  ``CategoryCollections(IsMultiplicativeElementWithOne)``,
  ``CategoryCollections(IsMultiplicativeElementWithInverse)``,
  ``CategoryCollections(IsAssociativeElement)``,
  ``CategoryCollections(IsFiniteOrderElement)``,
``CategoryCollections(IS_PERM)``,
  ``IsGeneralizedDomain'', ``IsMagma'', ``IsMagmaWithOne'',
  ``IsMagmaWithInversesIfNonzero'', ``IsMagmaWithInverses'' ]
```

2.1.8　プログラムがうまく動かないとき

プログラムを作って行く仮定で、実行エラーに遭遇した場合、GAP は、プロンプト `brk>` を出して停止します。そんなときは、`Where` を使ってプログラムがどんな関数を実行している途中で、エラーを出してしまったのかを確認知ることが出来ます。また、`DownEnv` を使うことで、関数の階層レベルを、移動して、各レベルでの変数の状態を確認することが出来ます。次の例では、

関数 Error を使って意図的にエラーを発生させています。階層レベルを上下することで、変数 n の値が変化していることが分かります。

```
gap> test:= function( n )
>    if n > 3 then Error( "!" ); fi; test( n+1 ); end;;
gap> test( 1 );
Error ! at
Error( "!" );
Entering break read-eval-print loop,
you can 'quit;' to quit to outer loop,
or you can return to continue
brk> Where();
test( n + 1 ); called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> n;
4
brk> DownEnv();
brk> n;
3
brk> Where();
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( 2 );
brk> n;
1
brk> Where();
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( -2 );
brk> n;
3
```

もう 1 つ、これは GAP でのプログラムに限った話ではありませんが、プログラムには、出来るだけ沢山のコメント文 (プログラムを説明するための文章で計算機からは無視される部分) を付けるようにしましょう。コメント文は、他人にプログラムを使ってもらったり、自分が過去に作ったプログラムを再活用するときに、とても役に立ちます。GAP では、記号 # の後に書かれた文字がコメント文と見なされ計算機からは、すべて無視されます。

2.1.9　置換 サッカーゲーム

最後に、有限群を使った簡単なゲームプログラムを紹介します。GAP は、もともと BASIC 言語などが持つ、随時入力関数がありませんそのため、プログラムの中に入力を促すような、動作を入れることが出来ません。そのため、GAP を使って、対話型プログラムを書くことは、とても大変だと感じています。次のプログラムでは、変数 p の中に、状況を表す要素を Record として記録して、関数 Go を随時動かしながら擬似対話型プログラムにしています。ゲームの内容は、２人で行なうサッカーで、プレイヤーはそれぞれ、２つ置換 [x,y] と [a,b] を持ちます。この置換の元をうまく作って駒 p を自分の

ゴールポイントに移動させることができると、得点です。例えば、Go(x,p)
とすると、置換 x 使って駒を移動させたことになります。

```
GAME_TIME:=10;
if not IsBound(n) then   n:=11; fi;
G:=AlternatingGroup(n); g1:=1; g2:=n;
if IsOddInt(n) then   Mp:=(n+1)/2; else   Mp:=n/2; fi;
p1:=0; p2:=0;
p:=rec(Start:=Mp,P1:=p1,P2:=p2,G1:=g1,G2:=g2,ps:=Mp,time:=GAME_TIME,pl:=1);
Print(" プレーヤー 1 のゴールポイントは、 ",g1," です。\n");
Print(" プレーヤー 2 のゴールポイントは、 ",g2," です。\n");
Print("   駒の最初やゴール後のの位置は、",Mp," です。\n");
Print("\n では、プレーヤー 1 からどうぞ!\n");
a:=Random(G); b:=Random(G);
while G<>Group([a,b]) do   a:=Random(G);   b:=Random(G); od;
x:=Random(G); y:=Random(G);
while G<>Group([x,y]) do   x:=Random(G);   y:=Random(G); od;
Go:=function(x,p)
  local pt, pos;
  pt:=p.ps^x;
  pos:=ListWithIdenticalEntries(n,'-');
  pos[1]:='1';  pos[n]:='2';  pos[pt]:='o';
  Print("駒の移動 : ",p.ps," -> ",pt,"\t\t",pos,"\n");
  p.ps:=pt;
  if pt=p.G1 then
    Print("  プレーヤー 1 ゴール!! \n");    p.P1:=p.P1+1;
    Print(" [ ",p.P1," - ",p.P2," ] \n");  p.ps:=p.Start;   p.pl:=1;
  fi;
  if pt=p.G2 then
    Print( " プレーヤー 2 ゴール!! \n");    p.P2:=p.P2+1;
  Print(" [ ",p.P1," - ",p.P2," ] \n");    p.ps:=p.Start;   p.pl:=2;
  fi;
  p.time:=p.time-1;
  if p.time=0 then
    Print(" ++ ゲーム終了 ++\n [ ",p.P1," - ",p.P2," ] \n"");
    if p.P1 = p.P2 then
      Print("  引き分けです。 \n");
    else
      if p.P1 > p.P2 then
        Print("  プレーヤー 1 の勝利 !! \n");
      else
        Print("  プレーヤー 2 の勝利 !! \n");
    fi;
  fi;
else
  p.pl:=2/p.pl;
  Print("残り時間 : ",p.time,"\n 次は、プレーヤー",p.pl,"です。\n");
fi;
end;
```

実際のゲームのようすは次のようになります。 下のゲームでは、プレーヤー 1 が自責点を与えてしまっています。得点が入ると次は、得点を入れられた方がプレーします。

```
gap> Read("game.gap");
```

```
 プレーヤー 1 のゴールポイントは、 1 です。
 プレーヤー 2 のゴールポイントは、 11 です。
   駒の最初やゴール後のの位置は、6 です。
では、プレーヤー 1 からどうぞ!
gap> Go(y,p);
駒の移動 : 6 -> 4                1--o------2
残り時間 : 5
次は、プレーヤー 2 です。
gap> Go(a,p);
駒の移動 : 4 -> 9                1-------o-2
残り時間 : 4
次は、プレーヤー 1 です。
gap> Go(x,p);
駒の移動 : 9 -> 11               1---------o
 プレーヤー 2 ゴール!!
 [ 0 - 1 ]
残り時間 : 3
次は、プレーヤー 1 です。
```

2.2　GAP での置換群の計算

この節では、GAP での群の計算を実感してもらうために、GAP のマニュアルにある Tutorial:Group and Homomorphisms の内容の 1 部を紹介して行きます。

2.2.1　置換群を定義する

ここでは、置換群を使っていくつかの簡単な計算をしてみましょう。

最初の例では、2 つの置換 (1,2) と (1,2,3,4,5,6,7,8) で 生成される置換群 s8 が定義されています。(もちろんこれは、8 次の対称群ですね)

```
gap> s8:= Group( (1,2), (1,2,3,4,5,6,7,8) );
Group( (1,2), (1,2,3,4,5,6,7,8) )
```

この s8 から例えば、s8 のすべての偶置換の群 (8 個の点の交代群) は s8 の交換子群を計算することで求めることが出来ます。

```
gap> a8:= CommutatorSubgroup( s8, s8 );
Group([ (1,2,3), (2,3,4), (3,4,5), (4,5,6), (5,6,7), (6,7,8) ])
```

交代群 a8 は 6 つの置換で生成される部分群として定義されました。余談ですが、このように生成元を沢山持つ群は、一般にその後のこの群に対する計算を遅くしてしまう可能性があります。関数 SmallGeneratingSet を使うことで、生成元の個数を減らすことが出来ます。

```
gap> SmallGeneratingSet(a8);
[ (1,3)(4,6,5,8), (1,5,6,2,3)(4,8,7) ]
```

2.2.2 置換群の性質を調べる

それでは、定義した置換群の基本的な性質を調べてみましょう。a8 の位数、可換群であるかどうか、完全群かどうかなどが次のようにして計算できます。

```
gap> Size( a8 ); IsAbelian( a8 ); IsPerfect( a8 );
20160
false
true
```

また、群の位数の約数となる素数 p に対するシロー p-部分群 は、関数 SylowSubgroup を呼ぶことで求めることが出来ます。ここでは、FactorsInt を使って位数を素因数分解したときの、素因数をまず計算しています。

```
gap> Set( FactorsInt( Size( a8 ) ) );
[ 2, 3, 5, 7 ]
gap> syl2:=SylowSubgroup( a8, 2 );
Group([ (1,8)(6,7), (2,3)(6,7), (4,5)(6,7), (2,4)(3,5), (1,6)(7,8),
  (1,2)(3,8)(4,6)(5,7) ])
```

上の例ではシロー 2-部分群が計算され変数 syl2 に入りました。次の計算はこの syl2 に対してその位数、正規化群、中心、その中心の元の中心化群 cent、更に この cent の交換子群列と降中心列が計算されています。変数 last は、1つ前の計算結果を表しています。

```
gap> Size( syl2 );
64
gap> Normalizer( a8, syl2 );
Group([ (1,8)(6,7), (4,5)(6,7), (2,3)(6,7), (2,4)(3,5), (1,6)(7,8),
  (1,5)(2,7)(3,6)(4,8) ])
gap> last = syl2;
true
gap> Centre( syl2 );
Group([ ( 1, 8)( 2, 3)( 4, 5)( 6, 7) ])
gap> cent:= Centralizer( a8, last );
Group([ (4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,2)(3,8) ])
gap> Size( cent );
192
gap> DerivedSeries( cent );
[ Group([ (4,6)(5,7), (2,6,4)(3,7,5), (1,4)(2,6)(3,7)(5,8),
      (1,2)(3,8)(4,6)(5,7), (4,5)(6,7), (2,3)(4,5), (1,8)(2,3)(4,5)(6,7)]),
  Group([ (2,6,4)(3,7,5), (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7),
      (4,5)(6,7), (2,3)(4,5), (1,8)(2,3)(4,5)(6,7) ]),
  Group([ (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7), (4,5)(6,7),
(2,3)(4,5), (1,8)(2,3)(4,5)(6,7) ]), Group([ (1,8)(2,3)(4,5)(6,7) ]),
Group(()) ]
gap> List( last, Size );
[ 192, 96, 32, 2, 1 ]
gap> low:= LowerCentralSeries( cent );
[ Group([ (4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,2)(3,8) ]),
  Group([ (2,6,4)(3,7,5), (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7),
      (4,5)(6,7), (2,3)(4,5), (1,8)(2,3)(4,5)(6,7) ]) ]
```

ある一点を固定させる置換の集合で与えられる部分群 (固定部分群) も計算できます。

```
gap> stab:= Stabilizer( a8, 1 );
Group([ (2,3,4), (3,4,5), (4,5,6), (5,6,7), (6,7,8) ])
gap> Size( stab );
2520
gap> Index( a8, stab );
8
```

2.2.3　置換群の元に対する計算

例えば、任意の群の元をランダム取ってくることが出来ます。 また、ある元の中心化群を計算したり、 部分群の共役を取ったり、 2 つの部分群の共通部分を求めることも出来ます。

```
gap> Random( a8 );
(1,6,3,2,7)(4,5,8)
gap> Random( a8 );
(1,3,2,4,7,5,6)
gap> cent:= Centralizer( a8, (1,2)(3,4)(5,8)(6,7) );
Group([ (5,6)(7,8), (5,7)(6,8), (3,4)(6,7), (3,5)(4,8), (1,2)(6,7),
  (1,3)(2,4) ])
gap> Size( cent );
192
gap> conj:= ConjugateSubgroup( cent, (2,3,4) );
Group([ (5,6)(7,8), (5,7)(6,8), (2,4)(6,7), (2,8)(4,5), (1,3)(6,7),
  (1,4)(2,3) ])
gap> inter:= Intersection( cent, conj );
Group([ (5,7)(6,8), (5,8)(6,7), (1,2)(3,4)(5,7)(6,8),(1,3)(2,4)(5,7)(6,8) ])
gap> Size( inter );
16
gap> IsElementaryAbelian( inter );
true
gap> norm:= Normalizer( a8, inter );
Group([ (5,7)(6,8), (5,8)(6,7), (1,2)(3,4)(5,7)(6,8),
(1,3)(2,4)(5,7)(6,8), (5,7,8), (3,4)(5,7,8,6), (2,3)(5,7,8,6),
(1,5,4,8,2,6)(3,7) ])
gap> Size( norm );
576
```

2.2.4　部分群の構成

例えば、a8 における $2^3{:}L_3(2)$ のような構造を持つ部分群を構成したいといった場合を考えてみましょう。固定点を持たない 3 つの involution(位数が 2 の元) から生成される基本可換群 elab を作り、その a8 の正規化群 norm を計算するのが 1 つの方法です。elab は、2^3 の部分になります。もし、norm が、$2^3{:}L_3(2)$ であれば、 $L_3(2)$ の位数が 168 より norm の位数は、$168 \times 8 = 1344$ となるはずです。

```
gap> elab:= Group( (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8),
```

```
>                  (1,5)(2,6)(3,7)(4,8) );;
gap> Size( elab );
8
gap> IsElementaryAbelian( elab );
true
gap> SetName( elab, "2^3" ); elab;
2^3
gap> norm:= Normalizer( a8, elab );;
gap> Size( norm );
1344
```

次に剰余群を計算してみましょう。norm という群と、それの剰余群 f は、f への自然な準同形写像 (核が elab となる準同形写像) hom で結ばれています。GAP は、その後の剰余群の計算を楽にするため出来るだけ剰余群 f の次数が小さくなるようにしています。そのため、f の作用域と norm との間には何の係わり合いもありません。剰余群 f の元に対する準同形写像 hom の逆像は norm の右剰余類となります。

```
gap> hom:=NaturalHomomorphismByNormalSubgroup( norm , elab );
<action homomorphism>
gap> f := Image( hom );
Group([(),(),(),(1,2)(3,4),(1,3)(2,4),(1,2)(5,6),(3,5)(4,6),(2,3)(6,7)])
gap> Size( f );
168
gap> Kernel( hom ) = elab;
true
gap> x:= Random( norm );
(1,7,4,3,6,8,5)
gap> Image( hom, x );
(1,2,3,5,4,7,6)
gap> coset:= PreImages( hom, last );
RightCoset(Group( [ (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8),
  (1,5)(2,6)(3,7)(4,8) ] ),(2,8,3,4,5,7,6))
gap> IsRightCoset( coset );
true
```

出力が示すように、でき上がった剰余群は、また置換群になります。norm の 8 つの生成元が剰余群でどの元に対応しているのかが分かります。最初の 3 つの元は、elab に含まれていたことになります。得られた群 f が、ほんとうに $L_3(2)$ であるか確認するため、関数 IsSimple で単純群であることをそして関数 ~~IsomorphismTypeFiniteSimpleGroup~~ で具体的に単純群の種類を求めています。 **IsomorphismTypeInfoFiniteSimpleGroup**

```
gap> IsSimple( f ); IsomorphismTypeFiniteSimpleGroup( f );
true
rec( series := "L", parameter := [ 2, 7 ],
  name := "A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~
  2A(1,7) = U(2\
,7) ~ A(2,2) = L(3,2)" )
gap> SetName( f, "L_3(2)" );
```

```
gap> IsomorphismTypeInfoFiniteSimpleGroup( f );
rec( name := "A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~ 2A(1,7)
= U(2,7) ~ A(2,2) = L(3,2)", parameter := [ 2, 7 ], series := "L" )
```

剰余群 f は、 (剰余類上でなく)f の元に右から作用しています。この作用は、f の正則置換表現を与えます。

```
gap> op:= Operation( f, f , OnRight );;
gap> IsPermGroup( op );
true
gap> IsSimpleGroup( op );
true
```

norm の 7 つの自明でない元の正規部分群 elab への共役による作用は $L_3(2)$ の 7 つの点への表現を与えます。この置換群を norm に埋め込むことで、norm が基本的可換群 2^3 の $L_3(2)$ により 拡大された群であることが分かります。

```
gap> op := Action( norm, elab );
Group( [ (), (), (), (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6),
  (2,3)(6,7) ] )
gap> IsSubgroup( a8, op ); IsSubgroup( norm, op );
true
true
gap> IsTrivial( Intersection( elab, op ) );
true
gap> SetName( norm, "2^3:L_3(2)" );
```

2.2.5　群の作用

置換群 a8 の別の置換表現を得るために、a8 の 1 つの共役類に対する a8 の作用を考えてみましょう。

```
gap> ccl := ConjugacyClasses( a8 );; Length( ccl );
14
gap> List( ccl, c -> Order( Representative( c ) ) );
[ 1, 2, 2, 3, 6, 3, 4, 4, 5, 15, 15, 6, 7, 7 ]
gap> List( ccl, Size );
[ 1,210,105,112,1680,1120,2520,1260,1344,1344,1344,3360,2880,2880 ]
```

共役類の集合 ccl は、共役類のリストになってますが、ここでは、その中で、元の個数が 112 の位数が 3 の元の共役類に注目してみましょう。

```
gap> class := First( ccl, c -> Size(c) = 112 );;
gap> op := Action( a8, AsList( class ) );;
```

関数 First は、共役類のリスト ccl の要素 (つまり共役類）のうち元の個数が 112 となる最初の要素 (つまり共役類）を出します。共役類 class と AsList(class) は、本質的には何等変わりない位数 3 の共役類ですが、実質的には class の方は基になっている群、代表元、中心化群の位数、この共役類に含まれる元の個数、などの情報を集めたものであるのに対して、AsList(class) は、まさに、この共役類に含まれる元をリストとして列挙したものとなってます。この計算によりこの 112 個の元に作用する置換群 op

が作られました。まずこの置換群が 原始的であるか確認してみましょう。置換群が原始的で無い場合は、関数 Blocks を使って最小ブロックシステムを計算することが出来ます。

```
gap> IsPrimitive( op, [ 1 .. 112 ] );
false
gap> blocks := Blocks( op, [ 1 .. 112 ] );
[[1,7],[2,13],[3,19],[8,14],[9,20],[16,27],[23,34],[15,21],[10,26],
[43,48],[4,25],[5,31],[44,53],[74,81],[22,28],[17,33],[18,39],
[79,86],[11,32],[73,77],[45,58],[6,37],[50,59],[56,65],[12,38],
[49,54],[55,60],[51,64],[52,69],[78,82],[30,41],[75,85],[29,35],
[76,89],[24,40],[36,42],[46,63],[47,68],[93,96],[94,99],[84,91],
[97,100],[105,107],[80,90],[98,103],[108,110],[62,71],[61,66],
[83,87],[57,70],[67,72],[88,92],[95,102],[106,109],[101,104],[111,112]]
```

上の計算で、作用域 [1..112] を指定していることに注意して下さい。作用域は、置換群から簡単に決定できると思われるかも知れませんが、例えば元 $(2,3,4)$ で生成された置換群の作用域は [1..4] かも知れないし、[2..5] かも知れません。上の例で作用域を [1..113] とすれば、群 op は、原始的どころか、可移にすらなりません。変数 blocks には、ブロックのリストが入ります。次の計算では、個々のブロックの大きさとブロックの総数を求めて、更にこの blocks に対する op の作用によって出来る置換群 op2 を求めています。今回の作用は、個々のブロックを 1 つの集合と見て、そのブロックに op を作用させ、別のどのブロックに、移るのかを見ています。ブロックの総数が 56 なので、op2 の作用域は [1..56] となります。

```
gap> Length( blocks[1] );  Length( blocks );
2
56
gap> op2 := Action( op, blocks, OnSets );;
gap> IsPrimitive( op2, [ 1 .. 56 ] );
true
```

以上の方法で次数が 56 の 原始的な置換群が得られました。この primitivity より、この群 op2 の 1 点固定部分群は極大部分群となります。そこで、a8 から op2 までを結ぶ準同形写像を計算し、この 1 点固定部分群の逆像を求め事で a8 の別の極大部分群を求めてみましょう。

```
gap> ophom := ActionHomomorphism( a8, op );;
gap> ophom2 := ActionHomomorphism( op, op2 );;
gap> composition := ophom * ophom2;;
gap> stab := Stabilizer( op2, 2 );;
gap> preim := PreImages( composition, stab );
Group([(2,5,7),(1,4)(2,7),(2,6,7),(1,3)(5,7),(6,8,7)])
```

このように新しい作用を構成することで、置換群の別の姿を見いだすことが出来ます。そしてそれらの橋渡しをするのが、準同形写像と言うことが出来ます。実は、2.2.4 で作った自然な準同形写像 hom も、ある作用域に対する作用からでき上がっています。それが何かを見るためには、次のようなコマンドを入力する必要があります。

```
gap> t := UnderlyingExternalSet( hom );
<xset:RightTransversal(2^3:L_3(2), Group([(5,6)(7,8),
  (5,7)(6,8),(3,4)(7,8),(3,5)(4,6),(1,2)(7,8),(1,3)(2,4)]))>
```

ここに現れた t は、外部集合 (external sets) と呼ばれるもので、作用域と作用している群さらに作用の方法を規定している関数 (つまり作用している群と作用域の直積から作用域への写像）の 3 つを 1 つにまとめたものとなります。今回の例では、位数が 192 の norm の部分群

```
Group([(5,6)(7,8),(5,7)(6,8),(3,4)(7,8),(3,5)(4,6),
                                  (1,2)(7,8),(1,3)(2,4)])
```

の右剰余類が作用域であることが分かります。

2.2.6　固定部分群

外部集合を使わないで、群の作用を定義することももちろん出来ます。ここでは、a8 の部分群を、ある作用域の 1 点固定部分群として構成してみましょう。まず最初はもっとも簡単な 8 点の集合を作用域としたときの 1 点固定部分群を見て行きましょう。

```
gap> u8 := Stabilizer( a8, 1 );
Group([(2,3,4),(2,4)(3,5),(2,6,4),(2,4)(5,7),(2,8,6,4)(3,5)])
gap> Index( a8, u8 );
8
gap> Orbit( a8, 1 ); Length( last );
[ 1, 3, 2, 4, 5, 6, 7, 8 ]
8
```

この計算例は、さらに a8 の部分群たちを見つけるためのヒントを与えています。というのも、すべての部分群は、ある作用域 (具体的にはその部分群の剰余類の集合) に対する 1 点固定部分群として与えられるいるからです。ですから、部分群を見つけることは、対応する作用域 (作用) を見つけることにほかなりません。 では、べつの作用を作るため、まず 1 から 8 までの数字の組を作ることから始めましょう。

```
gap> pairs := Tuples( [1..8], 2 );;
```

この数字の組を作用域として a8 の作用を考えてみましょう。ここでは、作用の方法を明示するため、OnPairs というオプションを付け加えて、数字の組 pairs に対する a8 の作用が可移であるか如何か聞いています。

```
gap> IsTransitive( a8, pairs, OnPairs );
false
```

作用は当然可移でないので、その軌跡を計算します。

```
gap> orbs := Orbits( a8, pairs, OnPairs );; Length( orbs );
2
gap> List( orbs, Length );
[ 8, 56 ]
gap> List( orbs, o -> o[1] );
[ [ 1, 1 ], [ 1, 2 ] ]
```

この計算により、a8 の pair 上での作用は長さ 8 と 56 の 2 つの軌跡を持ち、それぞれの軌跡の代表が [1,1] と [1,2] であることが分かりました。それでは、2 番目の軌跡についてその代表 [1,2] を固定する 1 点固定部分群を計算しましょう。

```
gap> u56 := Stabilizer( a8, orbs[2][1], OnPairs );; Index( a8, u56 );
56
```

さあこれで、2 つめの部分群が得られました。次に以後の計算をやりやすくするため、2 番目の軌跡である 56 個の数字の組に対する a8 の作用を、関数 ActionHomomorphism 使って、56 個の点に対する作用に変換してみましょう。でき上がった群 a8_56 は、56 個の点に作用する置換群となります。

```
gap> h56 := ActionHomomorphism( a8, orbs[2], OnPairs );;
gap> a8_56 := Image( h56 );;
```

もし、この置換群 a8_56 が原始的であれば、その 1 点固定部分群に対応する u56 は、a8 の極大部分群となりますが...

```
gap> IsPrimitive( a8_56, [1..56] );
false
```

残念ながら、u56 は、極大部分群では無いようです。では、当然 u56 を含むような a8 の部分群 (super groups of u56 in a8) を見つけたくなりますね。a8_56 が原始的でないので、関数 Blocks を使って、a8_56 の作用に対するブロックシステムを求めることが出来ます。

```
gap> blocks := Blocks( a8_56, [1..56], [1,14] );
[ [ 1, 11, 13, 14, 39, 44, 45 ], [ 2, 7, 15, 16, 30, 31, 46 ],
  [ 3, 8, 17, 25, 28, 41, 42 ], [ 4, 6, 24, 27, 29, 43, 52 ],
  [ 5, 12, 18, 26, 34, 40, 54 ], [ 9, 19, 21, 32, 37, 49, 53 ],
  [ 10, 20, 23, 33, 36, 48, 55 ], [ 22, 35, 38, 47, 50, 51, 56 ] ]
```

上の計算では、作用域 [1..56] をブロックに分解していますが、ただしその中の 1 つのブロックには 1 と 14 が含まれるように注文を付けています。この条件で、作用域 [1..56] を 8 つのブロックに分解することが出来ました。では、この 8 つのブロックを作用域としてそのうち 1 つめのブロックを固定する固定部分群を計算してみましょう。それぞれのブロックは、1 つの集合と考えて個々の集合単位で作用を行なうため、オプション OnSets を付け加えておきます。

```
gap> u8_56 := Stabilizer( a8_56, blocks[1], OnSets );;
gap> Index( a8_56, u8_56 );
8
gap> u8b := PreImages( h56, u8_56 );; Index( a8, u8b );
8
gap> IsConjugate( a8, u8, u8b );
true
```

上の計算では、この 1 点固定部分群 u8_56 から、逆像として a8 の部分群 u8b を得て、以前計算した u8 と u8b が共役である事も示しています。(実は、これはそれほどビックリすることではありませんというのも、u56 は、もともと a8 の 2 点固定部分群と思えるので、それを含む部分群として 1 点固定部分群が出てきたわけです。) 実は、次のようにすると別のブロックシステムを得ることが出来ます。

```
gap> blocks := Blocks( a8_56, [1..56], [1,7] );;
gap> u28_56 := Stabilizer( a8_56, [1,7], OnSets );;
gap> u28 := PreImages( h56, u28_56 );;
gap> Index( a8, u28 );
28
```

この指数が 28 の部分群が、極大部分群となることは a8 が 2,4,7 を指数とする部分群を持たないと云う事からあきらかですが、具体的に a8_56 の blocks 上での作用が原始的であることを確認することで確かめることが出来ます。

```
gap> IsPrimitive( a8_56, blocks, OnSets );
true
```

固定部分群を計算する関数 Stabilizer は、部分群にも適用できます。次の計算では、部分群 u56 の部分群で点 3 を固定するものを計算しています。(u56 は、点 1,2 を固定しますので、u336 は、3 つの点 1,2,3 を固定します。)

```
gap> u336 := Stabilizer( u56, 3 );;
gap> Index( a8, u336 );
336
```

他の関数もまた部分群に適用可能です。以下の計算では、u336 が、点 4〜点 8 の 5 つの点の 3 ツ組の集合 (元の個数は $5 \times 4 \times 3 = 60$ 個) に正則に作用していることが分かります。

```
gap> IsRegular( u336, Orbit( u336, [4,5,6], OnTuples ), OnTuples );
true
```

以前数字の組でやったように、次の計算で u336 の右剰余類の集合に対する作用を通じて、作用域 [1..336] に作用する新しい置換群を作ることが出来ます。

```
gap> t := RightTransversal( a8, u336 );;
gap> a8_336 := Action( a8, t, OnRight );;
```

この u336 を含むような部分群を得るため、別の自明でないブロックシステムを見つけます。

```
gap> blocks := Blocks( a8_336, [1..336] );; blocks[1];
[ 1, 43, 85 ]
```

つまり、u336 と 43 番目と 85 番目の右剰余類の和集合は指数 112 の部分群となることが分かります。そこで、次のように u336 の 43 番目の右剰余類の代表元を加えて生成される部分群を作れば指数 112 のこの部分群が得られます。

```
gap> u112 := ClosureGroup( u336, t[43] );;
gap> Index( a8, u112 );
112
```

同様に u112 を含む部分群 u56b を得ることが出来ますが、こちらは u56 とは、共役にならないばかりか、極大部分群となってしまうことが分かります。

```
gap> blocks := Blocks( a8_336, [1..336], [1,7,43] );;
gap> Length( blocks );
56
gap> u56b := ClosureGroup( u112, t[7] );; Index( a8, u56b );
56
gap> IsPrimitive( a8_336, blocks, OnSets );
true
```

群の作用 (2.2.5) で紹介したように、共役も 1 つの群の作用となります。作用を指定しないで関数 OrbitLength を使った場合は、置換群の元に対するデフォルトの作用である共役が採用されます。

```
gap> OrbitLength( a8, (1,2)(3,4)(5,6)(7,8) );
105
gap> orb := Orbit( a8, (1,2)(3,4)(5,6)(7,8) );;
gap> u105 := Stabilizer( a8, (1,2)(3,4)(5,6)(7,8) );; Index( a8, u105 );
105
```

注意:群 G の元 elm を代表元とする共役類の大きさは、OrbitLength(G,elm) で計算できますが、Size(ConjugacyClass(G, elm)) を使った方が計算効率が良いようです。

```
gap> Size( ConjugacyClass( a8, (1,2)(3,4)(5,6)(7,8) ) );
105
```

もちろん、u105 は、元 (1,2)(3,4)(5,6)(7,8) の中心化群です。では、いままでと同様に u105 を含むような部分群を見つめましょう。

```
gap> blocks := Blocks( a8, orb );; Length( blocks );
15
gap> blocks[1];
[ (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,8)(6,7), (1,8)(2,7)(3,5)(4,6),
  (1,7)(2,8)(3,6)(4,5), (1,5)(2,6)(3,8)(4,7), (1,6)(2,5)(3,7)(4,8),
  (1,4)(2,3)(5,7)(6,8)]
```

u105 を含む指数が 15 の部分群を作るために、ふたたび閉包を使います。ここからは、誤解や混乱を招かないように慎重に話を進めて行きたいと思いますが、もう少しこちらの話につきあってください。

部分群 u105 は、点 (1,2)(3,4)(5,6)(7,8) の一点固定部分群として定義されました。このとき、点 (1,2)(3,4)(5,6)(7,8) の軌跡の元と u105 の剰余類の間に一対一の対応ができ上がっています。点 (1,2)(3,4)(5,6)(7,8) には、u105 が対応しています。u105 を含むような指数 15 の a8 の部分群を得るためには、最初のブロック blocks[1] に含まれる (1,2)(3,4)(5,6)(7,8) 以外の点に対応する、剰余類の元を 1 つ u105 に加えるて閉包を取ることによって得られます。仮に、(1,2)(3,4)(5,6)(7,8) 以外の点として、点 (1,3)(2,4)(5,8)(6,7) を採用すると、a8 の元で共役の作用で (1,2)(3,4)(5,6)(7,8) を (1,3)(2,4)(5,8)(6,7) に移す元を見つける必要があります。そして、関数 RepresentativeAction は、まさにそれをしてくれるものです。この関数は、2 つの点と 1 つの群を与えて群の元の中で、1 つ目の元を 2 つめの元に移す元を見つけます。実は、 4 番目の引数として作用の方法も指定できますが、今回の例では、この引数を必要としません。もし、2 つの点を共役で結ぶような ã8 の元が見つからないとき、関数 RepresentativeAction は、False を結果として出力します。

```
gap> rep := RepresentativeAction( a8, (1,2)(3,4)(5,6)(7,8),
>                                     (1,3)(2,4)(5,8)(6,7) );
(2,3)(6,8)
gap> u15 := ClosureGroup( u105, rep );; Index( a8, u15 );
15
```

群 a8 は、指数 5,3 の部分群を持たないので、ここででき上がった指数 15 の部分群 u15 は、極大部分群となります。また、元 (2,3)(6,7) を、ũ105 に加えることで、別の指数 15 の部分群 u15b を得ることが出来ます。

```
gap> u15b := ClosureGroup( u105, (2,3)(6,7) );; Index( a8, u15b );
15
gap> RepresentativeAction( a8, u15, u15b );
fail
```

関数 RepresentativeAction は、u15 と u15b は、非共役であることを示しています。

第3章　置換群を計算するための基礎

この章では、置換群を素材としたプログラミングの感じを理解して頂くため、計算機を使って、置換群の計算をするときにもっとも基本となる固定部分群鎖 (Stabilizer Chain) を使った Backtrack search アルゴリズムを、GAP で具体的にプログラミングしてみたいと思います。前半では、70 年代に Charles Sims によって確立された **base** と **strong generating sets** の概念を使った置換群の元の表現方法を紹介します。そして後半では、計算機で置換群のプログラミングを行なうときに使われる基本的な知識を、GAP の内部データを見ながら、確認し Backtrack search アルゴリズムを、実行するプログラムを作ります。群 G と G の元 x が与えられたとき、$Stab_G(x) := \{y \in G | x^y = x\}$ として、これを G での、点 x の固定部分群と呼びます。

3.1　Strong Generating Sets と Base

置換群 G と その作用域Ωが与えられたとき、Ω の部分集合 $B = [b_1, \dots, b_n]$ で、$G^{(i)} := Stab_{G^{(i-1)}}(b_i)$ としたとき、$G^{(0)} = G, \dots, G^{(n)} = \{()\}$ となるものを置換群 G の **base** と呼びます。さらに、置換群 G の生成元 S が、任意の i について、集合 $S \cap G^{(i)}$ が $G^{(i)}$ の生成元の集合となるとき、S を **strong generating set (SGS)** と呼びます。(もっと正確言うと S は、base B に対する SGS) 更に、この $G^{(i)}$ を固定部分群鎖と呼びます。もし、任意の i について、常に $G^{(i)} \supsetneqq G^{(i+1)}$ が成り立つとき、**reduced** と呼びます。

今、$R^{(i)}$ を $G^{(i)}$ の $G^{(i-1)}$ の中での右剰余類の代表とします。すると、G の任意の元 g は、$g = r_n \dots r_1$ (ただし $r_i \in R^{(i)}$) と一意に表現することが出来ます。$G^{(i)}$ の $G^{(i-1)}$ の中での右剰余類の代表は、軌跡 $O^{(i)} := {b_i}^{G^{(i-1)}}$ の中の各点と 1 対 1 に対応しています。そこで、右剰余類の代表を 1 つのリスト `T` を使って表すことが出来ます。ここで、$O^{(i)}$ に含まれる点 p に対して、リスト `T` の p 番目の成分 `T[p]` は、この p に対応する右剰余類の代表 (つまり、点 b_i を点 p に移す $G^{(i-1)}$ に含まれる置換) となります。

この方法では、次数 n が非常に大きくなった場合、沢山の異なる置換を剰余類の代表として、GAP が記憶しておく必要が産まれます。そこで、ここでは、もっと効率良く剰余類の代表を記憶しておく方法を紹介します。

3.1.1　Schreier Tree の定義

以後次のようなリスト T を **Factorized Inverse Transversal** または、**Schreier tree** または、**Schreier vectors** と呼びます。今、O[i] に、$O^{(i)} := {b_i}^{G^{(i-1)}}$ の点がリストとして入っているとします。(このリスト O[i] の第一成分 O[i][1] は、b_i としておきます。) リスト T[i] の 成分 T[i][p] は、$G^{(i-1)}$ の生成元で、リスト O[i] に含まれる点 p に対して、$\mathrm{p}^{\mathtt{T[i][p]}}$ は、O[i] に含まれてさらにリスト O[i] の中で点 p より前方に位置しているものが入っているとします。$b_i = \mathtt{O[i][1]}$ であることを考えると、つまり O[i] に含まれる点 p に T[i][p] を作用させると、リスト O[i] の点で b_i により近い点に移る。と考えれば良いでしょう。このリスト、O[i] を帰納的に使う事により、点 p を点 b_i に移す $G^{(i-1)}$ の元 R[i][p] を構成することが出来ます。そしてこの構成された元の逆元がまさに、点 p に対応する剰余類の代表となります。 このように、群の生成元と Schreier tree T[i] そして 軌跡を表すリスト O[i] を記憶することで、剰余類の代表をいつでも呼び出すことが可能となります。

```
gap> G:=SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> S:=StabChain(G);
rec(
  labels      := [ (), (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ],
  genlabels   := [ 2, 3, 4, 5 ],
  generators  := [ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ],
  identity    := (),
  relativeOrders := [ 2, 3, 2, 2 ],
  stabilizer := rec(
      labels      := [ (), (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ],
      genlabels   := [ 2, 3 ],
      generators  := [ (3,4), (2,4,3) ],
      identity    := (),
      stabilizer  := rec(
          labels      := [ (), (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ],
          genlabels   := [ 2 ],
          generators  := [ (3,4) ],
          identity    := (),
          stabilizer  := rec(
              labels     := [ (), (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ],
              genlabels  := [  ],
              generators := [  ],
              identity   := ()
          ),
          orbit       := [ 3, 4 ],
          translabels := [ ,, 1, 2 ],
          transversal := [ ,, (), (3,4) ]
      ),
      orbit       := [ 2, 3, 4 ],
      translabels := [ , 1, 3, 3 ],
      transversal := [ , (), (2,4,3), (2,4,3) ]
```

```
  ),
  orbit       := [ 1, 3, 4, 2 ],
  translabels := [ 1, 4, 5, 4 ],
  transversal := [ (), (1,4)(2,3), (1,3)(2,4), (1,4)(2,3) ]
)
gap> O:=[];; T:=[];; St:=S;;
gap> while GroupStabChain(St)<>TrivialGroup(IsPermGroup) do
>      Add(O,St.orbit);
>      Add(T,St.transversal);
>      St:=St.stabilizer;
>    od;
gap> O; T;
[ [ 1, 3, 4, 2 ], [ 2, 3, 4 ], [ 3, 4 ] ]
[ [ (), (1,4)(2,3), (1,3)(2,4), (1,4)(2,3) ],
  [   ,         (),    (2,4,3),    (2,4,3) ],
  [   ,           ,         (),      (3,4) ] ]
gap> List([1..3],i->
>       List(O[i],p->
>  Position( O[i], p ^ T[i][ p ] )
>           )
>         );
[ [ 1, 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1 ] ]
gap> TracePoint:=function(G,ob,t,p)
> local rep;
> rep:=Identity(G);
> while p^rep <>ob[1] do
>   rep:=rep*t[p^rep];
> od;
> return rep;
> end;
function( G, ob, t, p ) ... end
gap> R:=[];;
gap> for i in [1..3] do
>       Rs:=[];
>       for p in O[i] do
>         Rs[p]:=TracePoint(G,O[i],T[i],p);
>       od;
>       Add(R,Rs);
>    od;
gap> R;
[ [ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ],
  [   ,         (),     (2,4,3),   (2,3,4) ],
  [   ,           ,          (),     (3,4) ] ]
```

3.1.2 Strong Generators と Bases を使う

それでは、いよいよ Base と Strong Generators に登場してもらいましょう。次の例では、GAP が計算した Strong Generators がきちんと条件を満たしていることを確認しています。ここでは、 `B` が Base, `Sg` が Strong Generators, そして、 `Stbs` が固定部分群鎖とします。(つまり、`Stbs[i]`$=G^{(i)}$)

```
gap> B:=BaseOfGroup(G);
[ 1, 2, 3 ]
gap> Sg:=StrongGeneratorsStabChain(S);;
gap> Sg:=Concatenation([Identity(G)],Sg);
[ (), (3,4), (2,4,3), (1,3)(2,4), (1,4)(2,3) ]
gap> St:=S;; Stbs:=[];;
gap> while GroupStabChain(St)<>TrivialGroup(IsPermGroup) do
>      St:=St.stabilizer;
>      Add(Stbs,GroupStabChain(St));
>    od;
gap> Stbs;
[ Group([ (3,4), (2,4,3) ]), Group([ (3,4) ]), Group(()) ]
gap> Stbs[1]=Stabilizer(G,B[1]);
true
gap> ForAll([2,3],i->Stbs[i]=Stabilizer(Stbs[i-1],B[i]));
true
gap> ForAll([1..3],i->Stbs[i]=Group(Intersection(Stbs[i],Sg)));
true
```

ここでは、この SGS を使って、ある置換 x が、群 G に含まれているかをチェックする方法を見てみましょう。G の固定部分鎖を $G = G^{(0)}, G^{(1)}, \ldots, G^{(r)} =$ () とします。 置換 x が、G に含まれているならば、どこかの 右剰余類 $G^{(1)}\mathtt{R}[1][\mathtt{p}]^{-1}$ に含まれることになります。よって、$x\mathtt{R}[1][\mathtt{p}]$ は、$G^{(1)}$ に含まれます。このとき、軌跡の点と右剰余類の間にある 1 対 1 対応により、$\mathtt{p} = {b_1}^{\mathtt{x}}$ であることが、言えるので、置換 $x\mathtt{R}[1][\mathtt{p}_1]$ は、瞬時に計算できます。あとは、帰納的に $G^{(r)}$ まで、計算を続けていきます。もし、x が G に属していれば、最後に置換 x が、恒等置換になるはずですので、ここで置換 x が 群 G に含まれているかどうかのチェックが出来ます。次の例では、置換 $(1, 2, 3, 4)$ が、G に含まれていることを示しています。

```
gap> x:=(1,2,3,4);;
(1,2,3,4)
gap> x:=x* R[1][B[1]^x];
(2,4)
gap> x:=x* R[2][B[2]^x];
(3,4)
gap> x:=x* R[3][B[3]^x];
()
```

3.2 Backtrack Search アルゴリズム

この章では、SGS を使った置換群の計算のうち Backtrack Search アルゴリズムを紹介します。このアルゴリズムは、置換群 G と この G を定義域、「真・偽」を値域とする関数 f が与えられて、G の部分集合 $H := \{x \in G | f(x) = \text{真}\}$ が G の部分群となるとき、この H の生成元を求めるアルゴリズムです。例えば、$f(x)$ を x が、$(1, 2)$ と可換なら 「真」 非可換なら「偽」を返す関数なら H は、G の中での $(1, 2)$ の中心化群となりますし、2 つの部分群 K_1, K_2 を考えて、両者に含まれているとき「真」そうでないとき「偽」とすれば、$H = K_1 \cap K_2$ となります。

Backtrack Search アルゴリズムでは、3.2.1 で定義する順序に基づいて、順に各元 x に対して $f(x)$ をチェックして行きます。$f(x)$ の値が真となった元を集めて暫定的な H を構成するのですが、この暫定的な H の軌跡から、もう調べる必要の無い (たとえ $f(x)$ が真であっても、いままで、見つけた H の元の積で表せる) ものをスキップして、チェックを進めます。このように調べる必要の無い元を無視することにより効率的に H の生成元を求める事が出来ます。

3.2.1 置換群の元に順序を導入する

ここでは、 Backtrack Search において、重要となる置換群の元に対する順序について説明して行きます。置換群 G の各元は、Base の各点をどこに移すかで一意に特徴づけられています。そのため、G の元を、長さが Base の長さと等しい数列を使って表すことが出来ます。(これを **base image** と呼んだりします。) 次の例では、元 `x` の base image (`[2,3,4]`) を計算した後に、復元する関数 `BaseImageToElm` を定義して、base image からもとの置換を再構成しています。

```
gap> BIx:=List(B,i->i^x);
[ 2, 3, 4 ]
gap> BaseImageToElm:=function(G,O,T,bi)
>   local g,ai,u;
>   g:=Identity(G);
>   ai:=ShallowCopy(bi);
>   for i in [1..Size(bi)] do
>     u:=TracePoint(G,O[i],T[i],ai[i]);
>     g:=u^-1*g;
>     ai:=List(ai,p->p^u);
>   od;
>   return g;
> end;
function( G, O, T, bi ) ... end
gap> BaseImageToElm(G,O,T,BIx);
(1,2,3,4)
```

さて、この置換に対し、対応する base image を使って辞書式順序と呼ばれる順序を導入します。すなわち、2 つの置換 x, y に対して、

$$x < y \Longleftrightarrow \exists k \leqq r; {b_i}^x = {b_i}^y (i < k) \text{ and } {b_k}^x < {b_k}^y$$

とします。この順序で、群 G の元を小さい順に並べると以下の例のようになります。

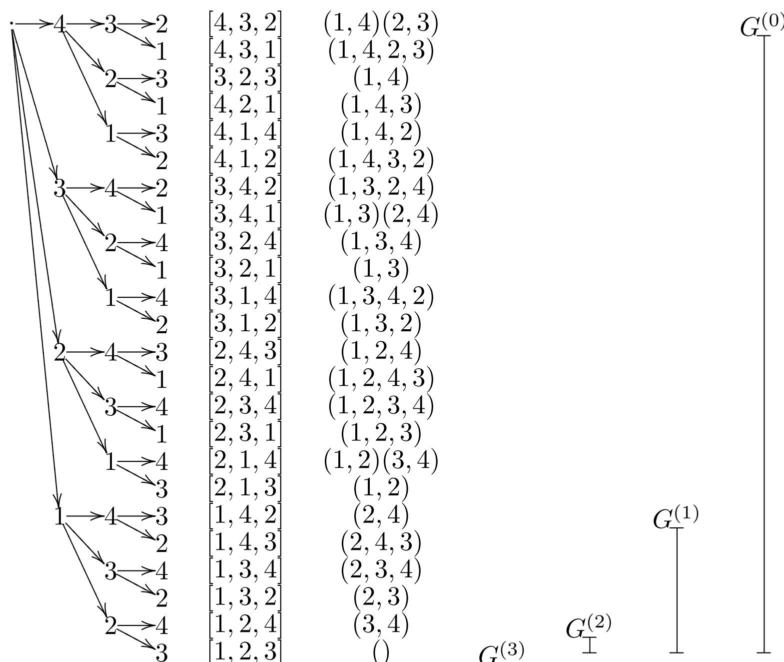
```
gap> eltG:=Elements(G);
[ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4), (1,2,3),
  (1,2,3,4), (1,2,4,3), (1,2,4), (1,3,2), (1,3,4,2), (1,3), (1,3,4),
  (1,3)(2,4), (1,3,2,4), (1,4,3,2), (1,4,2), (1,4,3), (1,4), (1,4,2,3),
  (1,4)(2,3) ]
gap> BIs:=List(eltG,x->List(B,i->i^x));
```

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

この base image を使うと次の図のように、置換群 G の元を、木構造の端末と見ることが出来ます。この図では、上から下に大きい順に元を並べています。右に付けた固定部分群鎖の範囲を合わせて見ると、

$$
\begin{array}{rcl}
x \in G^{(i)} & \Longleftrightarrow & {b_k}^x = b_k, \quad (k \leqq i) \\
G^{(i)}x = G^{(i)}y & \Longleftrightarrow & {b_k}^x = {b_k}^y, \quad (k \leqq i)
\end{array}
$$

と言うように、この順序が固定部分群鎖やその剰余類と密接に関係していることが良く分かります。



3.2.2　GAP を使った木構造の定義

3.2.1 の木構造を GAP で、構築してみましょう。1.1.3 にあるとおり、GAP の基本的なデータ構造は、リスト (List) とレコード (Record) で構成することが出来ます。今回の木構造は、点 (`pt`) とその点から出る枝 (`br`) を要素として持つレコード (仮にこれを**枝データ構造** と呼ぶことにしましょう。) を使い、木構造は枝データ構造のリストと定義しましょう。枝データ構造の要素 `pt` には、base image の 1 つの要素が入ります。`br` は、その点 `pt` から伸

びる新しい枝を集めたリスト（つまり、br は、枝データ構造のリスト）とします。ただし、その点が端点で、そこから先に伸びる枝が存在しない場合は、br=[] と言う空のリストにしておきます。

次に上げるプログラムでは、関数 MakeInitTree で、基本となる木の幹を設定して、さらに関数 AddBranch で、この幹に枝を付け加えます。この 2 つの関数を使って木構造を作る関数が MakeTreeFromBaseImages です。

```
MakeInitTree:=function(bi)
 local tr;
 if Size(bi)=1 then
   tr:=rec( pt:=bi[1], br:=[] );
 else
  tr:=rec( pt:=bi[1], br:=MakeInitTree(bi{[2..Size(bi)]}) );
 fi;
 return [tr];
end;

AddBranch:=function( tr, bi )
 local pos, pts;
 pts:=List(tr,t->t.pt);
 pos:=Position(pts,bi[1]);
 if pos=fail then
   Add(tr,MakeInitTree(bi)[1]);
 else
   if Size(bi)>1 then
     tr:=AddBranch(tr[pos].br,bi{[2..Size(bi)]});
   else
     Print("This base image exits in the tree.\n");
   fi;
 fi;
 end;

 MakeTreeFromBaseImages:=function(bis)
   local tr, bi;
   tr:=MakeInitTree(bis[1]);
   for bi in bis{[2..Size(bis)]} do
      AddBranch(tr,bi);
   od;
   return tr;
 end;
```

このプログラムで、実際にデータ構造を作ると次のようになります。

```
gap> tr:=MakeTreeFromBaseImages(BIs);
[ rec( pt := 1,
      br := [ rec( pt := 2,
                  br := [ rec( pt := 3, br := [  ] ),
                         rec( pt := 4, br := [  ] ) ] ),
             rec( pt := 3,
                  br := [ rec( pt := 2, br := [  ] ),
                         rec( pt := 4, br := [  ] ) ] ),
             rec( pt := 4,
                  br := [ rec( pt := 2, br := [  ] ),
                         rec( pt := 3, br := [  ] ) ] ) ] ),
  rec( pt := 2,
      br := [ rec( pt := 1,
                  br := [ rec( pt := 3, br := [  ] ),
                         rec( pt := 4, br := [  ] ) ] ),
             rec( pt := 3,
```

```
                  br := [ rec( pt := 1, br := [  ] ),
                          rec( pt := 4, br := [  ] ) ] ),
             rec( pt := 4,
                  br := [ rec( pt := 1, br := [  ] ),
                          rec( pt := 3, br := [  ] ) ] ) ] ),
 rec( pt := 3,
      br := [ rec( pt := 1,
                  br := [ rec( pt := 2, br := [  ] ),
                          rec( pt := 4, br := [  ] ) ] ),
             rec( pt := 2,
                  br := [ rec( pt := 1, br := [  ] ),
                          rec( pt := 4, br := [  ] ) ] ),
             rec( pt := 4,
                  br := [ rec( pt := 1, br := [  ] ),
                          rec( pt := 2, br := [  ] ) ] ) ] ),
 rec( pt := 4,
      br := [ rec( pt := 1,
                  br := [ rec( pt := 2, br := [  ] ),
                          rec( pt := 3, br := [  ] ) ] ),
             rec( pt := 2,
                  br := [ rec( pt := 1, br := [  ] ),
                          rec( pt := 3, br := [  ] ) ] ),
             rec( pt := 3,
                  br := [ rec( pt := 1, br := [  ] ),
                          rec( pt := 2, br := [  ] ) ] ) ] ) ]
```

木構造を見やすく表示するために表示用の関数を作ってみるのもいいでしょう。

```
 PrintTree:=function(tr,TAB)
  local ed, tb;
  tb:=ShallowCopy(TAB);
  Append(tb,"\t");
  for ed in tr do
    if ed.br=[] then
Print(tb,ed.pt,"\n");
    else
        Print(tb,ed.pt,"\n");
        PrintTree(ed.br,tb);
    fi;
  od;
 end;
```

それでは、このデータ構造を使って Backtrack search アルゴリズムの原理を見てみましょう。

3.2.3 Backtrack Search アルゴリズムの基本原理

基本的な Backtrack Search アルゴリズムの進め方は、$H^{(i)} = H \cap G^{(i)} = \{x \in G^{(i)} | f(x) = 真\}$ が分かっているとの仮定の基で、$H^{(i-1)}$ を求める方法を取っています。 $G^{(0)} = G$ より、 $H^{(0)}$ が求める H となります。3.2.1 で導入された順序では、小さい順に並べるときちんと $G^{(i)}$ が入れ子式に収まっています。ですから、単純に小さい順にチェックして行けば、順次 $H^{(i)}$ が計算できる事になります。今、 $H^{(i)}$ が分かってるとします。すると、$G^{(i-1)} - G^{(i)}$ に含まれる元から小さい順に、$f(x)$ の真偽を確認して行きます。3.2.1 で述

べた base image の性質から、

$$x \in G^{(i-1)} - G^{(i)} \Longleftrightarrow {b_k}^x = b_k, \quad (k < i), \quad \text{and } {b_i}^x > b_i$$

となります。そしてある $x \in G^{(i-1)} - G^{(i)}$ において $f(x) =$ 真 となったとき、$H^{(i)}$ と x で生成される群の軌跡を計算します。計算された複数個の軌跡について、それぞれの軌跡の中の数字で最小のものをその軌跡の代表とします。つまり代表にならなかった数字 j は、$H^{(i)}$ と x で生成される群で、代表になったある数 $j_0(< j)$ に移す元 z が、$H^{(i)}$ と x で生成される群の中に存在することになります。さて、ある元 x より大きい 元 $y \in G^{(i-1)} - G^{(i)}$ について ${b_i}^y = j$、 $f(y) =$ 真 となったとします。$z \in G^{(i-1)}$ より、${b_k}^z = b_k$ $(k < i)$ となり、$y' = yz$ と置くと、${b_k}^{y'} = b_k$ $(k \leqq i)$、$f(y') =$ 真 が成り立ちます。よって、$y' \in H^{(i)}$ となり、このような y は、 $H^{(i)}$ と x で生成される群に含まれることが、分かります。

つまり、ほんとうにチェックする必要がある元は、${b_i}^y$ が軌跡の代表 (それぞれの軌跡の中の数字で最小のもの) となるものに限られます。では、GAP で作ったプログラムを見てみましょう。

まず、メインプログラム BacktrackSearch を見てみましょう。このプログラムでは、いままで出てきた、軌跡や、固定群鎖などを、関数 StabChain が求めた結果から引きしています。

その後、木構造を定義して、帰納的に定義された関数 BacktrackSearchIND の実行に移ります。変数 I は、プログラムが固定部分群 $G^{(\mathtt{I})}$ の中を動いていることを示しています。変数 J は、帰納的に定義された関数 BacktrackSearchIND を帰納的に何回呼び出されているかを示しています。

```
BacktrackSearch:=function(G,f)
  local S, St, O, T, gH, H, BIs, Tr, tb, I, J, bi;
  S:=StabChain(G);;
  O:=[];; T:=[];; St:=S;; tb:="";
  while GroupStabChain(St)<>TrivialGroup(IsPermGroup) do
    Add(O,St.orbit);
    Add(T,St.transversal);
    St:=St.stabilizer;
  od;
  B:=BaseOfGroup(G);
  I:=[Size(B)];
  J:=0;
  gH:=[()];
  BIs:=List(Elements(G),x->List(B,i->i^x));
  Tr:=MakeTreeFromBaseImages(BIs);
  H:=Group(gH);
  bi:=[];
  BacktrackSearchIND(G,O,T,f,Tr,bi,gH,tb,I,J);
  return gH;
end;
```

こちらが、帰納的に定義された関数 BacktrackSearchIND です。最初の For 文で枝を 1 つずつ周りながら、直後の If 文で行く必要の無い枝を無視しています。

```
BacktrackSearchIND:=function(G,O,T,f,Tr,BI,gH,TAB,I,J)
  local tr, bi, x, H, obH, rH, tb, i, j;
  tb:=Concatenation(TAB,"\t");
  H:=Group(gH);
  obH:=Orbits(H,MovedPoints(G));
  rH:=List(obH,ob->ob[1]);
  i:=I[1];
  j:=ShallowCopy(J);
#
  Print(tb,"Base Image=",BI," in G^(",i,") with  Depth=",j,"\n");
  Print(tb,"Rep. of Orbits is ",rH,"\n");
#
  for tr in Tr do
    if not IsBound(BI[i]) or BI[i] in rH then
      bi:=ShallowCopy(BI);
      Add(bi,tr.pt);
      if tr.br<>[] then
        BacktrackSearchIND(G,O,T,f,tr.br,bi,gH,tb,I,j+1);
      else
        x:=BaseImageToElm(O,T,bi);
Print(tb,"\t(*) ",bi,"\t",x);
        if f(x) then
           Print("\t (NEW !!!)");
           if x<>() then
             Add(gH,x);
           fi;
           H:=Group(gH);
           obH:=Orbits(H,MovedPoints(G));
           rH:=List(obH,ob->ob[1]);
        fi;
        Print("\n");
      fi;
    fi;
  od;
  if I[1]>j then
   I[1]:=j;
  fi;
end;
```

では、実行例を示します。次の例では、元 $(1,3)(2,4)$ と可換かどうかを判定する関数 f を定義し、$(1,3)(2,4)$ の中心化群の生成元をを計算しています。

```
gap> f:=x->(1,3)(2,4)*x=x*(1,3)(2,4);; BacktrackSearch(G,f);
        Base Image=[  ] in G^(3) with  Depth=0
        Rep. of Orbits is [ 1, 2, 3, 4 ]
                Base Image=[ 1 ] in G^(3) with  Depth=1
                Rep. of Orbits is [ 1, 2, 3, 4 ]
                        Base Image=[ 1, 2 ] in G^(3) with  Depth=2
                        Rep. of Orbits is [ 1, 2, 3, 4 ]
                                (*) [ 1, 2, 3 ] ()        (NEW !!!)
                                (*) [ 1, 2, 4 ] (3,4)
                        Base Image=[ 1, 3 ] in G^(2) with  Depth=2
                        Rep. of Orbits is [ 1, 2, 3, 4 ]
                                (*) [ 1, 3, 2 ] (2,3)
                                (*) [ 1, 3, 4 ] (2,3,4)
```

```
                Base Image=[ 1, 4 ] in G^(2) with  Depth=2
                Rep. of Orbits is [ 1, 2, 3, 4 ]
                        (*) [ 1, 4, 2 ] (2,4,3)
                        (*) [ 1, 4, 3 ] (2,4)     (NEW !!!)
        Base Image=[ 2 ] in G^(1) with  Depth=1
        Rep. of Orbits is [ 1, 2, 3 ]
                Base Image=[ 2, 1 ] in G^(1) with  Depth=2
                Rep. of Orbits is [ 1, 2, 3 ]
                        (*) [ 2, 1, 3 ] (1,2)
                        (*) [ 2, 1, 4 ] (1,2)(3,4) (NEW !!!)
                Base Image=[ 2, 3 ] in G^(1) with  Depth=2
                Rep. of Orbits is [ 1 ]
                Base Image=[ 2, 4 ] in G^(1) with  Depth=2
                Rep. of Orbits is [ 1 ]
        Base Image=[ 3 ] in G^(1) with  Depth=1
        Rep. of Orbits is [ 1 ]
        Base Image=[ 4 ] in G^(1) with  Depth=1
        Rep. of Orbits is [ 1 ]
[ (), (2,4), (1,2)(3,4) ]
```

今回は、Backtrack Search アルゴリズムの内容を理解してもらうため、プログラムの中で全ての群の元を予めすべて求めていますが、base と strong generating sets さえ分かっていれば、元をこの順序で発生させることが可能ですので、実際の GAP 内の計算では、チェックの必要な元のみが構成されています。また、ここで紹介した 中心化群を計算するのプログラムは、中心化群の元が持つ特有の性質を利用していません。実際の計算では、個々の関数に対してその関数から得られる特有の性質を利用することで、チェックを必要とする元を更に絞りこんで、計算効率を上げています。

関連図書

[1] G. Butler, *Fundamental Algorithms for Permutation Groups*, Lecture Notes in Computer Science 559, Springer-Verlag.

[2] Peter J. Cameron, *Permutation Groups*, London Mathematical Society Student Texts 45.

[3] G. O. Michler, *On the construction of the finite simple groups with a given centralizer of a 2-central inolution*, J. Algebra **234** (2000), 668-693.

[4] M. Schönert et al., *GAP 3.4 Manual (Groups, Algorithms and Programming)*, RWTH Aachen, Aachen, Germany, 1994.

[5] J. G. Thompson, *Finite-dimensional representations of free products with an amalgamated subgroup*,
J. Algebra **69** (1981), 146-149.

[6] Gerhard O. Michler, Katsushi Waki, Michael Weller, *Natural Existence Proof for Lyons Simple Group*, Journal of Algebra and its Applications (to appear)

索 引

?, 6
#, 18
->, 13
\\, 13
\n, 13

AbelianGroup, 9
Action, 10, 24
ActionHomomorphism, 27
Add, 3
AllGroups, 5, 10
AllPrimitiveGroups, 5, 10
AllSmallGroups, 5
AllTransitiveGroups, 5
Append, 3
AppendTo, 5
AsList, 24
AsVectorSpace, 7

base, 31
base image, 35
BaseOfGroup, 33
Basis, 7, 16
BasisVectors, 7
Blocks, 25, 27

CategoriesOfObject, 17
Centralizer, 22
Centre, 21
ClosureGroup, 28
Coefficients, 7
CommutatorSubgroup, 20
Concatenation, 3, 15
ConjugacyClass, 29
ConjugacyClasses, 24
ConjugateSubgrup, 22
CyclicGroup, 9

DerivedSeries, 21
DerivedSubgroup, 15
DihedralGroup, 9
Dimension, 7
Display, 16
DisplayCompositionSeries, 10
DownEnv, 18

E, 7
ElementaryAbelianGroup, 9
else, 14
Error, 17
Exec, 5
ExtraspecialGroup, 9

Factorized Inverse Transversal, 32
FactorsInt, 21
FieldOfMatrixGroup, 10
Filtered, 3, 10
First, 24
for, 14
ForAll, 3
ForAny, 3
FreeGroup, 8
function, 10, 13

GF, 7
GroupStabChain, 33
GroupStbChain, 32

if, 14
Image, 23
Index, 21, 26
InputLogTo, 5
Intersection, 22, 24, 33
IsAbelian, 15, 21
IsConjugate, 27
IsElementaryAbelian, 22
IsMatrixGroup, 9
IsMutable, 16
IsomorphismTypeFiniteSimpleGroup, 23
IsPcGroup, 9
IsPerfect, 21
IsPermGroup, 9, 24
IsPrime, 3
IsPrimitive, 25, 27
IsRegular, 28
IsRightCoset, 23
IsSimple, 10, 23
IsSimpleGroup, 24
IsSubgroup, 24

IsTransitive, 26

Kernel, 23

last, 21
Length, 25
List, 2, 24
ListWithIdenticalEntries, 3
local, 15
LogTo, 5
LowerCentralSeries, 21

NamesUserGVars, 14
NaturalHomomorphismByNormalSubgroup, 23
Normalizer, 21, 22
NrMovedPoints, 10

OnLines, 10
OnPairs, 26
OnSets, 27
OnTuples, 28
Operation, 24
Orbit, 26
OrbitLength, 29
Orbits, 26
Order, 24

PcGroupFpGroup, 9
PerfectGroup, 5
PreImages, 23, 27
Print, 13
PrintTo, 5

quit, 16

Random, 22
Rationals, 8
Read, 5
RecNames, 4
Record, 4
reduced, 31
Representative, 24
RepresentativeAction, 29
return, 13

SaveWorkspace, 5
Schreier tree, 32
Schreier Vectors, 32
SetName, 22
SGS, 31
ShallowCopy, 16
Size, 21
SmallGeneratingSet, 20
StabChain, 32
Stabilizer, 21, 26
strong generating set, 31
StrongGeneratorsStabChain, 33
Subspace, 8
SylowSubgroup, 21

tab キー, 6
TrivialGroup, 9, 32
Tuples, 26

UnderlyingExternalSet, 25

VectorSpace, 8

Where, 18
while, 19

Z, 7
Zero, 17