GAP の遊び方

庄 司 俊 明

名古屋大学大学院多元数理科学研究科

目　　次

1章　GAP World への招待 **1**

1.1　GAP の始め方と終り方 1
1.2　GAP による計算記録の残し方 2
1.3　GAP による簡単な計算 2
1.4　群演算の簡単な例 4
1.5　変数とオブジェクト 5
1.6　関数 6

2章　リストとレコード **7**

2.1　簡単なリスト 7
2.2　リストのコピー 10
2.3　集合 11
2.4　変域 12
2.5　For ループと While ループ 13
2.6　リストに対する操作 16
2.7　ベクトルと行列 17
2.8　簡単なレコード 21

3章　さらに関数について **22**

3.1　関数の書き方 22
3.2　If 構文 23
3.3　局所変数 24

4章　ファイルを使いこなす **25**

4.1　Read 25
4.2　PrintTo 26
4.3　AppendTo 26

5 章　群と準同型 **27**

5.1　置換群 27
5.2　群の共役類 33
5.3　ブロックと置換表現 36

6 章　ルービック・キューブ **41**

6.1　ルービック・キューブ群の導入 41
6.2　頂点ルービック・キューブ群 43
6.3　辺ルービック・キューブ群 47
6.4　ルービック・キューブ群の決定 50
6.5　生成元による表示 52
6.6　拡張版ルービック・キューブ $(3 \times 3 \times 3)$ 55

GAP の遊び方

GAP (Group, Algorithm and Programming) は, 環, 体などの抽象代数の計算のための専用ソフトウエアです. 無料で公開されています. ここでは GAP4 のバージョンを使います. GAP により有限群や, 有限体, 有理数体, 代数体, あるいはその上のベクトル空間における種々の計算が可能です. あくまで整数計算が基本ですから実数体や複素数体での数値計算には向きません.

GAP を個人的に取り寄せたい場合は, 以下のサイトから ダウンロードできます.

```
http://www-gap.dcs.st-and.ac.uk/gap
```

とそのミラーサイト

```
http://www.math.rwth-aachen.de/~GAP

http://www.ccs.neu.edu/mirrors/GAP

http://wwwmaths.anu.edu.au/research.groups/algebra/GAP/www/
```

1 章 GAP World への招待

1.1 GAP の始め方と終り方

GAP をスタートさせるには、ターミナル上で gap とタイプすれば始まる.

```
% gap
```

ここで return key を押すと, GAP のロゴと共に

```
gap>
```

が画面に表れて, 入力待ちの状態になる.

GAP を終らせるためには, quit; とタイプする. 重要なことは, 命令を書いたら必ず最後にセミコロン ; をつけることである. GAP では return key は命令の終了ではなく, 単に命令待ちを意味する.

```
gap> quit;
%
```

となり, GAP が終了する. 機種によっては, quit; の代わりに *ctl*-D (コントロールキーを押しながら D を押す) によっても終了できる. 試してみて下さい.

1.2 GAP による計算記録の残し方

GAP は対話型のソフトウエアなので, 画面上でコンピュータと交信しながら計算を進めて行く. その計算結果を残しておかないと, GAP を終了したときに計算結果が失われてしまう. 簡単に記録を残す方法は以下のようにログファイルに結果を書き込むことである.

```
gap> SizeScreen( [ 72, ] ); LogTo( "sample.log" );
```

これは, セミコロン ; で区切られたふたつの命令を表す. 最初の命令は, 計算結果を一行 72 のサイズで画面に打ち出すこと, 次の命令は, 画面に表れる文字や数字をそのままの形で sample.log というファイルに書き出すこと, である. 二つの命令は全く独立に書いても構わない. 計算に対する指令から計算結果まで, quit; により GAP が終るまで画面が全て sample.log に保存される. GAP を終えた後、このファイルを参照すれば, 計算結果が復元される. このファイルは編集可能なので, 後にファイルの不必要な部分を消し去れば有効にデータが保存される. なお, GAP を何回か動かしそれを同じ名前のログファイルに保存していけば新しいデータがファイルにどんどん追加されて行くことになる.

1.3 GAP による簡単な計算

まず整数の足し算, 引き算, かけ算をやってみよう. 例えば, $(9-7)\times(5+6)$ は (9 - 7) * (5 + 6) とタイプする. GAP は以下のように計算結果を返す.

```
gap> (9 - 7) * (5 + 6);
22
gap>
```

ここでもし, 最後のセミコロンを忘れて return key を押すと GAP は > のみを出力し沈黙を守る. GAP は更なる入力をひたすら待ち続けるのである. そこで, > の後に, セミコロン; を入力すると,

```
gap> (9 - 7) * (5 + 6)
>;
22
```

と以前の計算が復活する. しかし, 例えば最後のカッコ) を忘れたりすると, これは少し罪が重い. GAP は「ご主人様ごめんなさい, できません」 というわけで,

```
gap> (9 - 7)*(5 + 6;
Syntax error: ) expected
(9 - 7)*(5 + 6;
              ^
```

と, 間違いの箇所を推測し, その位置を指摘してくれる. Syntax error とは文法上の間違いである.

時に, 文法上のエラーから無限に続く迷路に入ってしまうことがある. GAP は, 決して見つかることのない答を求めて闇をさすらい, 無益な計算をし続けるのである. これをブレイクループ (break loop) (日本語では,「悪循環」??) といい, brk> という表示が出る. ブレイクループの中で, さらにエラーが起これば, brk_02>, brk_03> というように事態はどんどん悪化していく. ブレイクループから抜け出すには,

```
brk> quit;
```

とすればよい. あるいは ctl-D としてもよい. GAP のプロンプト gap> が復活する.

次に割算について見てみよう. 前にも書いたように, GAP は小数の計算には興味を示さない. あくまで, 分数の世界にこだわるのである . GAP は, 与えられた分数を約分し, 既約分数で表す. さらに, 頭の良いことには, 分数の足し算をやらせれば、通分して答えを既約分数の形で返してくる.

```
gap> 12345/25;
2469/5
gap> 123/4567 + 234/5678;
883536/12965713
```

巾乗の計算, 例えば 3^{132} は 3^132 とタイプする.

```
gap> 3^132;
955004950796825236893190701774414011919935138974343129836853841
```

整数の割算による余りは mod で計算できる. 17 を 3 で割ると余り 2 となるので

```
gap> 17 mod 3;
2
```

が得られる.

また, GAP は命題や数式が正しいか (true), 間違っているか (false) を判定する. これを論理演算という.

```
gap> (9 - 7)*5 = 9 - 7*5;
false
```

数の大小の比較には, =，<>，<，<=，>，>= が使われる. それぞれ, $a = b, a \neq b$, $a < b, a \leq b, a > b, a \geq b$ の意味である. 論理演算とあわせて,

```
gap> 10^5 < 10^4;
false
```

より複雑な論理演算は次のようになる.

```
gap> not true; true and false; true or false;
false
false
true
```

それぞれ,「正しい」の反対は間違い,「正しくて、同時に間違い」ということはない,「正しいかまたは間違っている」というのは常に正しい, を意味する. それでは, 次はどうなるか. 答えを予測してからやってみよう.

```
gap> not true or false and not not false or true and false;
```

GAP では, 文字を定数としてそのまま扱うことができる. 以下に述べる文字変数とは区別する必要がある. 文字 a や $*$ を定数として使いたい場合は'a' あるいは '*' と両側から' ではさむ.

```
gap> 'a';
'a'
gap> '*';
'*'
```

ただし、'ab' とはできず, 一文字のみ. GAP は文字定数が等しいか異なるかを判定する.

1.4 群演算の簡単な例

対称群 S_n の元 σ は整数 $I_n = \{1, 2, \ldots, n\}$ の置換

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & n \\ \sigma(1) & \sigma(2) & \cdots & \sigma(n) \end{pmatrix}$$

として表される. $\sigma \in S_n$ は共通な文字を含まない巡回置換の積として一意的に表される. 例えば,

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 1 & 3 \end{pmatrix} = (124)(35).$$

ここで例えば (124) は $1 \to 2 \to 4 \to 1$ となる巡回置換を表す.

GAP では, 対称群の計算がそのままの記号で計算できる. (1,2,3) と入力すると, GAP はこれを S_3 の巡回置換 $\sigma = (123)$ として認識する. σ の逆元 σ^{-1} は巡回置換 (132) になるが, GAP ではこれを

```
gap> (1,2,3)^-1;
(1,3,2)
```

と計算してくれる. 見れば分かるように GAP では^-1 を -1 乗, すなわち逆元として素直に理解してくれる (普通なかなかそうは行かない).

文字 2 に $\sigma = (123)$ が作用して, $\sigma(2) = 3$ となるのは, GAP では

```
gap> 2^(1,2,3);
3
```

と計算される. $\sigma(2) = 3$ の代わりに $2^{\sigma} = 3$ と書くことに注意する. 本来, ${}^{\sigma}2 = 3$ と書かないと左作用にならないが, 後者の書き方はタイプしにくいので, 前者を採用したい. そのために, S_n の I_n への作用は右作用を考える. この場合, S_n の積の計算は左から右へとすることになる. たとえば

```
gap> (1,2)*(2,3);
(1,3,2)
```

となる. 左作用で計算すれば $(12)(23) = (123)$ となるはずである.

また, 数の列 $(1, 2, 3)$ に $\sigma = (1, 2)$ を置換として作用させると,
$(1, 2, 3)\sigma = (\sigma(1), \sigma(2), \sigma(3)) = (2, 1, 3)$ となるが, GAP では

```
gap> (1,2,3)^(1,2);
(2,1,3)
```

となる.

1.5 変数とオブジェクト

GAP では変数を定義し, その変数に何か (数とは限らない) ある物を代入 することができる. 一般に, 変数に代入される物のことを オブジェクト という. 例えば, 文字 a を変数として定義する場合, `a:=` と書く. (`a :=` でもよい. ただし `: =` の様に, 間にスペースを入れてはいけない).

```
gap> a:= (9 - 7)*(5 + 6);
22
gap> a;
22
gap> a*(a+1);
506
gap> a = 10;
false
gap> a:= 10;
10
gap> a*(a+1);
110
```

最初の式で, 変数 `a` が定義され, そこに $(9-7) \times (5+6) = 22$ が値として代入される. 2 番目の式のように, 変数 a を呼び出せば, GAP はその値 22 を返す. 3 番目の式では, $a = 22$ として $a(a+1)$ を計算し, その値 506 が得られる. また a に新しい値 10 を代入するには, `a:= 10` と書けばよい. ここで `a = 10` では代入にならないことに注意する. GAP では `=` は等式が正しいか間違っているかの判定に使われる. 4 番目の場合, a には値 22 が入っているので, $a = 10$ は誤りであり, GAP は `false` と答える.
GAP では, 値を代入すると常に次の行にその値を書く.

```
gap> w:= 2;
2
```

いちいち変数の値を書く必要がない場合 (例えばプログラムを作る場合など), 次のようにセミコロンを 2 つ続けて書くと, 変数の値が次の行に書かれない.

```
gap> w:= 2;;
gap> w*(w+2);
8
```

GAP では, 基本的にまぎらわしくない限り, どんな文字や数字も変数として使える. たとえば abc や a0bc1. また, GAP は大文字と小文字を区別するので, a1 と A1 は異なる変数を表わす. しかし 1234 は数字 1234 と区別がつかないので, 変数としては使えない.

また 例えば quit は, 終了のコマンドとして登録されているので変数としては使えない. このような言葉は "キーワード" として区別される. さらに, GAP はそのライブラリに多くの変数や, 関数を持っている. これらの言葉は変数として使えない. しかし, GAP が定義している関数はすべて大文字で始まる (次節参照. 例えば, Factorial, Gcd, Print など). そこで小文字で始まる変数を定義する限りは, GAP の変数や関数と重複する心配はない.

1.6. 関数

GAP の言語で書かれたプログラムを関数という. 関数を GAP のオブジェクトに適用すると, 新しいオブジェクトが返される. GAP における関数は数学的な意味の関数を含む、はるかに一般的なものである. 関数の例としては, 例えば Factorial( ) がある. Factorial(n) は非負整数 n に対して n の階乗 $n! = 1 \cdot 2 \cdots (n-2)(n-1)n$ を与える.

```
gap> Factorial(50);
30414093201713378043612608166064768844377641568960512000000000000
```

すなわち $50!$ の値を一瞬で答える. 関数には常に ( ) が伴い, そこにいくつかのオブジェクトを書くことにより, 関数の答 (オブジェクト) が返って来る. 例えば, Gcd(,) は 2 つの整数の最大公約数を与える.

```
gap> Gcd(1234, 5678);
2
```

関数によってはオブジェクトを返す代わりに、何らかの動作をするものもある. 例えば, 関数 Print( ) は計算結果を画面に打ち出させる時に使われる.

```
gap> Print(1234, "\n");
1234
```

ここで後ろの "\n" はデータを画面にプリントした後の改行の命令である. これがないと以下のように計算結果と同じ行に gap> のプロンプトが現れてしまう.

```
gap> Print(1234);
1234gap>
```

自分で関数 (この場合は一変数の関数) を定義する手軽な方法は写像 (対応) を表す矢印 -> を使うものである. 例えば関数 $f(x) = x^3$ を与える関数 $\mathrm{f} = \mathrm{cubed}$ を定義するには, `cubed:= x -> x^3;` を打ち込む ("cubed" は立方の意味).

```
gap> cubed:= x -> x^3;
function( x ) ... end
```

GAP は「関数 `function( x )` が定義できました. よろしく」と答える (2 行目). 以後, 3 乗は `cubed(x)` で計算できる.

```
gap> cubed(5);
125
gap> cubed(37);
50653
```

より複雑な関数を定義する方法については後に説明する.

2 章 リスト と レコード

群や環を扱う基礎になるのは集合であり, そもそもコンピュータで集合をどのように表現するかというのが基本的な問題になる. GAP では以下に説明するように, 集合をリストの特別な場合として扱う. そのため, リストの考え方とその使い方に慣れることが要求される.

2.1 簡単なリスト

コンマで区切られ, 4 角括弧でまとめられたオブジェクトの集まりをリストという. たとえば, 素数を小さい順に 10 個ならべたものを `primes` と名前をつけてリストにする.

```
gap> primes:= [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

リストはデータが一定順序でならんだ表 (リスト) だと思っていいが、そこに新しいデータを付け加える 2 通りの関数がある. 一つはリストの最後にリストをつなげるもので関数 `Append` が使われる. たとえば, 29 の次の素数は 31 と 37 であるが, `primes` の後ろにリスト `[31, 37]` をつけ加えて,

```
gap> Append(primes, [31, 37]);
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 ]
```

関数 `Append` はリスト `primes` にデータ 31, 37 を加えたものを新しいリスト `primes` として返す. 一方, リストの最後に一つの要素をつけ加えるのには, 関数 `Add` を使う.

```
gap> Add(primes, 41);
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 ]
```

リスト primes は数列 primes[n], あるいは整数 $1, 2, \ldots, n, \ldots$ 上の関数 primes[ ] とみることもできる. たとえば, primes の最初から 7 番目の項 17 は

```
gap> primes[7];
17
```

として得られる. primes[7] は値として扱うことができる. 例えば, 3 倍したり, 関数に代入したりできる.

```
gap> primes[7]*3;
51
gap> Factorial(primes[7]);
355687428096000
```

primes[ ] により, 指定された位置にデータを置くことができる. たとえば, 41 の次の素数は 43 なので, 43 はリストの最後の位置につけ加えられる. そこでまず関数 Length により, リストの最後の位置が何番目かを知り, その次の位置に 43 を入れる.

```
gap> Length(primes);
13
gap> primes[14]:= 43;
43
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 ]
```

リストの途中を飛ばして, リストに値を代入することもできる. たとえば 20 番目の素数は 71 なので, primes[20] に 71 をしよう. すると

```
gap> primes[20]:= 71;
71
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> Length(primes);
20
```

となる. 余分なコンマが 5 個ならんでいるが, これは primes [15] から primes[19] までの 5 個に値が定義されていないことを示している. そして, primes は長さ 20 を持つリストになる.

CAP におけるリストは普通のプログラム言語の配列 (Array) に相当する. しかし配列があらかじめ n の上限 (配列の次元) を決めておかなければならないのに対し, リストは一度定義されてしまえば, どんな番号もとり得る. これは不精な人間にはおおいに助かるが, それ以上に素数の列 primes[n] のように無限に続く列を扱う場合に威力を発揮する.

しかし, ある番号での値を指定するためには, その前にリストが定義されていなければならないということに注意する. たとえば, リスト lll が定義されていない段階で 1 番目の値を指定すると

```
gap> lll[1]:= 2;
Variable: 'lll' must have a value
```

「リスト lll は値を持たない. 何を考えておるのか」と文句を云われる. これを避けるためには, まず空集合 [ ] としてリスト lll を定義しておく. 長さは 0 になる.

```
gap> lll:= [ ];
[ ]
gap> Length(lll);
0
```

その後は, 普通に値を取らせることができる.

```
gap> lll[1]:= 2;
2
```

リストに含まれるオブジェクトは, どんなタイプでもまた異なったタイプのものが混ざっても構わない. 例えば

```
gap> lll:= [true, "This is a String",,, 3];
[ true, "This is a String",,, 3 ]
```

またあるリストが他のリストの要素になることもできる. もっと, とんでもないことには, リスト自身をそのリストの要素として入れることもできる (再帰リスト).

```
gap> lll[3]:= [4,5,6];; lll;
[ true, "This is a String", [ 4, 5, 6 ],, 3 ]
gap> lll[4]:= lll;
[ true, "This is a String", [ 4, 5, 6 ], ~, 3 ]
```

2 番目の命令はリスト lll の 4 番目の位置にリスト lll 自身を代入せよというものであり, 結果は最後の式のようになる. ここで 4 番目の tilde ~ は, そこに全体と同じものが入っていることを示している. これはちょっと分かりにくいが, 例えば落語に出てくる, 頭のてっぺんにくぼみがあってそこに雨がたまり, やがて池ができ, その自分の頭の池に飛び込んで自殺する男の話を思い起こせば, 理解も深まるだろう. あるいはドラゴンボールで, 悟空とベジータが魔人ブーの体内で暴れている所にブー自身も登場する場面を思い出すのも良い. そこにブーが現れたとすると, その現れたブーの体内にも悟空とベジータがいるはずで, そこへまたブーが現れることになり, ... とこれもまた再帰リストである. このように再帰リストは我々の身近なところに転がっている.

"This is a String" のように文章の両端を " で囲ったものをストリング (ひも) という. ストリングは次のようにして, 特殊なリスト (途中に穴のない, つまりどんな番号にも値の入っている, 文字記号のみからなるリスト) として扱われる.

```
gap> s1:= ['H', 'a', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'];
"Hallo world"
gap> s1 = "Hallo world";
true
gap> s1[7];
'w'
```

リストが与えられたとき, その部分リストを項の番号を中括弧 { } で指定することにより定義する.

```
gap> s1 := lll{[1,2,3]};
[ true, "This is a String", [ 4, 5, 6 ] ]
gap> s1{[2,3]} := ["New String", false];
[ "New String", false ]
gap> s1;
[ true, "New String", false ]
```

最初の指令は, リスト lll の最初の 3 項からなる部分リストを s1 とせよ, ということであり, 次は, リスト s1 の 2 番目と 3 番目を新しいデータで置き換えよということである.

2.2 リストのコピー

リストが与えられているとき, 同じリストを作るには別の名前のリストに前のリストを代入すればよい.

```
gap> numbers := primes;; numbers = primes;
true
```

これで新しい numbers というリストに, primes のデータがすべて入ったことになる. ところが, こうして得られたリスト numbers はもとのリスト primes と一身同体, 切っても切れない関係になっている. (数学用語では inseparable = 別れられない = 非分離的という) になっている. primes の値を変えれば, numbers もそれに応じて変わり, numbers を変えれば primes も変わる. コピーを作るのは大抵もとのデータを利用して新しい表を作るような場合だから, そのときに元の表まで変わってしまっては, すこぶる不便である.

```
gap> numbers[3]:= 100;; numbers = primes
true
gap> primes[3];
100
```

このような事態が起こるのは, 以下の事情による. そもそもリスト primes を定義するということは, コンピュータのメモリーのある番地に名前 (primes さん) をつけて, primes 関係のデータはすべてそこに保存しておくということである. ところが, numbers := primes とすると, 新しい番地は用意されず numbers さんの住所も primes さんと同じにされてしまう. これでは, 2 人が同じ銀行口座を持っているようなものだから, numbers さんがお金を使うと primes さんの貯金も減ってしまうというわけである.

独立した家を持つにはどうしたらいいだろうか. 実は ShallowCopy という関数が用意されている. 深い関係になると離れられないから, 浅いおつきあいにしましょうね, ということでしょうか.

```
gap> primes[3]:= 5;; primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> numbers:= ShallowCopy(primes);; numbers = primes;
true
gap> numbers[3] := 100;; numbers = primes;
false
gap> primes[3];
5
```

ShallowCopy によって生み出された新しいリストは, もとのリストと別の番地を持ち, 独立なリストを作る. primes さんのもとを離れた numbers さんは, かくして新しい人生の一歩を踏み出すことになる.

2.3 集合

GAP では, 穴のないリスト (これを密なリストと云う) で, 重複がなく, 決められた順番に並んでいるものを集合 という. ただし, この場合リストを構成してるデータが同じ性質のものである必要がある. 例えば成分が数の場合, [2，3，2，5，4] はリストであるが、これを集合としてみると, 重複を除き小さい順に並べて [2，3，4，5] が集合になる. GAP には, リストが重複なく小さい順に並んでいる (strictly sorted list) かどうかを判定する関数 IsSSortedList がある. これが正しい時, リストは集合として扱われる. また与えられたリストから集合を作る関数 Set がある.

```
gap> number := [ 2, 3, 2, 5, 4];
gap> IsSSortedList(number);
false
gap> number := Set(number);
[ 2, 3, 4, 5 ]
gap> IsSSortedList(number);
true
```

関数 Set(number) はリスト number を変えないことに注意しておく. Set としての number を得るには, 上記のように置き直さなければならない.

上と同様のことが, アルファベットのストリングをデータとするリストについても成立する. この場合, 順番はアルファベットに関する辞書式順序である.

```
gap> animals := ["tiger", "puma", "giraffe", "panther", "tiger" ];
[ "tiger", "puma", "giraffe", "panther", "tiger" ]
gap> animals := Set(animals);
[ "giraffe", "panther", "puma", "tiger" ]
gap>IsSSortedList;
true
```

一方, ある元が集合に含まれるかどうかは in により判定できる.

```
gap> "giraffe" in animals;
true
gap> "jackal" in animals;
false
```

集合に新しい元を付け加えるには, 関数 AddSet を使う. AddSet はその元が集合に含まれているかどうかを判定し, 含まれていなければ, その元を順番にしたがって適正な位置に挿入した集合を返す. もし, その元がすでに含まれていれば, もとと同じ集合を返す.

```
gap> AddSet(animals, "jackal");
gap> animals;
[ "giraffe", "jackal", "panther", "puma", "tiger" ]
gap> AddSet(animals, "panther");
gap> animals;
[ "giraffe", "jackal", "panther", "puma", "tiger" ]
```

"jackal" がちゃんと 2 番目に入っていることに注目しよう. 3 番目の命令では, "panther"はすでに含まれているので集合 animals は変化しない.

集合が定義できると, 次は集合算であろう. GAP はこれも苦もなくこなす. GAP には関数 Intersection (共通部分) と Union(和集合) が用意されている.

```
gap> wildcats := ["tiger", "panther", "lion", "cheetah" ];
[ "tiger", "panther", "lion", "cheetah" ]
gap> Intersection(animals, wildcats);
[ "panther", "tiger" ]
gap> Union(animals, wildcats);
[ "cheetah", "giraffe", "jackal", "lion", "panther", "puma", "tiger" ]
```

wildcats は集合ではなくリストである (順番がそろっていない). それにも拘らず, 関数 Intersection や Union の返す値は集合 (順番がそろっている) になっていることに注意する. これは表の整理をするとき上手に使えば役に立つだろう.

2.4 変域

GAP では, 変域 (Range) とは整数のなす (有限) 等差数列であり, 特別なタイプのリストに他ならない. この等差数列を表わすのに, リストとしてすべての元を書く代わりに, [初項，第 2 項 .. 最終項] と書く簡便法がある. 変域は次の節で説明するループ演算との関連で重要であるが, 変域の典型的な例としては, 1 から始まり, $1, 2, \cdots, 1000$ と 1 ずつ増加して, 特定の数 1000 まで進む等差数列である. このように, 等差が 1 の場合には, 第 2 項を省略して, [1..1000] と簡便に表す. 以下に例をあげる.

```
gap> [1..99999];
[ 1 .. 99999 ]
gap> [1,3..99999];)
[ 1, 3 .. 99999 ]
gap> Length (last);
50000
gap> [99999,99997..1];
[ 99999, 99997 .. 1 ]
```

1 番目は等差 1 の変域, 2 番目は等差 2, 最後は等差 -2 の変域である. ここで, Length(last) と書いたのは, すぐ前の値 (=last) を関数 Length に代入せよという意味で. 前の値を定義することなく参照できるので便利な記法である.

上に説明した変域 [1..9999] は, あくまで記法であるから, 変域自体は [1, 2, 3 .. 999] という形のリストに過ぎない. GAP には, 与えられたリストが変域であるかどうかを判定する関数 IsRange がある. また変域がリストとして与えられた場合にその表記を上のような簡便な形に表す関数 ConvertToRangeRep がある.

```
gap> a:= [-2,-1, 0, 1, 2, 3, 4, 5];
[ -2, -1, 0, 1, 2, 3, 4, 5 ]
gap> IsRange(a);
true
gap> ConvertToRangeRep( a );; a;
[ -2 .. 5 ]
gap> a[1] := 0;; IsRange(a );
false
```

2.5 For ループと While ループ

for ループと While ループ を上手に使いこなすことがコンピュータによる計算の決め手になる. まず, for ループから説明しよう. pp を置換からなるリストとし, そのリストに含まれているすべての置換を掛け合わせることを考える. GAP では for ループを使って次のようにやる.

```
gap> pp:= [(1,3,2,6,8)(4,5,9), (1,6)(2,7,8), (1,5,7)(2,3,8,6),
>          (1,8,9)(2,3,5,6,4), (1,9,8,6,3,4,7,2)];;
gap> prod:= ();
()
gap> for p in pp do
>       prod:= prod*p;
>    od;
gap> prod;
(1,8,4,2,3,6,5,9)
```

1 行目の命令でリスト pp を定義している. 一行に書き切れないときは, return key を押して次の行以下に書いていけばよい. セミコロン ; を打ち込まない限り, GAP は唯 > を返すだけで、一続きの命令だと解釈する. リストのデータである置換はすべて巡回置換の積で表されていることに注意する. さて変数 prod を使って積を計算する. 初期値として恒等置換 ( ) を入れておく. 5 行目から 7 行目までが for ループである. p をループの変数とし, リスト pp に含まれる元を小さい順に一つ一つ取りだし, 変数 prod に右からかけていく. do がループの開始命令で, do を逆にした od; がループの終りを意味する. コンピュータは do と od; の間を p の条件にしたがって, ぐるぐる回ることになる.

For ループの一般的な書式は,

```
for 変数 in リスト do 命令 od;
```

である. for ループにより, リストに含めれるすべての元に対して命令が実行される (リストに穴があっても, そこは飛ばしてやってくれる. 後で分かるように, これがまあ実に気の効いた配慮なのである). ただし, for ループは実行するだけで値は返さないから, あらかじめ計算結果を記録する変数や関数を用意しておかなければならない. 先ほどの例では 変数 -> p, リスト -> pp, 命令 -> prod := prod*p; であって, 計算結果が変数 prod に書き込まれた.

一般のプログラム言語では, for ループは次の形を取るのが普通である.

```
for 変数 from 初項 to 最終項 do 命令 od;
```

GAP の場合, これは単に「変域」という特別なリストを考えているのに過ぎない. 2.4 節で述べた, 変域を表す簡単な記法にしたがえば, この特別な場合は,

```
for 変数 in [初項 .. 最終項] do 命令 od;
```

と書くことができる. 例えば, 50! を for ループを使って計算するには以下のようにする.

```
gap> ff:= 1;
1
gap> for i in [1..50] do
>          ff:= ff*i;
>    od;
gap> ff;
30414093201713378043612608166064768844377641568960512000000000000
```

ここで, for ループを使って 1000 以下の素数のリスト primes を作るプログラムを紹介しておく.

```
gap> primes := [];;
gap> numbers := [2 .. 1000];;
gap> for p in numbers do
>      Add(primes, p);
>      for n in numbers do
>         if n mod p = 0 then
>         Unbind(numbers[n-1]);
>         fi;
>      od;
```

```
>    od;
```

ここで使っている方法は「エラトステネスのふるい」である. 2 から 1000 までの数のリスト numbers を用意する. まず 2 は素数だから, 素数のリスト primes の最初におく. 次に numbers から 2 で割れる数を全部排除する. 残っている数で 2 の次は 3. そこで 3 を primes のリストに加え, numbers から 3 で割れる数を全部排除する. numbers に残っている次の数は 5. そこで 5 を primes に加え, ... この操作を繰り返すことにより素数のリスト primes が得られる. プログラムの中に if が出てきたが, これは fi: と組になって条件文「if 命令」を構成する. より複雑な if 命令については次章 3.2 節で解説する.

次に while ループについて説明する. while ループの書き方は

```
while 条件 do 命令 od;
```

の形をとる. While ループは条件がみたされる限り, 命令の上をループで回り続ける. for ループと同様に while ループは od; により終了する.

次の例は, 与えられた整数の約数となる素数をすべて求めるプログラムである. ここでは, 素数の表 primes を利用して計算する. for ループと while ループがうまく組み合わされているのに注目して欲しい.

```
gap> n := 1333;;
gap> factors := [];;
gap> for p in primes do
>         while n mod p = 0 do
>              n := n/p;
>              Add(factors, p);
>         od;
>    od;
gap> factors;
[ 31, 43 ]
gap> n;
1
```

素数の表がある限り, n := 1333 を変えることによりどんな数 n に対しても素因数が決定できる. while ループは素数 p を固定したとき, その数が p で何回割れるか, すなわち n の素因数に p が何個現れるかを計算するのに使われている.

素数の表を利用するもう一つの例として, 双子素数を求めるプログラムを載せておく. p と $p+2$ が共に素数のとき, $(p, p+2)$ を双子素数という. 双子素数は無限個存在すると予想されているが, まだ証明はされていない.

```
gap> twins := [];;
gap> for p in primes do
>          if p+2 in primes then
>               Add(twins, [p, p+2]);
>          fi;
>    od;
```

```
gap> twins;
[ [ 3, 5 ], [ 5, 7 ], [ 11, 13 ], [ 17, 19 ], [ 29, 31 ], [ 41, 43 ],
[ 59, 61 ], [ 71, 73 ], [ 101, 103 ], [ 107, 109 ], [ 137, 139 ],
[ 149, 151 ], [ 179, 181 ], [ 191, 193 ], [ 197, 199 ] ]
gap> Length(twins);
15
```

この例では, 双子素数の表 twins はリスト [p, p+2] を成分とするリストになっている. また後半のリストは, primes を 200 以下の素数として, twins を計算させた結果である. 200 以下の素数に対しては, 15 個の双子素数が存在することが分かる.

さらに, $p, p+6, p+12$ が素数のとき, $(p, p+6, p+12)$ を三子素数ということにすると, 三子素数は次のように計算される.

```
gap> trios := [];;
gap> for p in primes do
   >     if p+6 in primes then
   >        if p+12 in primes then
   >           Add(trios, [p, p+6, p+12]);
   >        fi;
   >     fi;
   >   od;
gap> trios;
[ [ 5, 11, 17 ], [ 7, 13, 19 ], [ 11, 17, 23 ], [ 17, 23, 29 ],
[ 31, 37, 43 ], [ 41, 47, 53 ], [ 47, 53, 59 ], [ 61, 67, 73 ],
[ 67, 73, 79 ], [ 97, 103, 109 ], [ 101, 107, 113 ],
[ 151, 157, 163 ], [ 167, 173, 179 ] ]
gap> Length(trios);
13
```

2.6 リストに対する操作

2.5 節で置換からなるリスト pp を定義し, pp に含まれる元の積を計算した. 実は GAP には, リストに含まれる元の積を計算する関数 Product が既に用意されている. これを用いると,

```
gap> Product(pp);
(1,8,4,2,3,6,5,9)
gap> Product([1..15]);
1307674368000
```

2 番目の例は整数のリスト [1..15] に関数 Product を適用している. これで, $15! = 1307674368000$ が計算できる. 一般に関数 Product は, そのリストの元に積が定義されている限り, リストの元すべてにわたる積を計算するのである.

一方, リストの元に和が定義されていれば, 同様の関数 Sum が使える. 例えば, $1+2+\cdots+100=5050$ は以下のように書ける.

```
gap> Sum([1..100]);
5050
```

他によく役に立つ関数として List がある. 関数 List はリストと関数を変数として持ち, その関数をリストの元に作用させて得られるリストを値として返す. 例えば, 以前定義したように cubed を関数 $f(x)=x^3$ とすると

```
gap> cubed:= x -> x^3;;
gap> List([2..10], cubed);
[ 8, 27, 64, 125, 216, 343, 512, 729, 1000 ]
```

すなわち, 2 から 10 までの数 x に対し, x^3 を並べたリストが得られる. 関数 Product や Sum は, 変数として, リスト [2..10] と関数 cubed を取ることもできる.

```
gap> Product([2..10], cubed);
4778472583987200000
gap> Sum([2..10], cubed);
3024
```

それぞれ, 関数 cubed から得られたリスト [8, 27, ..., 729, 1000] に対する積と和を計算している.

素数のリスト primes が例えば 1000 以下の素数に対して定義されているとして, その中から一定の条件にあう元を取り出して新たにリストを作りたい. このようなとき関数 Filtered が使える. 関数 Filtered はリストと, 元を取り出す性質を与える関数とを, 変数として持つ. 例えば, 30 以下の素数を取り出したければ, 関数 x -> x < 30 を使う.

```
gap> Filtered(primes, x -> x < 30);
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

以前述べた部分リストを作る操作{ }を思いだそう. この操作はリストの位置を変数として, その位置におけるリストの元を値として返す.

```
gap> primes{[1..10]};
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

2.7 ベクトルと行列

GAP では, 行ベクトル とは同種の元からなる密なリストを意味する. (記法上, 列ベクトルよりは行ベクトルが基本になる. 列ベクトルは次に説明する行列の特殊な場合として扱う.)

```
gap> v := [3, 6, 2, 5/2];; IsRowVector(v);
true
```

IsRowVector( ) は与えられたリストが行ベクトルかどうかを判定する関数である. GAP は行ベクトルに対して, 加法, スカラー倍, 内積を計算する.

```
gap> u := [1, 3, -2, 4];;
gap> v + u;
[ 4, 9, 0, 13/2 ]
gap> 2 * v; v * 1/3;
[ 6, 12, 4, 5 ]
[ 1, 2, 2/3, 5/6 ]
gap> v * u;
27
```

次に行列を定義する. GAP では同じ長さを持ったリストを元とする密なリストを行列 という. たとえば, 次は 3 行 3 列の行列を定義する.

```
gap> m := [[1,-1, 1],
>          [2, 0,-1],
>          [1, 1, 1]];
[ [ 1, -1, 1 ], [ 2, 0, -1 ], [ 1, 1, 1 ] ]
gap> m[2][1];
2
```

行列 m の (i, j) 成分は m[i][j] で与えられる. たとえば, 行列 $m = (m_{ij})$ の (2,1) 成分 m_{21} は m[2][1] = 2 となる. 通常のように, 行列は加法, スカラー倍, 乗法を持つ. ただし, 当然ながらサイズがあわなければ演算はできない.

```
gap> n := [[3, 4, 6],
>          [0,-2, 1],
>          [2, 4, 1],
>          [1, 0, 1]];
[ [ 3, 4, 6 ], [ 0, -2, 1 ], [ 2, 4, 1 ], [ 1, 0, 1 ] ]
gap> n * m;
[ [ 17, 3, 5 ], [ -3, 1, 3 ], [ 11, -1, -1 ], [ 2, 0, 2 ] ]
gap> m * n;
Vector *: <right> must have the same length as <left> (3)
```

m * n の計算では, 右と左のサイズが合わないというのでつむじを曲げられ, ブレイクループに逃げ込まれてしまう. 一方行列とベクトルの演算も可能である. しかしこの場合, ちょっと奇妙なことも起こる.

```
gap> [1,0,0] * m;
[ 1, -1, 1 ]
gap> m * [1, 0, 0];
[ 1, 2, 1 ]
```

最初の例は普通の行ベクトルと行列の計算だが, 2 番目の例では普通, この順番で行列と行ベクトルをかけることはできない. 実は GAP は [1，0，0] を列ベクトルとみなして, 行列との積を計算してくれるのである. (有難いことではあるが, 試験でこういう答案を書いたら, 先生は GAP 程やさしくありません).

GAP は整数計算以上に有限体での計算を得意としている. たとえば, 素数 p に対し整数 $\mathbb{Z}$ を $p\mathbb{Z}$ で割った剰余環 $\mathbb{Z}/p\mathbb{Z}$ は体になる. これを有限体, あるいは最初の発見者ガロアにちなんでガロア体, といい, CAP では GF(p) と表す. GF はガロア体 (Galois Filed) の略である. たとえば, GF(5) は整数を 5 で割った余り $\{0,1,2,3,4\}$ に mod 5 で積と和を定義したものである. より一般に GF(p) の有限次拡大 (n 次拡大) は $q = p^n$ 個の元を持つ有限体 GF(q) を定義する. 有限体 $GF(q)$ の乗法群 $GF(q) - \{1\} = GF(q)^*$ は位数 $q-1$ の巡回群になり, その生成元を GAP では Z(q) と表す. (q が素数 p の場合, Z(q) は素数 p の原始根と呼ばれる.)

整数を成分とする行列やベクトルに有限体 GF(q) の元 Z(q)^i をスカラーとしてかけることにより, 行列やベクトルを有限体を成分とする行列やベクトルに変換し, そこで計算を進めるようにすることができる.

```
gap> m * One(GF(5));
[ [ Z(5)^0, Z(5)^2, Z(5)^0 ], [ Z(5), 0*Z(5), Z(5)^2 ],
[ Z(5)^0, Z(5)^0, Z(5)^0 ] ]
```

ここで One(GF(5)) は有限体 GF(5) の単位元 1 を表す. $GF(5) = \{0,1,2,3,4\}$ とすれば, 要するに 1 であるが, これをかけることにより, $\mathbb{Z}$ から $\mathbb{Z}/5\mathbb{Z}$ に移行することになる. 5 の原始根は 2 であるから Z(5) = 2 と思ってよい. この行列を $\{0,1,2,3,4\}$ の中で表現したければ, 関数 Display を使う.

```
gap> Display(m * One(GF(5));
1 4 1
2 . 4
1 1 1
```

これがもとの行列 m の各成分を　mod 5 で取った有限体 GF(5) 上の行列になる (ドット . はゼロを意味する). 直接 Z(2) や Z(4) をかけると,

```
gap> Display(m^2 * Z(2) + m * Z(4));
z = Z(4)
z^1 z^1 z^2
  1   1 z^2
z^1 z^1 z^2
```

親切なことには, Z(4) を z で置き換えて, z の巾として求める行列を書き下してくれるのである.

与えられた行列から小行列を作る操作は, リストから部分リストを作る方法で, 以下のように簡単に行なわれる.

```
gap> sm := m{ [1, 2]}{[2, 3]};
[ [ -1, 1 ], [ 0, -1 ] ]
gap> sm{ [1, 2]}{[2]} := [[-2], [0]];; sm;
[ [ -1, -2 ], [ 0, 0 ] ]
```

最初の例では, 3 次行列 m から 1 行目, 2 行目と 2 列目, 3 列目を選んで作った 2 次の小行列 sm を求めている. 2 番目の例ではこうして作った 2 次行列 sm の 2 列目を列ベクトル [[-2], [0]] で置き換えた行列を sm として書き出している.

群論に関係した関数として, GAP は与えられた群の元の位数を計算する関数 Order( ) を持っている. 有限体の元を成分とする行列の位数は有限であり次のようになる.

```
gap> Order(m * One(GF(5)));
8
gap> Order(m);
infinity
```

最初の例は, 行列 A = m * One(GF(5)) の位数が 8, つまり, $A^m = I$ となる最小の数が 8 であることを示している. 次の例では整数行列 m の位数が無限大であることを示している.

以下に行列に関する種々の関数をあげておく. 行列 A の転置行列 ${}^tA$ を求めるには, 関数 TransposedMat を使う.

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

2 番目の例では, [[1..5]] は [1, 2, 3, 4, 5] を行ベクトルとする 1 行 5 列の行列を表す. したがってその転置行列は 5 行 1 列の行列になる.

n 次単位行列 I_n は関数 IdentityMat(n) で表される. また, m 行 n 列の零行列は関数 NullMat(m,n) で与えられる.

```
gap> IdentityMat(3);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> NullMat(2,3);
[ [ 0, 0, 0 ], [ 0, 0, 0 ] ]
```

正方行列 $A = (a_{ij})$ に対し, $\mathrm{Tr}\ A = \sum_i a_{ii}$ を A のトレース (trace) という. $\mathrm{Tr}\ A$ は関数 TraceMat で計算できる.

```
gap> TraceMat([[1,2,3], [4,5,6], [7,8,9]]);
15
gap> I:= IdentityMat(3);;
gap> TraceMat(I);
3
```

正方行列 A の行列式 $\det A$ および 行列 A の階数 $r(A)$ はそれぞれ関数 DeterminantMat, RankMat で計算できる.

```
gap> a:= [[1,2,-1], [2, 5, 0], [3, 3, 1]];
[ [ 1, 2, -1 ], [ 2, 5, 0 ], [ 3, 3, 1 ] ]
gap> DeterminantMat(a);
10
gap> b:= [[4,1,2], [3,-1,4], [-1,-2,2]];;
gap> RankMat(b);
2
```

しかし, 行列式や階数の計算は基本的に行列の基本変形に基づいているので, 大きなサイズの行列の計算には時間がかかる.

正則行列 A に対し, A の逆行列 A^{-1} は次のように書けばよい.

```
gap> a^-1;
[ [ 1/2, -1/2, 1/2 ], [ -1/5, 2/5, -1/5 ], [ -9/10, 3/10, 1/10 ] ]
```

2.8 簡単なレコード

GAP では幾つかのデータを複合的に記録させておく手段として, レコード (記録) が用意されている. リストと同じように, レコードも部分オブジェクト (レコード成分という) からできている. しかしリストの部分オブジェクトが primes[1], primes[2] のように番号によって与えられるのと異なり, レコードでは, 各成分が直接名前で参照される. 例えば日付けに関するデータを保存しておくには

```
gap> date := rec(year := 2003,
>                month := "June",
>                day := 14);
rec( year := 2003, month := "June", day := 14 )
```

とする. これで, year, month, day を成分とするレコードが定義された. レコードの各成分を参照するには,

```
gap> date.year; date.month; date.day;
2003
"June"
14
```

とすればよい. レコードの中に, さらに詳しいデータをレコードとして付け加えることもできる.

```
gap> date.time := rec(hour:= 19, minute := 23, second := 12);
rec( hour := 19, minute := 23, second := 12 )
gap> date;
rec( year := 2003, month := "June", day := 14,
     time := rec( hour := 19, minute := 23, second := 12 ) )
```

このように, どんどん新しいデータを付け加えてレコードを完成させていくわけだが、時としてレコードの中にどんな項目があったか忘れてしまう. そんな場合は, 関数 RecNames によりレコードの各成分を参照できる.

```
gap> RecNames(date);
[ "year", "month", "day", "time" ]
```

例えば誕生日のデータを集めて保存しておくような場合に, レコードは有効である.

3 章　さらに関数について

3.1 関数の書き方

画面に hello, world. と文字を書く関数は次のように定義される.

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

GAP の関数はアラジンの魔法のランプのようなもので, 呼ばれると大男が出てきて命令を実行し, 終るとランプに戻っていく. 多くの場合関数の値を返すが, 上の例では Print 命令を実行するのみである. すなわち 2 行目で hello world と画面に書き, 次いで改行 "\n" する.

関数は次の書式で書かれる.

```
function ( 変数 ) 命令 end
```

まず, function で関数を宣言し, その後の ( ) の中に関数の変数 (パラメータ) を, コンマで区切って書く. 変数の数は上のように 0 個でもいいし, 何個でも許される. function( ) の後ろにはセミコロンを書かない. プログラムは最後に end; と書くことにより終了する. 定義した関数 sayhello の中身を見るには次のようにする.

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

sayhello の後ろに改行指令 "\n" をつけないと, endgap> のように, 最後の end と次の GAP のプロンプト gap> がつながってしまう.

新しく定義された関数 sayhello を実行するには sayhello(); と書けばよい.

```
gap> sayhello();
hello world.
```

3.2 If 構文

次の例は整数の正負を判定する符合関数を定義するプログラムである.

```
gap> sign := function (n)
>           if n < 0 then
>              return -1;
>           elif n = 0 then
>              return 0;
>           else
>              return 1;
>           fi;
>     end;
function( n ) ... end
gap> sign(0); sign(-99); sign(11);
0
-1
1
```

符合関数 sign(n) は整数 n が 正, 負 または 0 にしたがって値 $1, -1, 0$ を返す. ここで使われているのが if 命令である. if 命令は次の書式で表され, 条件によって別れ道を指定する.

```
if  条件  then   命令  elif  命令  then  命令  else  命令  fi
```

elif はもちろん else if をつなげた造語で, 間にいくつ入れてもいいが最後は, else できちっと締める. 必要がなければ, 2.5 節の例のように elif も else も省略できる.

山のガイドブックの中に, もし山の中で熊に会ったらどうするかという話で, 熊を刺激しないように笑顔で話しかける, もしそれで駄目だったらそうーっと離れる, もしそれで駄目だったら必死になって逃げる, もしそれでも駄目だったら木に登る, もしそれでも駄目だったら川に飛び込む, もしそれでも駄目だったら死んだふりをする, ..., というのがあった. もし駄目だったらと何回も云っているところを見ると著者自身もこれで助かるとは思っていないようである. そもそも死んだふりをして助からなかった人の話は聞けないから最後の方法がどの程度有効かも分からないが, とにかくこれが典型的な if 構文である.

3.1 節で, 関数はランプの精のようなものだと云ったが, 関数を定義する過程は, 我々自身がアラビアンナイトの世界に飛び込むことを意味する. function(a，b，c) と関数が宣言された時点で, 我々はこちらの世界から変数 a，b，c を携えて別世界に行く. end 命令でプログラムが終了しもとの世界に戻れるが, プログラムの途中で宝物を抱えて逃げて来ることもできる. それが return 命令で, 上の例では, それぞれ -1，0，1 と共に現実世界に戻って来るのである.

2.1 節で再帰リストについて述べたが, CAP では関数を定義する文章の中でその関数自身を使うことができる. これは, にわとりが先か卵が先かという問題とも密接につながっているようだが, 漸化式で定義された数列を扱うときなどはすこぶる都合がよい. 以下の例はフィボナッチ数列を計算するプログラムである.

```
gap> fib:= function(n)
>       if n in [1,2] then
>             return 1;
>       else
>             return fib(n-1) + fib(n-2);
>       fi;
>    end;
function( n ) ... end
gap> fib(15); fib(20); fib(30);
610
6765
832040
```

ここでは, 初期条件 `fib(1) = 1`, `fib(2) = 1` とフィボナッチ数列の漸化式 (5 行目) を与えるだけで, `fib(n)` をすべて計算してくれる. しかしプログラムは短くて書き易いけれど, 再帰命令の入ったプログラムのループの回数は増加する. そこでプログラムの効率, つまり計算時間は長くなってしまう. 効率の良い計算をするためには, たとえプログラムは長くなっても再帰命令ではなく, 繰り返し命令を使ったプログラムにした方がよい (次節参照) .

3.3 局所変数

次の例は 2 つの整数 `a`, `b` の最大公約数 `gcd(a,b)` をユークリッドの互除法を用いて計算するプログラムである.

```
gap> gcd:= function(a,b)
>            local c;
>            while b <> 0 do
>               c:= b;
>               b:= a mod b;
>               a:= c;
>            od;
>           return c;
>    end;
function( a, b ) ... end
gap> gcd (30,63);
3
```

`a` と `b` の最大公約数を求めるには, 組 `(a,b)` を `a' = b` と `a` を `b` で割った余り `b'` の組 `(a', b')` で置き換えるという操作を繰り返し, `b'` がゼロになったときの `b` が最大公約数を与える. これを実際に行なおうとすると, 計算の途中で `b` の値を退避させておくために, どうしてももう一つ変数 `c` が必要になる. この変数は, 関数の定義が書かれている別世界, つまりアラビアンナイトの世界でのみ意味を持つ変数である. これをプログラムの始めで `local c;` として宣言し, 局所変数 という. 局所変数は, あくまで `gcd(a,b)` の世界でのみ意味を持つので, 外の世界で同じ変数 `c` を使っても影響は及ばない. これは大事なことで, `function` の定義は隠されているか、自分で定義したとしても中身は覚えていないのが

常であるので, 局所変数にしておけばこちらの世界で安心して好きな変数を使えるのである. 次の例で変数 c の値が変化しないことを見て欲しい.

```
gap> c:= 7;;
gap> gcd(30,63);
3
gap> c;
7
```

局所変数を使って, フィボナッチ数列の以前より効率の良い (しかし以前より複雑な) プログラムを作ることができる.

```
gap> fib := function (n)
>             local f1, f2, f3, i;
>             f1 := 1;  f2 := 1;
>             for  i  in [3..n] do
>                  f3 := f1 + f2;
>                  f1 := f2;
>                  f2 := f3;
>             od;
>             return f2;
>           end;
function( n ) ... end
gap> List([1..10], fib);
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

4 章　ファイルを使いこなす

GAP で関数やプログラムを定義しても, 一度 GAP を終了すればこれらの関数やプログラムは消去されてしまい, 次に使うときはまた定義しなおさなければならない. また計算されたデータも終了と同時に消されてしまう. 一度定義した関数やプログラムを GAP に忘れずに記憶させ, 大量のデータを GAP に読み込ませるためにはファイルをうまく使わなければならない. 以下ファイルの使い方を説明する.

4.1 Read

例えば, 2.5 節で出てきた素数を計算するプログラムを保存しておきたいとする. その場合, 適当なエディターを使って, 次のようにファイル myfile.g を作る. (拡張子は別に.g でなくても良い).

```
###### myfile.g ######
primes := [];
numbers := [2 .. 5000];
for p in numbers do   ### prime numbers less than 5000
```

```
    Add(primes, p);
    for n in numbers do
      if n mod p = 0 then
      Unbind(numbers[n-1]);
      fi;
    od;
  od;
```

ファイルの中では, 命令は実行されないので, 以前のように primes := [];; と 2 重のセミコロンにする必要はない. また GAP の中では # はコメント文の働きをする. 各行で # 以下は無視されるので # を使ってプログラムにコメントをつけるのが普通である.

```
gap> Read("myfile.g");
```

とすることにより, myfile.g の内容が GAP に読み込まれる. 特に, リスト primes には 5000 以下の素数の表が蓄えられる. プログラムを編集する場合, 例えば 5,000 を 10,000 にしたかったらファイル myfile.g を編集し直せば良い. これは GAP の中で編集作業をするよりは, はるかに簡単である.

なお, 自分用のファイル myfile.g を GAP の立ち上げ時に自動的に読み込ませることもできる. ホームディレクトリにファイル .gaprc を作り, ここにファイルの読み込みや, 良く使う定義を以下のように書いておけばよい. GAP は起動時に最初に.gaprc を読みにいくので, ここで必要なデータがすべて GAP に読み込まれる. 例えば

```
Read("myfile.g");
a:= [[1,2,-1], [2, 5, 0], [3, 3, 1]];
b:= [[4,1,2], [3,-1,4], [-1,-2,2]];
```

4.2 PrintTo

4.1 節で計算した 5000 以下の素数のリストをデータとして保存しておくには, 関数 PrintTo を使う. 1.6 節に出てきた関数 Print との違いは, 画面に打ち出す代わりにファイルに書き出すということである. 関数 PrintTo は

PrintTo(filename, obj1, obj2 ...)

の書式を持つ. obj1, obj2, ... はファイルに書き出すオブジェクトを表す.

```
gap> PrintTo("primelist.g", primes);
```

これにより, リスト primes の内容が, ファイル primelist.g にデータとして保存される. 次に述べる関数 AppendTo との違いは, PrintTo では実行されるたびにデータが上書きされ, 以前のデータが消去されるのに対し, AppendTo では, 以前のデータの後ろに新しいデータが付け加えられていくことである.

4.3 AppendTo

AppendTo は指定したファイルにデータを順次付け加えて保存していく関数を表す. 以下の書式を持つ.

AppendTo(filename, obj1, obj2 ...)

5 章 群と準同型

5.1 置換群

対称群の部分群を置換群という. GAP では非常に簡単に置換群を扱うことができる.

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

これにより, GAP は 互換 (1,2) と巡回置換 (1,2,3,4,5,6,7,8) によって生成された置換群 ($S_n, n \gg 1$ の部分群) を s8 として定義する. (1,2) と (1,2,3,4,5,6,7,8) は 8 次対称群 S_8 を生成するので, S_8 が GAP で定義できたことになる. 以後, 対称群 S_8 を扱うには, 単に s8 を呼び出せばよい.

GAP では群 s8 は集合として定義される. そこで群 s8 の位数を知るには, 集合の元の個数を与える関数 Size を利用する. 以下のように $|S_8| = 8! = 40320$ が得られる.

```
gap> Size(s8);
40320
```

交代群 A_8 は S_8 の偶置換の全体として定義されるが, S_8 の導来群 (derived subgroup) になっている. (注. G を群とするとき, $xyx^{-1}y^{-1}(x, y \in G)$ で生成された G の部分群を G の導来群という.) GAP には与えられた群の導来群を計算する関数 DerivedSubgroup が用意されている. そこで以下のように簡単に交代群 a8 が定義できる.

```
gap> a8 := DerivedSubgroup(s8);
Group([ (1,2,3), (2,3,4), (2,4)(3,5), (2,6,4), (2,4)(5,7),
          (2,8,6,4)(3,5) ])
gap> Size(a8); IsAbelian(a8);
20160
false
```

交代群 A_8 が置換 $(1,2,3), (2,3,4), (2,4)(3,5), \cdots$ で生成された群として定義され, a8 と名前が付けられたのである. ここで 2 行目の表示は, あくまで生成元を表わしているのであって, a8 の元を全部書いているのではないことに注意する. A_8 の位数は 20160 であるから元を全部書いたら大変なことになる. 生成元による表示が如何に効率が良いか分かるだろう. 生成元を与えることにより群は完全に決定されるから, 集合 a8 を表示するにはこれで充分である. 表舞台には現れないが, コンピュータには全ての元がストックされており, 必要なときに必要な元を使ってコンピュータが計算を実行してくれるのである. 我々の目に見えないところで大変な努力がなされているのですね.

とは言え, せっかく新しい群ができたのだから, 生成元などとけちくさいことを言わず, 全ての元を書き出して見たいという誘惑にかられるのも人情である. そんなときは関数 Elemnts( ) を使うことができ

る. しかし, ゆめゆめ関数 Elements を s8 や a8 に対して試みようとしてはいけない. 悲惨な結果になること請合いである.

```
gap> s3:= Group((1,2), (1,2,3));
Group([ (1,2), (1,2,3) ])
gap> Elements(s3);
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

GAP は群 s3 を集合として扱うから, s3 の元は順番に従って並んでいる. 上の場合, 順番は数字 1, 2, 3 に関する辞書式順序である.

なお GAP には, 与えられた群がアーベル群かどうかを判定する関数 IsAbelian が用意されている. 交代群 A_8 は非可換な群であるから, 「残念ながら違います」 (false) という答が返って来る.

今までやった s8 や a8 に関する種々の計算は, 全てその群のデータとして保存され, 後に呼びだすことができる. 例えば, a8 には a8 が位数 20160 の群であり, アーベル群ではないというデータが保存され, s8 には, その導来群が a8 になるというデータが保存される.

G を有限群とする. 素数 p に対して, G の位数を割る最大の p 巾を p^m をとする. G には位数 p^m の部分群が存在し, それらはすべて G で共役になる. このような部分群を G の (p-) シロー部分群 (Sylow subgroup) という. GAP にはシロー部分群を計算する関数が用意されている. 交代群 A_8 の 2-シロー群 ($p = 2$ の場合) は

```
gap> syl2 := SylowSubgroup(a8,2);
Group([ (1,8)(5,7), (2,6)(5,7), (3,4)(5,7), (3,5)(4,7),
  (1,2)(6,8),  (1,3)(2,5)(4,8)(6,7) ])
gap> Size(syl2);
64
```

syl2 は A_8 の一つの 2 シロー群を表す. それでは, a8 には何個の 2 シロー群が存在するだろうか. $G = A_8, P =$ syl2 とおく. G は 2 シロー群全体の集合 X に共役により作用する. 2 シロー群はすべて G で共役であるから, X は一つの G 軌道をなす. ここで一般的な定理から P を含む G 軌道の個数は $|G|/|N_G(P)|$ に一致する. ただし,

$$N_G(P) = \{g \in G \mid gPg^{-1} = P\}$$

は共役の作用に関する $P \in X$ の固定化群である. $N_G(P)$ を P の G における正規化群 (normalizer) という. 以上の議論から, G の 2 シロー群の個数は,X の個数, すなわち $|G|/|N_G(P)|$ で与えられることが分る.

GAP には H を G の部分群とするとき, $N_G(H)$ を計算する関数 Normalizer(G,H) が用意されている. そこで

```
gap> norm := Normalizer(a8, syl2);
Group([ (1,8)(3,4), (2,6)(3,4), (3,4)(5,7), (3,7)(4,5),
   (1,6)(2,8), (1,7)(2,4)(3,6)(5,8) ])
gap> Size(norm);
64
```

これより A_8 の 2 シロー群の個数は, $|G|/|N_G(P)| = 20160/64 = 315$ となる.

ところで正規化群 $N_G(P)$ は P を含む G の部分群である. 上の計算から $|N_G(P)| = |P| = 64$ なので, $N_G(P) = P$ となることが分る. すなわち, `norm` = `syl2` が成り立つ. しかし, 直接 GAP に聞くこともできる.

```
gap> norm = syl2;
true
```

= は定義式ではなく等式なので, このように書くと等式が正しいか間違いかを尋ねることになる. 答えは「正しい」. (注. 群は同じでも, 生成元の表示は異っていることに注意する. 生成元の表示は何通りもあり得るので, そこだけ見ても判定できない.)

正規化群と対になる概念として次がある. H を G の部分群とするとき

$$C_G(H) = \{g \in G \mid ghg^{-1} = h \text{ for any } h \in H\}$$

を H の G のおける中心化群 (centralizer) という. $C_G(H)$ は $N_G(H)$ の 正規部分群になる. GAP には中心化群を計算する関数 `cent(G,H)` がある. また G の中心 (centre)

$$Z(G) = \{g \in G \mid gxg^{-1} = x \text{ for any } x \in G\}$$

を計算する関数 `Centre(G)` もある.

```
gap> Centre(syl2);
Group([ ( 1, 8)( 2, 6)( 3, 4)( 5, 7) ])
gap> Size(last);
2
gap> cent := Centralizer(a8, Centre(syl2));
Group([ (3,4)(5,7), (3,5)(4,7), (2,3)(4,6), (1,2)(6,8) ])
gap> Size(cent);
192
```

G を有限群とするとき, G の導来群を $[G, G]$ と表す. G の部分群の列

$$G = G_0 \supset G_1 \supset \cdots \supset G_n \supset \cdots$$

を $G_n = [G_{n-1}, G_{n-1}]$ により定義し, G の導来列 (derived series) という. 有限群の重要なクラスとして可解群があるが, 導来列は可解群になるための判定条件を与える. G の 導来列が有限回の後, 単位群 $\{1\}$ に到達すること, つまりある $n \gg 1$ に対して $G_n = \{1\}$ となることが, G が可解になるための必要十分条件である.

GAP は与えられた群の導来列を計算する関数 DerivedSeries( ) を持っている. 例えば,

```
gap> DerivedSeries(cent);
[ Group([ (3,5)(4,7), (2,5,3)(4,6,7), (1,3)(2,5)(4,8)(6,7),
          (1,2)(3,5)(4,7)(6,8), (3,4)(5,7), (2,6)(3,4),
          (1,8)(2,6)(3,4)(5,7) ]),
  Group([ (2,5,3)(4,6,7), (1,3)(2,5)(4,8)(6,7),
          (1,2)(3,5)(4,7)(6,8),
          (3,4)(5,7), (2,6)(3,4), (1,8)(2,6)(3,4)(5,7) ]),
  Group([ (1,3)(2,5)(4,8)(6,7), (1,2)(3,5)(4,7)(6,8),
          (3,4)(5,7), (2,6)(3,4), (1,8)(2,6)(3,4)(5,7) ]),
  Group([ (1,8)(2,6)(3,4)(5,7) ]), Group(()) ]
gap> List(last, Size);
[ 192, 96, 32, 2, 1 ]
```

前半で群 cent の導来列を計算し, 現われる部分群をすべて決定している. しかし, これではごちゃごちゃして見づらいので、後半は, 導来列に現れる群の位数を並べて、リストを作ってある. これは $G =$cent とするとき,

$$G = G_0 \supset G_1 \supset G_2 \supset G_3 \supset G_4 = \{1\}$$

であって,

$$|G_0| = 192, \quad |G_1| = 96, \quad |G_2| = 32, \quad |G_3| = 2, |G_4| = 1$$

を示している. したがって, $G =$cent は可解群である. それでは a8 の導来列はどうなるだろうか.

```
gap> DerivedSeries(a8);; List(last, Size);
[ 20160 ]
```

これは, $G = A_8$ の導来群がまた A_8 に一致し, したがって導来列は

$$G = G_0 = G_1 = G_2 = \cdots$$

となることを意味している. これではいつまでたっても $\{1\}$ には到達できないので, $G = A_8$ は可解群ではない. 実は $n \geq 5$ のとき, A_n は可解群にはならないことが知られている. それが, 5 次以上の代数方程式に根の公式が存在しない (巾根と 4 則演算によって解けない) というアーベル, ガロアの理論の根拠になっている.

素数 p に対して $(\mathbb{Z}/p\mathbb{Z})^n$ に同型な群を基本アーベル (p) 群 (elementary abelian group) という. $(\mathbb{Z}/p\mathbb{Z})^n$ は有限体 $\mathbb{Z}/p\mathbb{Z}$ 上の n 次元ベクトル空間の加法群に他ならない. ここで交代群 a8 の中に次のように基本アーベル 2 群を定義する. GAP は与えれた群が基本アーベル群かどうかを判定する関数 IsElementaruAbelian( ) を持っている.

```
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
```

elab は $(\mathbb{Z}/2\mathbb{Z})^3$ に同型な基本アーベル群である. ところで GAP では計算の結果現れる群は, すべて生成元で書かれる. これは, かなり見づらいことである. もし a8 や elab のように, あらかじめ定義された群が出てくれば, その名前で書いてもらった方が分かり易い. しかし GAP は出て来た群を elab と認識してくれない. それは, ある群を G と書いたり H と書いたりするように, elab は単に与えられた群を指し示す一時的な名前 (identifier) に過ぎないからである. 群そのものに名前をつけるには関数 SetName を使う.

```
gap> SetName(elab, "2^3"); elab;
2^3
```

以後, 計算の過程で elab が現れれば, GAP はそれを生成元による表示ではなく, 2^3 と書いてくれる. しかし, GAP にこちらの命令を伝える場合には, 2^3 ではなく identifier elab を使わなければならない. いくつもの群に同じ名前 2^3 を付けることもあるので, 名前だけでは同姓同名の群を区別できないからである. こちらからの指示は identifier (elab), GAP からの返答は名前 (2^3), と使い分ける必要がある.

$G =$a8, $H =$elab として正規化群 $N_G(H)$ を計算する.

```
gap> norm := Normalizer(a8, elab);
Group([ (1,5)(2,6)(3,7)(4,8), (1,3)(2,4)(5,7)(6,8),
        (1,2)(3,4)(5,6)(7,8),
        (5,6)(7,8), (5,7)(6,8), (3,4)(7,8),
        (3,5)(4,6), (2,3)(6,7) ])
gap> Size(norm);
1344
```

これで, elab の正規化群として位数 1344 の群 norm が確定した. 一般に H を G の部分群とすると, H の正規化群 $N = N_G(H)$ は H を正規部分群として含み商群 (剰余群) N/H が定義される. 写像 $\varphi : N \to N/H, g \mapsto gH$ は全射準同型を与える. φ を, H による自然な準同型 (natural homomorphism) という. GAP は次のようにこの準同型写像を定義する.

```
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),
          (1,4)(5,7) ])
gap> Size(f);
168
```

一行目が準同形 hom を定義する関数である. 何の省略もなく, 公明正大, 実に分り易い命令であるが, これだけ長いと綴りを間違いなく書くのに苦労する. By の B は大文字です.

一旦準同型が定まると, その像 (image) として商群が定義される. 商群の生成元は置換として表されているが、あくまで剰余類の代表元としての意味である. 準同形の核 (kernel) も定義される.

```
gap> ker := Kernel(hom);
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) ])
```

写像 φ による元 $x \in N$ の像 $\varphi(x)$ および, $y \in N/H$ の逆像 $\varphi^{-1}(y)$ は, それぞれ次のように表される.

```
gap> x := (1,8,3,5,7,6,2);; Image(hom, x);
(1,5,6,3,7,2,4)
gap> coset := PreImages(hom, last);
RightCoset(2^3,(2,8,6,7,3,4,5))
```

Image(hom,x) が像 $\varphi(x)$ を与える. $y = \varphi(x) \in N/H$ に対して PreImages(hom, y) が逆像 $\varphi^{-1}(y)$ を与える. $\varphi : N \to N/H$ であるから, $\varphi(x_1) = y$ となる $x_1 = (2,8,6,7,3,4,5) \in N$ により $\varphi^{-1}(y) = x_1 H$ (H による右剰余類) と表される. それが最後の式である. ここで注意すべきは, GAP は代表元を GAP 自身の規則によって選ぶので, x_1 が必ずしも最初に取った x と一致しないということである. しかし, もちろん x と x_1 は H の同じ剰余類に含まれるはずで、それは次のように確かめられる.

```
gap> rep := Representative( coset );
(2, 8, 6, 7, 3, 4, 5)
gap> x * rep^-1 in ker
true
```

3 行目の命令が 「$x x_1^{-1} \in H$ が成立するや否や」と GAP に問うているわけで, それに対して, GAP が「然り」と答えているのである.

剰余群 f$= N/H$ は単純群になる. ここで単純群とは自分自身と単位群以外に正規部分群を持たない群のことで, これ以上は分解できない素粒子のようなものである. 有限群の構造を調べる問題は多くの場合単純群に帰着する. GAP は与えられた群が単純群かどうかを瞬時に判定する.

```
gap> IsSimple(f);
true
gap> IsomorphismTypeFiniteSimpleGroup( f );
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)");
```

当然のことながら, 有限単純群には皆名前がついている.
IsomorphismTypeFiniteSimpleGroup(f) は有限単純群 f についている名前をリストアップする関数である. f は長年にわたって人々に愛され, 親しまれてきた名高い群であり, それ故に多くの名前を持っている. ここでは f = L_3(2) ($\mathbb{F}_2$ 上の 3 次行列で, 行列式が 1 のもの全体のなす群; $SL_3(\mathbb{F}_2)$) を使うことにする.

$H =$elab の正規化群 $N =$norm は定義により, H に共役によって作用する. H は 8 個の元からなる集合であり, したがって N はこの 8 個の元の置換を引き起こし, (しかし, N の元はすべて H の単位元を固定するので, 実際には 7 個の元の置換になる) 準同形 $\phi : N \to S_8$ が導かれる. GAP では, H を順序の

ついた集合として扱うので (Elements(s3) の項を参照), H の元の置換の全体は自然に 8 次対称群 S_8 と同一視できることに注意する . ここで H がアーベル群であることから N の作用の H への制限は, 8 個の元を全て固定する. したがって $H \subset \mathrm{Ker}\,\phi$ であり, (準同形定理により) 準同形 $\phi_1 : N/H \to S_8$ が得られる. $N/H =$ L_3(2) は単純群なので, ϕ_1 は単射である.

写像 ϕ_1 の像 $K = \mathrm{Im}\,\phi_1$ は関数 Action により次のように求められる.

```
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);
true
gap> IsSubgroup(norm, op);
true
```

$K =$ op が S_8 の部分群として得られた. 所で, もともと norm は $S_8 =$ s8 の部分群であったことを思い出そう. そこで得られた群 K が a8, norm の部分群になっているかどうか調べる. もちろん norm に含まれることが分れば a8 をやる必要はないが, 関数 IsSubgroup(, op) を使って一歩一歩敵を追い詰めていくところに, カー・チェイスならぬグループ・チェイスの醍醐味がある.

かくして $N =$ norm の部分群 $K =$ op が確定した. ここで $H \cap K = \{1\}$ を確かめる.

```
gap> IsTrivial( Intersection (elab, op));
true
gap> SetName(norm, "2^3:L_3(2)");
```

関数 Intersection(elab, op) が共通部分として得られる部分群 $H \cap K$ を定義する. 次いで関数 IsTrivial ( ) により, 与えられた群が単位群 $\{1\}$ かどうかを判定する.

以上の計算をまとめると

$$N/H \simeq K, \qquad K \subset N, \qquad H \cap K = \{1\}.$$

これより, N が H と K の半直積 $H \rtimes K$ に一致することが導かれる. (注意. 群 N が H と K の半直積 $H \rtimes K$ であるとは $N = HK$, $H \lhd N$, $H \cap K = \{1\}$ が成り立つことをいう. この場合, 集合として $N \simeq H \times K$ であり, N の元 g は $g = hk, h \in H, k \in K$ と一意的に表される.)

5.2 群の共役類

群 G の元 g, g' は $h^{-1}gh = g'$ となる $h \in G$ が存在するとき, 共役であるといい $g \sim g'$ と表す. $g \sim g'$ は同値関係であり, その同値類を共役類 (conjugacy class) という. $x^{-1}gx$ を g^x とも表す. そこで $g \in G$ を含む共役類は $g^G = \{x^{-1}gx \mid x \in G\}$ に他ならない. G は共通部分のない共役類の和集合 (disjoint union) に分解される.

$$G = \coprod_{g \in G/\sim} g^G$$

ただし, $G/\sim$ は G の共役類の (代表元の) 集合を表す. 特に,

$$|G| = \sum_{g \in G/\sim} |g^G| \tag{5.2.1}$$

が成り立つ. 各 $g \in G$ に対し, $Z_G(g) = \{x \in G \mid x^{-1}gx = g\}$ を g の中心化群 (centralizer) という. 各 $g \in G$ に対し

$$|G| = |g^G||Z_G(g)|$$

が成立する. (5.2.1) と合せて

$$1 = \sum_{g \in G/\sim} \frac{1}{|Z_G(g)|} \tag{5.2.2}$$

が得られる. (5.2.1) または (5.2.2) を有限群 G の類等式という.

さて交代群 $G = A_8$ の共役類のリスト `ccl` は次のように与えられる.

```
gap> ccl := ConjugacyClasses( a8 );
[ ()^G, (1,2)(3,4)^G, (1,2)(3,4)(5,6)(7,8)^G, (1,2,3)^G,
  (1,2,3)(4,5)(6,7)^G, (1,2,3)(4,5,6)^G, (1,2,3,4)(5,6)^G,
  (1,2,3,4)(5,6,7,8)^G, (1,2,3,4,5)^G, (1,2,3,4,5)(6,7,8)^G,
  (1,2,3,4,5)(6,8,7)^G, (1,2,3,4,5,6)(7,8)^G,
  (1,2,3,4,5,6,7)^G, (1,2,3,4,5,6,8)^G ]
gap> Length (ccl);
14
```

G は 14 個の共役類に分解される. GAP は各共役類を代表元をひとつ決めて, 例えば, $g^G = (1,2)(3,4)^G =$ `(1,2)(3,4)^G` のように表示するが, 共役類の代表元だけを取り出したかったら, 関数 `Representative( )` を使って,

```
gap> rep := List (ccl, c -> Representative (c));
[ (), (1,2)(3,4), (1,2)(3,4)(5,6)(7,8), (1,2,3),
  (1,2,3)(4,5)(6,7), (1,2,3)(4,5,6), (1,2,3,4)(5,6),
  (1,2,3,4)(5,6,7,8), (1,2,3,4,5), (1,2,3,4,5)(6,7,8),
  (1,2,3,4,5)(6,8,7), (1,2,3,4,5,6)(7,8),
  (1,2,3,4,5,6,7), (1,2,3,4,5,6,8) ]
```

とする. 1 行目は, リスト `ccl` の各要素 `c` にその代表元 `Representative(c)` を対応させて新たにリストを作れという命令である. 各共役類が何個の元からなるかを知るには,

```
gap> List(ccl, Size);
[ 1, 210, 105, 112, 1680, 1120, 2520, 1260, 1344, 1344,
  1344, 3360, 2880, 2880 ]
```

`List(ccl, Size)` はリスト `ccl` の各要素 (これもリスト) に, そのリストとしてのサイズを対応させてリストを作る命令である.

```
gap> Sum(List(ccl,Size)) = Size(a8);
true
```

によって (5.2.1) が確かめられる. 上のリストで例えば 3 番目の共役類は, 105 個の元からなる集合である. その元を全て表示したければ,

```
gap> Elements(ccl[3]);
```

とすればよい. ここには結果を書かないが, 興味のある人は自分でやってみて下さい.

GAP では例えば, 元 $g = (1,2)(3,4)$ の中心化群 $Z_G(g)$ は

```
gap> Centralizer(a8, (1,2)(3,4));
Group([ (6,7,8), (5,6)(7,8), (3,4)(7,8), (1,2)(7,8), (1,3)(2,4) ])
gap> Size(Centralizer(a8, (1,2)(3,4)));
96
gap> 96 * 210 = Size(a8);
true
```

によって計算できる. $|g^G||Z_G(g)| = |G|$ も確かめられる. 各代表元 $g \in G/\sim$ に対する $1/|Z_G(g)|$ のリストは

```
gap> inv:= List(rep, c -> 1/Size(Centralizer(a8,c)));
[ 1/20160, 1/96, 1/192, 1/180, 1/12, 1/18, 1/8, 1/16,
  1/15, 1/15, 1/15, 1/6, 1/7, 1/7 ]
```

によって得られる. そこで (5.2.2) 式が確かめられる.

```
gap> Sum(inv);
1
```

具体的にこの式を書き下してみると,

$$1 = \frac{1}{20160} + \frac{1}{96} + \frac{1}{192} + \frac{1}{180} + \frac{1}{12} + \frac{1}{18} + \frac{1}{8} + \frac{1}{16} + \frac{1}{15} + \frac{1}{15} + \frac{1}{15} + \frac{1}{6} + \frac{1}{7} + \frac{1}{7}$$

最初の項の分母の大きさから考えて, この式はかなりショッキングな関係式である. 試みに $1/20160$ に分数 $1/N$ (N は 20160 の約数) をいくつか少ない個数を加えて総和を 1 にすることを考えてみればよい. そう簡単な話ではないことが理解されるだろう. 類等式は簡単な等式ながら有限群の持つ隠された対称性を如実に表現する関係式なのである.

群 G の元 g に対して, $g^n = 1$ となる最小の正の整数 n を g の位数 (order) という. g の位数は g で生成される G の部分群の位数に一致する. GAP では, $g = (1,2,3)(2,3,4,5)$ に対して

```
gap> Order((1,2,3)*(2,3,4,5));
6
```

と計算できる. ひとつの共役類に含まれる元の位数は一定である. したがって各共役類の代表元の位数が分ればすべての G の元の位数が分る. 各共役類に含まれる元の位数のリストは次のようにして得られる.

```
gap> List(ccl, c -> Order(Representative(c)));
[ 1, 2, 2, 3, 6, 3, 4, 4, 5, 15, 15, 6, 7, 7 ]
```

10 番目, 12 番目の共役類の代表元は,

```
gap> Representative(ccl[10]); Representative(ccl[12]);
(1,2,3,4,5)(6,7,8)
(1,2,3,4,5,6)(7,8)
```

であるから, 確かに位数はそれぞれ, 15, 6 になっている.

以上の議論で, Order, Size, Length が使い分られていることに注意しよう. GAP では, Order は元の位数を意味する. 一般に集合に含まれる元の個数は Size により与えられる. 群 G の元の個数は数学用語では位数というが, GAP では集合の場合と同様に Size で計算することに注意する. Length はリストの長さを表す. 例えば共役類の個数は共役類のリストの長さとして与えられる.

5.3 ブロックと置換表現

今 class を 112 個の元からなる A_8 の共役類とする. それは, 共役類のリスト ccl の 4 番目にあるから,

```
gap> class := ccl[4];
(1,2,3)^G
gap> Size(class);
112
```

$X =$ class は 112 個の点を持つ集合であり, $G = A_8$ が X に推移的に作用する. G は 112 個の点の置換を引き起こすので準同形 $\varphi : G \to S_{112}$ が定義される. 一般にこのような群 G から対称群 S_n への準同形を G の置換表現といい, n を表現の次数という. そこで, $G = A_8$ の 112 次の置換表現が得られたことになる. 交代群 A_8 は単純群なので, φ は単射準同形である. $G \simeq \mathrm{Im}\,\varphi$ は A_8 の S_{112} への埋め込みを与える. GAP では op $= \mathrm{Im}\,\varphi$ は次のようにして得られる.

```
gap> op := Action(a8, AsList( class ));
<permutation group with 6 generators>
gap> Size(op);
20160
```

S_{112} は巨大な群なので, op の生成元を具体的に書くだけでも大変なことになる. しかし群はちゃんと確定している. ここで, 前のように直接 Action(norm, elab) と書く代わりに AsList(class) によって共役類 class をその元からなるリストに置き換えた後, Action を実行していることに注意する. それは, GAP で共役類を確定する方法は必ずしも全ての元を決めるものではなく, 集合 (リスト) として与えられた方が共役類のままよりも計算が早くなるからである. しかし, 群が大きくなればなるほど, 集

合として全てを確定するのは無理になるというのも極めて当然な話なのであって, そうした意味からも, 集合として表すといった直接的な方法を GAP は避けようと努力しているのである.

有限群 G が集合 X に推移的に作用しているとする. $|X| = n$ とすると, この作用により G の n 次の置換表現が得られる. X の部分集合 Δ がブロックとは, 任意の $g \in G$ に対して

$$\Delta^g = \Delta \quad \text{または} \quad \Delta^g \cap \Delta = \emptyset$$

が成り立つことをいう. X に $1 < |\Delta| < |X|$ となるブロック Δ (自明でないブロック) が存在するとき, G は非原始的な置換群であるといい, そのようなブロックが存在しないとき, 原始的であるという. Δ を G のブロックとすると, Δ^g もまたブロックであり, $\{\Delta^g \mid g \in G\}$ によって X は共通部分のないいくつかのブロックの和集合に分割される (X の分割を与えるブロックの集合をブロック系という).

$$X = \coprod_{g \in \mathrm{St}_G(\Delta)\backslash G} \Delta^g \qquad \text{(disjoint union)} \tag{5.3.1}$$

ただし $\mathrm{St}_G(\Delta) = \{g \in G \mid \Delta^g = \Delta\}$ は Δ の固定化群 (集合 Δ を不変にする元 $g \in G$ の全体) である. そこで各ブロックを点とみなして集合 $Y = \{\Delta^g \mid g \in \mathrm{St}_G(\Delta)\backslash G\}$ を考えると G は Y の上に再び推移的に作用する. G が非原始的ならば $|Y| < |X|$ なので, より次数の低い m 次の置換表現が得られる ($m = |Y|$).

$x \in X$ の固定化群 $\mathrm{St}_G(x)$ を H とおくと, G の X への作用が推移的なことから, X は (G の作用込みで) 右剰余類の集合 $H\backslash G$ と同一視できる. 一方, Δ をブロックとすると $x \in \Delta$ に対して $\mathrm{St}_G(x) \subset \mathrm{St}_G(\Delta)$ が成り立つ. (実際, $g \in \mathrm{St}_G(x)$ ならば, $x \in \Delta \cap \Delta^g$ より $\Delta = \Delta^g$). Y は右剰余類の集合 $\mathrm{St}_G(\Delta)\backslash G$ と同一視される. このことから, G が原始的であることと, 一点の固定化群 $\mathrm{St}_G(x) = \{g \in G \mid x^g = x\}$ が G の極大部分群であることは同値になる. (H が G の極大部分群とは, H を含む G の部分群が G と H 以外に存在しないことをいう.)

a8 の X =class への置換表現 op に戻ろう. GAP は 置換表現が原始的かどうかを判断してくれる. しかしそのためには op の作用域 $\{1, 2, \ldots, 112\}$ をはっきりと指定する必要がある. 例えば群 Group((2,3,4)) は集合 [1..4] $= \{1, 2, 3, 4\}$ にも作用しているし, {2..4] にも作用している. op を集合 [1..113] の上への作用とみれば, 点 113 は a8 の作用の固定点になってしまう. この作用は推移的にならずブロックの議論は適用できない. そこで GAP では次のように書く.

```
gap> IsPrimitive(op, [1 .. 112]);
false
```

後半の [1 .. 112] が作用 op を 112 個の点の集合への置換とみなした上で, 原始的かどうかを問うているのである. 答は「非原始的」ということなので, 集合 class には自明でないブロックが存在する. GAP はそのようなブロックの中で個数が最小のブロックを選び, 集合 class を (5.3.1) のように分解する.

```
gap> blocks := Blocks( op, [1 .. 112]);
 [ [ 1, 7 ], [ 8, 14 ], [ 15, 21 ], [ 10, 26 ], [ 24, 40 ],
   [ 2, 13 ], [ 3, 19 ], [ 49, 54 ], [ 5, 31 ], [ 4, 25 ],
   [ 16, 27 ], [ 22, 28 ], [ 57, 70 ], [ 12, 38 ], [ 36, 42 ],
```

```
[ 43, 48 ], [ 6, 37 ], [ 44, 53 ], [ 9, 20 ], [ 17, 33 ],
[ 73, 77 ], [ 55, 60 ], [ 51, 64 ], [ 47, 68 ], [ 46, 63 ],
[ 11, 32 ], [ 29, 35 ], [ 45, 58 ], [ 18, 39 ], [ 50, 59 ],
[ 80, 90 ], [ 76, 89 ], [ 106, 109 ], [ 101, 104 ], [ 84, 91 ],
[ 67, 72 ], [ 30, 41 ], [ 78, 82 ], [ 75, 85 ], [ 74, 81 ],
[ 93, 96 ], [ 97, 100 ], [ 61, 66 ], [ 95, 102 ], [ 94, 99 ],
[ 83, 87 ], [ 108, 110 ], [ 23, 34 ], [ 52, 69 ], [ 88, 92 ],
[ 111, 112 ], [ 98, 103 ], [ 62, 71 ], [ 56, 65 ],
[ 105, 107 ], [ 79, 86 ] ]
```

blocks は class のブロック系を与えるリストである. リストの各オブジェクトが 2 個の元からなるブロックになっている. 例えば, blocks[1] = [1, 7] がひとつのブロック Δ を与える. 他のブロック [8,14], [15, 21] などは, すべて Δ^g $(g \in G)$ により得られる. GAP に直接聞いてみれば

```
gap> Length(blocks[1]); Length(blocks);
2
56
```

これは, class が 2 個の元からなるブロック 56 個の和に分割されたことを意味する. そこで先に述べたように群 op をブロックの集合 Y (= blocks) に作用させることにより, 新しい置換表現 $\varphi_1 : \mathrm{op} \to S_{56}$ が得られる. $\mathrm{Im}\,\varphi_1$ を op1 とおくと, 交代群 $G = A_8$ の S_{56} への埋め込み a8 $\simeq$ op1 $\subset S_{56}$ が得られる.

```
gap> op1 := Action(op, blocks, OnSets);
<permutation group with 6 generators>
```

ここで注意することは, リスト blocks を構成しているオブジェクト (各ブロック) は集合であるから, それらのブロックを点とみなして群 op を作用させなければならない. OnSets は「集合の集まりの上に群を作用させなさい」という命令である. 新しく得られた置換群 op1 は原始的になる.

```
gap> IsPrimitive( op1, [1 .. 56]);
true
```

そこで先の議論により, op1 における一点の固定化群 M_1 は極大部分群になる. op1 は交代群 A_8 と同型であるからこれにより A_8 の極大部分群 M が得られることになる. しかし今 op1 は S_{56} の部分群として構成しているので, A_8 の極大部分群を具体的に書くためには同型 $\varphi_2 : A_8 \xrightarrow{\sim} \mathrm{op1}$ を決定し, $M = \varphi_2^{-1}(M_1)$ を求めなければならない. $G = A_8, H = \mathrm{op} \subset S_{112}$, $H_1 = \mathrm{op1} \subset S_{56}$ とおく. 構成の仕方を思い出せば,

$$\varphi_2 : G \xrightarrow{\varphi} H \xrightarrow{\varphi_1} H_1$$

が同型 $\varphi_2 = \varphi \circ \varphi_1 : G \simeq H_1$ を与える (右作用のため写像の合成の順番が逆になっていることに注意.)

以上のプロセスは GAP では次のように実行される.

```
gap> ophom := ActionHomomorphism(a8, op);
<action homomorphism>
gap> ophom1 := ActionHomomorphism(op, op1);
<action homomorphism>
```

命令 ActionHomomorphism により a8 の置換表現 φ =ophom, op の置換表現 φ_1 =ophom1 が定義される. $\varphi_2 = \varphi \circ \varphi_1$ であるから

```
gap> composition := ophom*ophom1;;
```

により, φ_2 =composition が定義できた. H_1 の点 $x = 2$ での固定化群 $M_1 = \mathrm{St}_G(x)$ を stab とすると

```
gap> stab := Stabilizer(op2, 2);
<permutation group of size 360 with 5 generators>
```

によって H_1 の位数 360 の極大部分群 M_1 が定まる. $M = \varphi_2^{-1}(M_1)$ であるから

```
gap> preim := PreImages( composition, stab);
Group([ (2,5,7), (1,4)(2,7), (2,6,7), (1,3)(5,7), (6,8,7) ])
```

により, A_8 の極大部分群 M =preim が確定した. A_8 の位数が 20160 であることを考えると, 位数 360 の群 M が極大部分群になるというのは, 驚くべきことである. M は A_8 に比べてはるかに小さい群でありながら, それを含む A_8 の部分群は存在しないのである. これも A_8 が単純群であることの反映であろうか. (蛇足ながら, 単純群の条件は正規部分群が存在しないことであって, 単なる部分群については何も言っていない. しかし単純群においては部分群の存在も非常に制限されることを上の事実は示唆している.)

共役類 class から代表元 c1 を選び, c1 で生成される巡回群を xx とおく. a8 における xx の正規化群として得られる a8 の部分群を K = sgp とする. K も位数 360 の部分群であり, 実は M と共役になる, 即ち $g^{-1}Kg = M$ となる $g \in G$ が存在する.

```
gap> c1 := Representative(class);
(1,2,3)
gap> xx := Group((c1));
Group([ (1,2,3) ])
gap> sgp := Normalizer(a8, xx);
Group([ (1,2,3), (5,6,7), (5,8)(6,7), (4,5,7,6,8), (2,3)(5,7,6,8) ])
gap> Size(sgp);
360
gap> RepresentativeAction(a8, sgp, preim);
(2,4)(7,8)
```

RepresentativeAction(a8, sgp, preim) が共役を与える元 $g \in G$ を見つける命令である. これより g = (2,4)(7,8) が得られる. 実際, K を共役により移してみると

```
gap> sgp^(2,4)(7,8) = preim;
true
```

により, $K^g = M$ が確かめられる.

6 章 ルービック・キューブ

6.1 ルービック・キューブ群の導入

ルービック・キューブは下図のように立方体を各面を 9 等分してできる 26 個の小立方体 (キューブと呼ぶ) を移動させて, 各面の色を合わせて行くゲームである. 今, 立方体の各面は, 緑 (Green), 赤 (Red), 青 (Blue), 黄色 (Yellow), ピンク (Pink), 白 (White) に色分けされているとする. 以後, 色を表すのに, G, R, B, Y, W, P を使うことにする. また, 座標を固定するために, ルービック・キューブは常に図のように置かれているとする. ただし

$$\text{前面} \Leftrightarrow \mathrm{G}, \quad \text{上面} \Leftrightarrow \mathrm{R}, \quad \text{右面} \Leftrightarrow \mathrm{B}, \quad \text{左面} \Leftrightarrow \mathrm{W}, \quad \text{下面} \Leftrightarrow \mathrm{P}, \quad \text{背面} \Leftrightarrow \mathrm{Y}$$

である. ルービック・キューブの変換は各面の中心にあるキューブを動か さないとして良いので (実際にはこの位置での回転が入るが, 一色になって いる限り区別できない. 図のように文字を使うと, 実はこの文字は回転する) 以後は中心にあるキューブによってその面の色を固定して考える. そこで実質的に動くのは 20 個のキューブである. その内 8 個は立方体の頂点にあり, 残りの 12 個は立方体の辺の上に乗っている. これらをそれぞれ頂点キューブ, 辺キューブと呼ぶことにする.

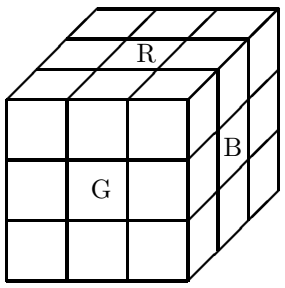

FIGURE 1. ルービック・キューブ $(3 \times 3 \times 3)$

ルービック・キューブの動きは, 軸を中心とする各面での 90 度の回転である. それらを, $x, x', y, y'z, z'$ と記すことにする. x は x 軸を中心とする G 面での回転, x' は x 軸を中心とする裏面 (Y 面) での回転 (以下同様) を表す. x, x', y, y', z, z' を繰り返して得られる変換をルービック変換と云う. ルービック変換は 20 個のキューブの回転付き置換とみることもできる. しかしここでは, 各面上の 8 個の正方形、合計 48 個に下図のように番号を付け, $x, \ldots, z'$ をこれらの 48 文字の置換とみなす. 置換 $x, x', y, y'z, z'$ で生成された S_{48} の部分群をルービック・キューブ群 ということにする.
`x_1, x_2, y_1, y_2, z_1, z_2` を G, Y, B, W, R, P をそれぞれ中心とする左回りの 90 度回転とする.

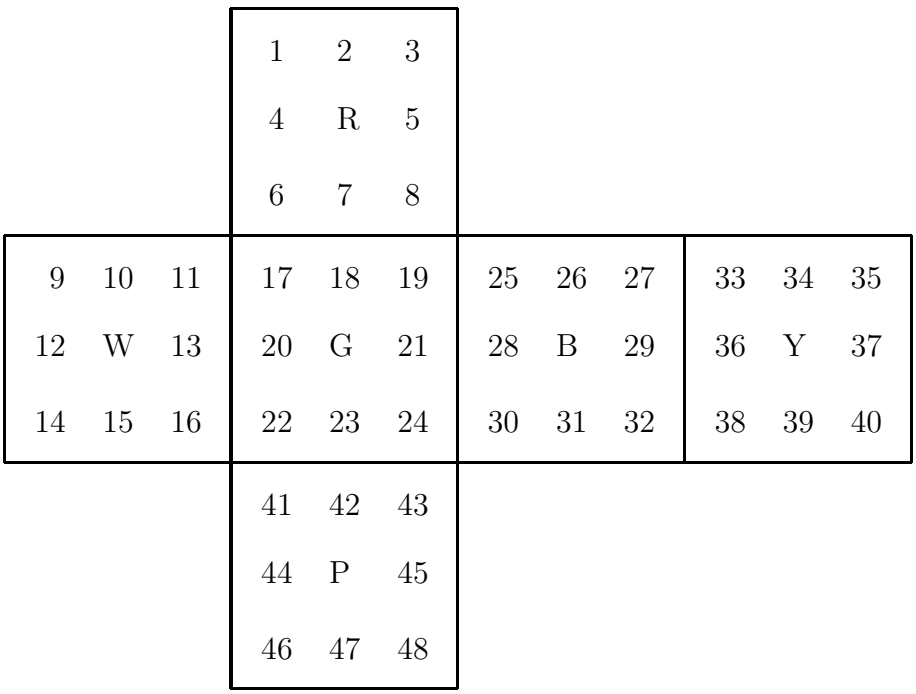

FIGURE 2. ルービック・キューブの展開図

```
gap> x_1 := (17, 19, 24, 22)(18, 21, 23, 20)(6, 25, 43, 16)
            (7, 28, 42, 13)(8, 30, 41, 11);
gap> x_2 := (33, 35, 40, 38)(34, 37, 39, 36)(3, 9, 46, 32)
            (2, 12, 47, 29)(1, 14, 48, 27);
gap> y_1 := (25, 27, 32, 30)(26, 29, 31, 28)(3, 38, 43, 19)
            (5, 36, 45, 21)(8, 33, 48, 24);
gap> y_2 := (9, 11, 16, 14)(10, 13, 15, 12)(1, 17, 41, 40)
            (4, 20, 44, 37)(6, 22, 46, 35);
gap> z_1 := (1, 3, 8, 6)(2, 5, 7, 4)(9, 33, 25, 17)
            (10, 34, 26, 18)(11, 35, 27, 19);
gap> z_2 := (41, 43, 48,46)(42, 45, 47, 44)(14, 22, 30, 38)
            (15, 23, 31, 39)(16, 24, 32, 40);
gap> cube := Group(x_1, x_2, y_1, y_2, z_1, z_2);
gap> Size(cube);
     43252003274489856000
```

これによりルービック・キューブ群 H =cube が位数 43252003274489856000 の S_{48} の部分群として確定する. この巨大な位数を因数分解すると,

```
gap> Collected(Factors(last));
[ [ 2, 27 ], [ 3, 14 ], [ 5, 3 ], [ 7, 2 ], [ 11, 1 ] ]
```

を得る. すなわち, $|H| = 2^{27} \cdot 3^{14} \cdot 5^3 \cdot 7^2 \cdot 11$.

以下, 群 H =cube の 48 点への作用 (置換表現) を詳しく調べていく. まず軌道分解を求めると

```
gap> orbits := Orbits(cube, [1..48]);
   [ [ 1, 14, 17, 3, 48, 9, 22, 19, 41, 38, 8, 27, 24, 46,
       11, 33, 30, 40, 43, 6, 32, 35, 16, 25 ],
     [ 2, 12, 5, 47, 10, 36, 7, 29, 44, 13, 34, 45, 28, 4,
       31, 37, 42, 15, 26, 21, 20, 39, 23, 18 ] ]
gap> List(orbits, c -> Size(c));
   [ 24, 24 ]
```

と 2 個の軌道に分解する. 数字を比べて見れば分かるように 1 番目の軌道が頂点キューブの置換から得られる軌道で, 2 番目の軌道が辺キューブの置換から得られる軌道である. 以後, これらを頂点軌道, 辺軌道ということにする. ルービック・キューブ群 cube の群構造を決定するために cube の頂点軌道への作用と辺軌道への作用をそれぞれ個別に調べる.

6.2 頂点ルービック・キューブ群

まず, cube の頂点軌道 orbits[1] への置換表現を調べる. $f_V : H \to S_{24}$ を H =cube の orbits[1] への置換表現とし, $H_V = \mathrm{Im}(f_V)$ =cube1, f_V =hom1 とおく. H_v を頂点ルービック・キューブ群という.

```
gap> cube1 := Action(cube, orbits[1]);
<permutation group with 6 generators>
gap> Size(cube1);
88179840
gap> hom1 := ActionHomomorphism(cube, orbits[1]);
<action homomorphism>
```

これにより, 置換群 H_V = cube1 $\subset S_{24}$ と全射準同型 f_V = hom1 : $H \to H_V$ が確定する. orbits[1] は H 軌道のひとつだから当然 H_V の作用は推移的である. でも, 念のため確かめてみよう. 人生何事も慎重にやるに越したことはない. まず cube から始めると,

```
gap> IsTransitive(cube, orbits[1]);
true
```

となり何の問題もない. そこで調子の乗って cube1 に移ると

```
gap> IsTransitive(cube1, orbits[1]);
false
```

となって, ヤヤヤ !!ということになる. 大人物はそういう時「コンピュータだってたまには間違うこともあるさ」と泰然自若としていられるのであるが, 我々小人は呆然自失してとまどうばかり. 実は cube1 の作用域は既に [1..24] になっている. orbits[1] のままでは, 作用域は [1..48] の部分集合に過ぎないので推移的にはならないのである. そこで反省し心を入れ換えて,

```
gap> IsTransitive(cube1, [1..24]);
true
```

とすると, cube1 の作用が推移的であることが確かめられる. めでたし, めでたし.

元のお話に戻って cube1 の [1..24] への推移的な作用は果して原始的かどうかを考えよう.

```
gap> corners := Blocks(cube1, [1..24]);
[ [ 1, 6, 22 ], [ 2, 14, 18 ], [ 3, 15, 20 ], [ 4, 12, 16 ],
  [ 5, 10, 21 ], [ 7, 9, 23 ], [ 8, 11, 24 ], [ 13, 17, 19 ] ]
```

従って, cube1 は原始的ではなく, 最小ブロックは 3 個の元からなる. orbits[1] と [1..24] との対応表

1	14	17	3	48	9	22	19	41	38	8	27	24	46	11	33	30	40	43	6	32	35	16	25
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

により, block [1, 6, 22] は元の番号付け [1, 9, 35] に対応し, [2, 14, 18] は [14, 46, 40] に対応する. 以下同様にして, 8 個のブロックが立方体の 8 個の頂点に対応することが分かる (図 4 参照). 従ってこれらのブロックの置換は立方体の頂点の置換に対応し, 各ブロックを動かさない cube1 の元は対応する頂点における頂点キューブの回転に対応する.

次の図のように, $2 \times 2 \times 2$ 個のキューブからなる立体を考える. これはミニキューブとかポケットキューブとか呼ばれているルービック・キューブの変形版である. 上に述べたように $H_V =$cube1 を考えることはもとのルービック・キューブの代わりに, 頂点キューブのみからなるミニキューブを扱うことになる. H_V での生成元 $a_1, a_2, b_1, b_2, c_1, c_2$ が, それぞれ前面, 背面, 右面, 左面, 上面, 背面 での 90 度回転に対応する.

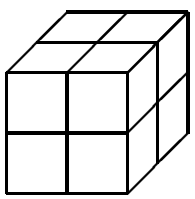

FIGURE 3. ミニキューブ $(2 \times 2 \times 2)$

orbits[1] と [1..24] の対応のもとにミニキューブの展開図を以下に載せておく. 以前に与えた生成元 $a_1, a_2 \ldots, c_2$ の表示はここに載せた番号付けのもとでのミニキューブの動作に一致している.

$H_V =$cube1 のブロックの集合への置換表現 $\varphi_V : H_V \to S_8$ の像を $\mathrm{Im}\,\varphi_V =$cube1b とおく.

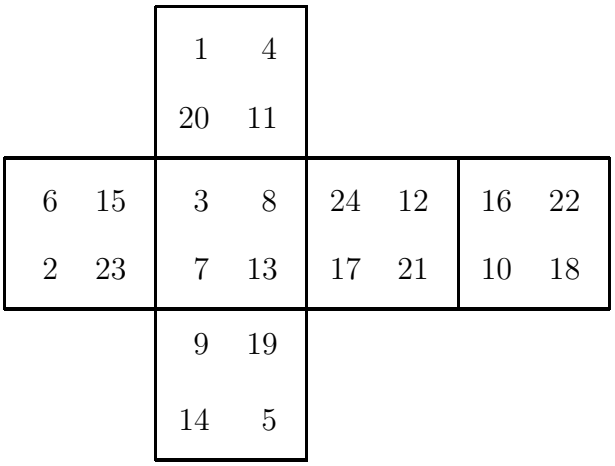

FIGURE 4. ミニキューブの展開図

```
gap> cube1b := Action(cube1, corners, OnSets);
Group([ (3,7,8,6), (1,2,5,4), (4,5,8,7), (1,3,6,2), (1,4,7,3), (2,6,8,5) ])
gap> Size(cube1b);
40320
```

cube1b は S_8 の部分群である. 一方, GAP の階乗関数を使って,

```
gap> Factorial(8);
40320
```

従って, S_8 の位数は 40320 で, それは cube1b の位数に一致する. $H_V \simeq S_8$ が成立し, 全射準同型 $\varphi_V : H_V \to S_8$ が得らえた. 次に $\mathrm{Ker}\,\varphi_V$ について調べよう. まず写像 φ_V =blockhom1 を GAP に読み込む.

```
gap> blockhom1 := ActionHomomorphism(cube1, cube1b);
<action homomorphism>
```

$K_V = \mathrm{Ker}\,\varphi_V$ =ker1 について調べてみよう.

```
gap> ker1 := Kernel(blockhom1);
<permutation group with 7 generators>
gap> Size(elab);
2187
gap> Factors(Size(ker1));
[ 3, 3, 3, 3, 3, 3, 3 ]
gap> IsElementaryAbelian(ker1);
true
```

これで, ker1 が位数 3^7 の基本アーベル群になることが分かった. すなわち $K_V = \mathrm{Ker}\,\varphi_V \simeq (\mathbb{Z}/3\mathbb{Z})^7$ が成り立つ. 以上の議論から完全系列

$$1 \longrightarrow (\mathbb{Z}/3\mathbb{Z})^7 \longrightarrow H_V \xrightarrow{\varphi_V} S_8 \longrightarrow 1 \tag{6.2.1}$$

が得られる.

K_V について更に詳しく調べる. 準同形 φ_V はブロックの上への置換として定義されたので, $K_V = \mathrm{Ker}\,\varphi_V$ は各ブロックを全て (集合として) 保存する. それは, ミニキューブの各頂点を全て固定する変換である. ミニキューブでの動きを考慮するとそのような変換は各頂点での頂点キューブの回転に他ならないことが分る. 従って K_V は各頂点での回転から生成される群 $(\mathbb{Z}/3\mathbb{Z})^8$ に含まれる . $K_V \simeq (\mathbb{Z}/3\mathbb{Z})^7$ が $(\mathbb{Z}/3\mathbb{Z})^8$ のどのような部分群かを決定しよう. まず, 8 個の頂点を

```
[1,6,22], [4,12,16], [8,11,24], [3,15,20],
[7,9,23], [13,17,19], [5,10,21], [2,14,18]
```

の順番に並べる. これらはミニキューブにおいて辺で結ばれた頂点が隣り合うように並べたものである. i 番目の頂点での 120 度の左回転を ε_i と表す. $\varepsilon_1, \ldots, \varepsilon_8$ が $(\mathbb{Z}/3\mathbb{Z})^8$ の生成元を与える. ε_1 =(1,6,22), ε_2 =(4,16,12) に対して

```
gap> (1,6,22) in cube1;
false
gap> (1,6,22)(4,12,16) in cube1;
true
```

つまり, ε_1 は cube1 に含まれず, $\varepsilon_1\varepsilon_2^{-1}$ は cube1 に含まれることが分る. 言い替えれば, $\varepsilon_1\varepsilon_2^{-1} \in K_V$ が成り立つ. ミニキューブの対称性より $\varepsilon_2\varepsilon_3^{-1}, \ldots, \varepsilon_7\varepsilon_8^{-1}$ も K_V に含まれるはずである. 実際確かめてみると,

```
gap> (4,16,12)(11,8,24) in cube1;
true
gap> (11,24,8)(20,15,3) in cube1;
true
gap> (7,9,23)(3,20,15) in cube1;
true
gap> (13,17,19)(7,23,9) in cube1;
true
gap> (19,13,17)(5,10,21) in cube1;
true
gap> (5,21,10)(14,2,18) in cube1;
true
```

以上より, $\varepsilon_1\varepsilon_2^{-1}, \varepsilon_2\varepsilon_3^{-1}, \ldots, \varepsilon_7\varepsilon_8^{-1}$ が K_V の生成元を与え,

$$K_V = \left\{(\varepsilon_1^{\lambda_1}, \ldots, \varepsilon_8^{\lambda_8}) \in (\mathbb{Z}/3\mathbb{Z})^8 \mid \sum_{i=1}^{8} \lambda_i \equiv 0 \pmod 3\right\} \tag{6.2.2}$$

となることが確かめられた.

最後に完全系列 (6.2.1) が分裂すること, すなわち H_V が $(\mathbb{Z}/3\mathbb{Z})^7$ と S_8 の半直積 $S_8 \ltimes (\mathbb{Z}/3\mathbb{Z})^7$ に同型になることを示そう. もし半直積になるとすれば, H_V は S_8 と同型な部分群を含むはずであり, この部分群は頂点キューブの (回転なしの) 置換の全体として得られるはずである. そこで,8 個の頂点の集合 [1,2,3,4,5,7,8,13] の置換として得られる H_V =cube1 の部分群を comp1 とする. 言い替えると comp1 は, [1,2,3,4,5,7,8,13] の集合としての固定化群になっている. そこで

```
gap> comp1 := Stabilizer(cube1, [1,2,3,4,5,7,8,13], OnSets);
<permutation group of size 40320 with 8 generators>
```

により, S_8 と同じ位数を持つ cube1 の部分群 comp1 が定義される. $\varphi_V : H_V \to S_8$ の comp1 への制限は全射になる. 実際

```
gap> Action(comp1, corners, OnSets) = cube1b;
true
```

従って, φ_V の comp1 への制限は同型 comp1 $\simeq S_8$ を与え, これより (6.2.1) が分裂することが導かれる. 直接, 半直積になることを確かめるには, comp1 $\cap K_V = 1$ であることと, H_V が K_V と comp1 で生成されることを見れば良い.

```
gap> Size(Intersection(comp1, ker1));
1
gap> Closure(comp1, ker1) = cube1;
true
```

以上で, $H_V \simeq S_8 \ltimes (\mathbb{Z}/3\mathbb{Z})^7$ となることが確かめられた.

注意. 一般に対称群 S_n と巡回群 $\mathbb{Z}/r\mathbb{Z}$ とのレス積, すなわち S_n と $(\mathbb{Z}/r\mathbb{Z})^n$ との半直積 $S_n \ltimes (\mathbb{Z}/r\mathbb{Z})^n$ (S_n は $(\mathbb{Z}/r\mathbb{Z})^n$ に置換として作用) を $G(r,1,n)$ と表す (複素鏡映群としての記号). $(\mathbb{Z}/r\mathbb{Z})^n$ の生成元を $\varepsilon_1, \ldots, \varepsilon_n$ とするとき半直積 $S_n \ltimes E_r$ を $G(r,r,n)$ と表す. ただし E_r は

$$E_r = \left\{ (\varepsilon_1^{\lambda_1}, \ldots, \varepsilon_n^{\lambda}) \in (\mathbb{Z}/r\mathbb{Z})^{n-1} \mid \sum_{i=1}^{n} \lambda_i \equiv 0 \pmod{r} \right\}$$

で定義される $(\mathbb{Z}/r\mathbb{Z})^n$ の部分群である. $G(r,r,n)$ は位数 $n! \times r^{n-1}$ の複素鏡映群になる. 以上の記号のもとに, $H_V \simeq G(3,3,8)$ と表される.

6.3 辺ルービック・キューブ群

次に cube の辺軌道 orbits[2] への置換表現を調べる. $f_E : H \to S_{24}$ を H =cube の orbits[2] への置換表現とし, $H_E = \mathrm{Im}(f_E)$ =cube2, f_E =hom2 とおく. H_E を辺ルービック・キューブ群という.

```
gap> cube2 := Action(cube, orbits[2]);
<permutation group with 6 generators>
gap> Size(cube2);
980995276800
gap> hom2 := ActionHomomorphism(cube, orbits[2]);
<action homomorphism>
```

辺ルービック・キューブ群 H_E が位数 980995276800 の群として確定する. H_E =cube2 の oribts[2] への作用は推移的であるから, 前と同様にブロック分解ができる.

```
gap> edges := Blocks(cube2, [1..24]);
[ [ 1, 11 ], [ 2, 16 ], [ 3, 19 ], [ 4, 22 ], [ 5, 14 ],
  [ 6, 8 ], [ 7, 24 ],  [ 9, 18 ], [ 10, 21 ], [ 12, 15 ],
  [ 13, 20 ], [ 17, 23 ] ]
```

orbits[2] と [1..24] との対応表は次のようになる.

2	12	5	47	10	36	7	29	44	13	34	45	28	4	31	37	42	15	26	21	20	39	23	18
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

この対応表により, 次の辺キューブの展開図が得られる.

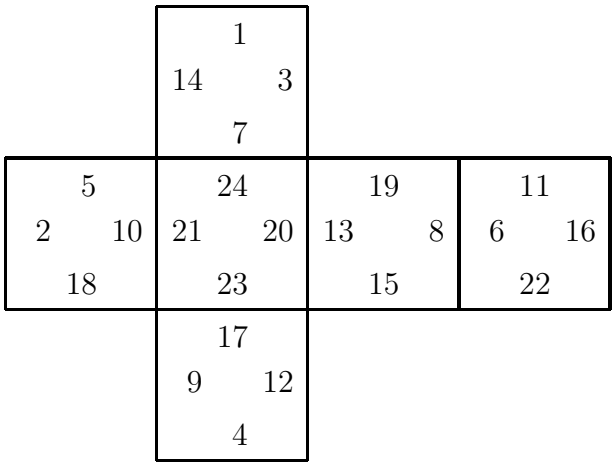

FIGURE 5. 辺キューブの展開図

これより, 各ブロックが 12 個の辺キューブに対応していることが分かる.

H_E =cube2 のブロックの集合への置換表現 $\varphi_E : H_E \to S_{12}$ の像を $\mathrm{Im}\,\varphi_E$ =cube2b とおく.

```
gap> cube2b := Action(cube2, edges, OnSets);
Group([ ( 7,11,12, 9), ( 1, 2, 4, 6), ( 3, 6,10,11),
        ( 2, 5, 9, 8), ( 1, 3, 7, 5), ( 4, 8,12,10) ])
gap> Size(cube2b);
479001600
```

cube2b は S_{12} の部分群であり, 前と同様に $12!$ を計算して

```
gap> Factorial(12);
479001600
```

cube2b $\simeq S_{12}$ を得る. 特に, $\varphi_E : H_E \to S_{12}$ は全射準同型になる. φ_E =blockhom2, $\mathrm{Ker}\,\varphi_E = K_V$ =ker2 とおく. K_E =ker2 を計算しよう.

```
gap> blockhom2 := ActionHomomorphism(cube2, cube2b);
<action homomorphism>
gap> ker2 := Kernel(blockhom2);
<permutation group with 11 generators>
gap> Size(ker2);
2048
gap> Factors(Size(ker2));
[ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> IsElementaryAbelian(ker2);
true
```

以上により, ker2 が位数 2^{11} の基本アーベル群であることが分かった. $K_E \simeq (\mathbb{Z}/2\mathbb{Z})^{11}$ であるから, 完全系列

$$1 \longrightarrow (\mathbb{Z}/2\mathbb{Z})^{11} \longrightarrow H_E \xrightarrow{\varphi_E} S_{12} \longrightarrow 1 \tag{6.3.2}$$

が得られる.

$K_E \simeq (\mathbb{Z}/2\mathbb{Z})^{11}$ の構造を決めよう. 辺キューブの動きを見ることにより, K_E は $(\mathbb{Z}/2\mathbb{Z})^{12}$ に含まれる. ここに各成分 $\mathbb{Z}/2\mathbb{Z}$ は辺キューブの 180 度回転に対応する. 前と同様に 12 個のブロックに次のような順序を入れる.

```
[1,11], [3,19], [7,24], [5,14], [2,16], [9,18],
[10,21], [17,23], [13, 20], [12,15], [6,8], [4,22]
```

この順序で隣り合ったブロックに対応する辺キューブは立方体の中で最短距離を取るものになっている. つまり, 隣り合った辺キューブの位置関係はどれも同じである. i 番目のブロックに対応する辺ブロックの 180 度の回転を ε_i とおく. $\varepsilon_1, \ldots, \varepsilon_{12}$ が $(\mathbb{Z}/2\mathbb{Z})^{12}$ の生成元を与える. $\varepsilon_1 = (1, 11)$, $\varepsilon_2 = (3, 19)$ に対し

```
gap> (1,11) in cube2;
false
gap> (1,11)(3,19) in cube2;
true
```

従って, $\varepsilon_1 \notin K_E$, $\varepsilon_1\varepsilon_2 \in K_E$ である. $\varepsilon_1\varepsilon_2, \varepsilon_2\varepsilon_3, \ldots \varepsilon_{11}\varepsilon_{12}$ は全てルービック・キューブの中で対称な位置を占めることから, $\varepsilon_i\varepsilon_{i+1} \in K_E$ $(1 \leq i \leq 11)$ となることが分かる. これより,

$$K_E = \left\{(\varepsilon_1^{\lambda_1}, \ldots, \varepsilon_{12}^{\lambda_{12}}) \in (\mathbb{Z}/2\mathbb{Z})^{12} \mid \sum_{i=1}^{12} \lambda_i \equiv 0 \pmod 2\right\}$$

が得られる.

最後に, 完全系列 (6.3.2) が分裂すること, すなわち H_E が S_{12} と $(\mathbb{Z}/2\mathbb{Z})^{11}$ の半直積になることを示す.

```
gap> comp2 := Stabilizer(cube2, [1,2,3,4,5,6,7,9,10,12,13,17], OnSets);
<permutation group of size 479001600 with 8 generators>
gap> Action(comp2, edges, OnSets) = cube2b;
true
```

頂点キューブの場合と同様に, cube2 の中に S_{12} と同じ位数を持つ部分群 comp2 が定義され, $\varphi_E : H_E \to S_{12}$ の comp2 への制限は同型 comp2 $\simeq S_{12}$ を与える. 従って, $H_E \simeq S_{12} \ltimes (\mathbb{Z}/2\mathbb{Z})^{11}$ が得られる. 前節の記号に従えば, これは $H_E \simeq G(2, 2, 12)$ を意味する.

6.4 ルービック・キューブ群の決定

ルービック・キューブ群 H は 48 点の置換群であり, その 48 点がふたつの軌道 orbits[1] と orbits[2] に分解する. H から誘導された orbits[1] 上の置換群が H_V であり, orbits[2] 上の置換群が H_E である. $f_V : H \to H_V$, $f_E : H \to H_E$ を前節で定義した全射準同型とすると $f : H \to H_V \times H_E, g \mapsto (f_V(g), f_E(g))$ は単射準同型を与える. $\varphi_V : H_V \to S_8$, $\varphi_E : H_E \to S_{12}$ を前節のように取る. 次の図式を考える.

$$\widetilde{f} : H \xrightarrow{f} H_V \times H_E \xrightarrow{\varphi_V \times \varphi_E} S_8 \times S_{12} \tag{6.4.1}$$

S_8 は文字 $\{1, 2, \ldots, 8\}$, S_{12} を文字 $\{9, 10, \ldots, 21\}$ に関する置換群とみて $S_8 \times S_{12}$ を S_{21} の部分群とみなす. $A(S_8 \times S_{12}) = (S_8 \times S_{12}) \cap A_{21}$ とおく (A_{21} は 21 次の交代群). $A(S_8 \times S_{12})$ は $S_8 \times S_{12}$ の指数 2 の部分群になる. 次を示す.

$$\widetilde{f}(H) \subset A(S_8 \times S_{12}). \tag{6.4.2}$$

実際 $g = x_1, x_2, \ldots, z_2$ に対して, $\varphi_V \circ f_V(g)$ は位数 4 の巡回置換になる. これは $\varphi \circ f_V$ が回転を無視した頂点キューブの置換であることより当然の結果であるが, 直接確かめようとすれば

```
gap> hom11 := hom1*blockhom1;;
gap> Image(hom11, x_1); Image(hom11, x_2);
(3,7,8,6)
(1,2,5,4)
gap> Image(hom11, y_1); Image(hom11, y_2);
(4,5,8,7)
```

```
(1,3,6,2)
gap> Image(hom11, z_1); Image(hom11, z_2);
(1,4,7,3)
(2,6,8,5)
```

同様に, $\varphi_E \circ f_E(g)$ もまた位数 4 の巡回置換である.

```
gap> hom22 := hom2*blockhom2;;
gap> Image(hom22, x_1); Image(hom22, x_2);
(7,11,12,9)
(1,2,4,6)
gap> Image(hom22, y_1); Image(hom22, y_2);
(3,6,10,11)
(2,5,9,8)
gap> Image(hom22, z_1); Image(hom22, z_2);
(1,3,7,5)
(4,8,12,10)
```

上の結果から $g = x_1, x_2, \ldots, z_2$ に対して, $\widetilde{f}(g) = (\varphi_V \circ f_V(g), \varphi_E \circ f_E(g)) \in S_8 \times S_{12}$ は位数 4 の巡回置換の積である. 従って, $\widetilde{f}(g) \in A(S_8 \times S_{12})$. $x_1, x_2, \ldots, z_2$ は H の生成元であるから, これより (6.4.2) が得られる.

最後に

$$H \simeq f(H) = A(S_8 \times S_{12}) \ltimes ((\mathbb{Z}/3\mathbb{Z})^7 \times (\mathbb{Z}/2\mathbb{Z})^{11}) \tag{6.4.3}$$

を示そう. ルービック・キューブ群 H の構造は (6.4.3) により完全に記述される.

(6.4.2) より, $H \simeq f(H) \subset (\varphi_V \times \varphi_E)^{-1}(A(S_8 \times S_{12}))$. ここで

$$(\varphi_V \times \varphi_E)^{-1}(A(S_8 \times S_{12})) \simeq A(S_8 \times S_{12}) \ltimes ((\mathbb{Z}/3\mathbb{Z})^7 \times (\mathbb{Z}/2\mathbb{Z})^{11})$$

は $H_V \times H_E$ の指数 2 の部分群になる. そこで, $H =$cube の位数と $H_V \times H_E =$ cube1 $\times$ cube2 の位数の半分を比較して

```
gap> Size(cube) = Size(cube1)*Size(cube2)/2;
true
```

これより $f(H) = (\varphi_V \times \varphi_E)^{-1}(A(S_8 \times S_{12}))$ となり, (6.4.3) が成り立つ.

最後にルービック・キューブ群 H の中心を調べておこう.

```
gap> Center(cube);
Group([ ( 2,34)( 4,10)( 5,26)( 7,18)(12,37)(13,20)
        (15,44)(21,28)(23,42)(29,36)(31,45)(39,47) ])
```

すなわち, H の中心は $\{1, g\}$ からなる. ここで

$$g = (2,34)(4,10)(5,26)(7,18)(12,37)(13,20)(15,44)(21,28)(23,42)(29,36)(31,45)(39,47)$$

は, 12 個の辺キューブを全て 180 度ひっくり返す位数 2 の変換である.

6.5 生成元による表示

ルービック・キューブ群 H は $x_1, x_2, y_1, y_2, z_1, z_2$ から生成されていた. 与えられた H の元をこれらの生成元の積として表すことを考えてみよう. これがルービック・キューブの「バラバラになったパターンを如何にして元の状態に復元させるか」というゲームの主要テーマに他ならない.

まず頂点をそろえることから始める. H の生成元 $x_1, x_2, y_1, y_2, z_1, z_2$ の H_V での像をそれぞれ $a_1, a_2, b_1, b_2, c_1, c_2$ とおくと以下のようになる.

```
gap> a_1:= Image(hom1, x_1); a_2 := Image(hom1, x_2);
( 3, 8,13, 7)( 9,15,11,17)(19,23,20,24)
( 1, 2, 5,12)( 4, 6,14,21)(10,16,22,18)
gap> b_1 := Image(hom1, y_1); b_2 := Image(hom1, y_2);
( 4,10,19, 8)( 5,13,11,16)(12,21,17,24)
( 1, 3, 9,18)( 2, 6,15,23)( 7,14,22,20)
gap> c_1 := Image(hom1, z_1); c_2 := Image(hom1, z_2);
( 1, 4,11,20)( 3, 6,16,24)( 8,15,22,12)
( 2, 7,17,10)( 5,14, 9,19)(13,21,18,23)
```

H_V の元 $q = a_1 b_1 a_1^{-1}$, $r = c_1^{-1} b_1^{-1} c_1 b_1$ を考える.

```
gap> q := a_1*b_1*a_1^-1;
( 3, 4,10,24)( 5, 8,15,16)(11,20,12,21)
gap> r := c_1^-1*b_1^-1*c_1*b_1;
( 3,24,20,11,15, 8)( 4,10,16, 5,12,21)
```

$q, r \in H_V$ の $\varphi_V : H_V \to S_8$ への像を計算すると,

```
gap> qq := Image(blockhom1, q);
(3,4,5,7)
gap> rr := Image(blockhom1,r);
(3,7)(4,5)
gap> rr*qq;
(4,7)
```

特に rq の像は S_8 の互換を与える. つまり rq は回転を無視すれば互換になる. rq 自身を書いてみると

```
gap> r*q;
( 4,24,12,11,16, 8)( 5,21,10)
```

すなわち rq はブロック [4,12,16] と [8,11,24] に対応する頂点を交換し, ブロック [5,10,21] に対応する頂点の回転を引き起こす. Figure 4 を参照して, $rq = (c_1^{-1}b_1^{-1}c_1b_1)(a_1b_1a_1^{-1})$ がミニキューブの隣り合った 2 頂点の置換であることが分かる. ルービック・キューブの対称性から, 生成元 a_1, b_1, c_1 を適当に他の生成元と取り換えることにより任意の隣り合った 2 頂点の間の置換が同様にして得られる. 今 $g \in H_V$ を取る. $\varphi_V(g) \in S_8$ はこれらの互換の積で書けるので, g に右から rq 達を繰り返し掛けることにより $g' \in K_V = \mathtt{ker1}$ に変形することができる. (言い替えると, rq 達を繰り返し適用することにより, ミニキューブの各頂点を (回転を除いて) 正しい位置に持って来ることができる).

そこで $g \in K_V$ と仮定してよい. $\varphi_V(rq)$ は互換だから, $(rq)^2 \in K_V$ である.

```
gap> (r*q)^2 in ker1;
true
```

計算してみると,

```
gap> (r*q)^2;
( 4,12,16)( 5,10,21)( 8,24,11)
```

すなわち $(rq)^2$ は 3 頂点 [4,12,16], [5,10,21], [8,24,11] おける 120 度の右回転である. 一方, a_1, b_1, c_1 を c_1, b_1, a_2 に取り換えて,

```
gap> r_1 := a_2^-1*b_1^-1*a_2*b_1;
( 1,12,22,16, 6, 4)( 5,13,21,17,10,19)
gap> q_1 := c_1*b_1*c_1^-1;
( 1,10,19,12)( 4, 6, 5,13)(16,22,21,17)
gap> r_1*q_1;
( 4,10,12,21,16, 5)(13,17,19)
gap> (r_1*q_1)^2;
( 4,12,16)( 5,10,21)(13,19,17)
```

そこで, $(r_1q_1)^{-2}(rq)^2$ を計算すると

```
gap> (r_1*q_1)^-2*(r*q)^2;
( 8,24,11)(13,17,19)
```

これは, となりあった 2 頂点 [8,11,24], [13,17,19] での右 120 度, 左 120 度の回転である. 生成元を取り換えることにより, 全ての隣り合った 2 頂点で, このような変換を作ることができて, それらが, K_V を生成する. 従って $g \in K_V$ は, $(r_1q_1)^{-2}(rq)^2$ の形の元の積で書ける. 以上の操作で頂点は全て (回転も含めて) 正しい位置に持って行くことができる.

$p_V : H \to H_V$ を $H_V \times H_E$ から H_V への射影を H に制限したものとする. $\operatorname{Ker} p_V = H \cap H_E \simeq A_{12} \ltimes (\mathbb{Z}/2\mathbb{Z})^{11}$ である. 今までの議論から $g \in \operatorname{Ker} p_V$ (隅はすべてそろっている) としてよい. $u = (x_1^2y_1^2)^3$ とおく.

```
gap> u := (x_1^2*y_1^2)^3;
( 5,45)( 7,42)(18,23)(26,31)
gap> Image(hom2, u);
( 3,12)( 7,17)(15,19)(23,24)
```

従って, u は Figure 5 における 辺キューブ [7,24] と [17,23] を入れ換え, [3,19] と [12,15] を入れ換え, 他はすべて固定する変換である. $\varphi_E \circ f_E(u) \in S_{12}$ は互換 2 個の積になる. $\varphi_E \circ f_E : H \to S_{12}$ は全射なので, A_8 の全ての元を u の共役元いくつかの積 (の像) として構成できる. そこで g に u の共役元達を適当に掛けることにより, $\varphi_E \circ f_E(g) = 1$, すなわち $g \in (\mathbb{Z}/2\mathbb{Z})^{11}$ とできる. 回転を除いて辺の位置はを全てそろった状態になる.

最後に, $(\mathbb{Z}/2\mathbb{Z})^{11}$ の元を H の生成元を使って現わそう. $v_1 = y_1^{-1}(x_2 y_1 x_2^{-1} y_1^{-1})$ とおき, u の共役元 $w_1 = v_1^{-1} u v_1$ を考える.

```
gap> v_1 := y_1^-1*(x_2*y_1*x_2^-1*y_1^-1);
( 3,27,33)( 5,21,45)( 8,24,46,32,25,30,40,38,19,43,14,48)
(26,28,31)(29,47)(36,39)
gap> w_1 := v_1^-1*u*v_1;
( 5,21)( 7,42)(18,23)(26,28)
gap> Image(hom2, v_1);
( 3,20,12)( 4, 8)( 6,22)(13,15,19)
gap> Image(hom2, w_1);
( 3,20)( 7,17)(13,19)(23,24)
```

w_1 の f_E での像は, 辺キューブ [7,24] と [17,23] を入れ換え, [3,19] と [20,13] をこの順序で入れ換える. 一方, $v_2 = x_1^{-1} z_1^{-1} x_1 y_1^2 x_2^{-1} y_1^{-1}$ とおき, u の共役元 $w_2 = v_2^{-1} u v_2$ を考える.

```
gap> v_2 := x_1^-1*z_1^-1*x_1*y_1^2*x_2^-1*y_1^-1;
( 1,46, 9,40,35,14)( 2, 4,47,12)( 3,25,33,19,27, 8)( 5,26)(10,39,37,34)
(21,29,31,28,36,45)(24,32,30,38,43,48)
gap> Image(hom2, v_2);
( 1,14, 4, 2)( 3,19)( 5,22,16,11)( 6,12,20, 8,15,13)
gap> w_2 := v_2^-1*u*v_2;
( 5,28)( 7,42)(18,23)(21,26)
gap> Image(hom2, w_2);
( 3,13)( 7,17)(19,20)(23,24)
```

w_2 の f_E での像は, 辺キューブ [7,24] と [17,23] を入れ換え, [3,19] と [13,20] をこの順序でいれ換える. そこで

```
gap> w_1^-1*w_2;
( 5,26)(21,28)
gap> Image(hom2, w_1^-1*w_2);
( 3,19)(13,20)
```

すなわち $w_1^{-1}w_2$ は最短距離にある 2 個の辺キューブを位置を動かさずにそれぞれ 180 度回転させる. 特に $w_1^{-1}w_2 \in (\mathbb{Z}/2\mathbb{Z})^{11}$ である. ルービック・キューブの対称性から最短距離にあるどんな 2 個の辺キューブについても同様の変換が (生成元を取り換えることにより) 構成できる. $(\mathbb{Z}/2\mathbb{Z})^{11}$ はこれらの変換により生成されるので, 結局辺キューブを (回転まで含めて) 全て正しい位置に持っていくことができる. これで $g \in H$ は生成元の積で表され, 同時にゲームが完成する.

6.6 拡張版ルービック・キューブ $(3 \times 3 \times 3)$

ルービック・キューブを構成している各キューブが色ではなく, 下図のように文字によって識別されているとする.

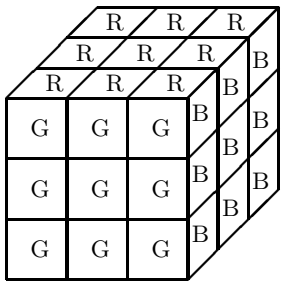

FIGURE 6. 拡張版ルービック・キューブ $(3 \times 3 \times 3)$

この場合, 頂点キューブと辺キューブについては前と変わりないが, 中心キューブについてはその位置での 90 度回転も考慮する必要が生じる. 今までの結果から, 通常のルービック・キューブ群 H は S_{48} の部分群として構成されている. そこで, 6 個の中心キューブの, それぞれ 90 度回転に合わせて, 24 個の文字, $\{49, 50, \ldots, 72\}$ を付け加え, S_{72} の部分群として拡張版ルービック・キューブ群 $\widetilde{H}$ を構成する. H の生成元 $x_1, \ldots, z_2$ に対応するルービック・キューブの操作によって各面の中心キューブの回転が誘導される. そこで $\widetilde{H}$ の生成元として `xx_1`, `xx_2`, `yy_1`, `yy_2`, `zz_1`, `zz_2` を次のようにおく.

```
gap> xx_1 := x_1*(49,50,51,52);;
gap> xx_2 := x_2*(53,54,55,56);;
gap> yy_1 := y_1*(57,58,59,60);;
gap> yy_2 := y_2*(61,62,63,64);;
gap> zz_1 := z_1*(65,66,67,68);;
gap> zz_2 := z_2*(69,70,71,72);;
```

そこで $\widetilde{H} =$ `excube` を以下のように定義する.

```
gap> excube := Group(xx_1, xx_2, yy_1, yy_2, zz_1, zz_2);
<permutation group with 6 generators>
gap> Size(excube);
88580102706155225088000
```

$H \times (\mathbb{Z}/4\mathbb{Z})^6$ の部分群 $\widetilde{H}$ の構造を決定しよう. $p : \widetilde{H} \to H$ を射影 $H \times (\mathbb{Z}/4\mathbb{Z})^6 \to H$ の $\widetilde{H}$ への制限とする. $p(xx_1) = x_1, \ldots, p(zz_2) = z_2$ より, p は全射準同型になる. $\widetilde{K} = \mathrm{Ker}\, p$ とおく. 次の完全系列を考える.

$$1 \longrightarrow \widetilde{K} \longrightarrow \widetilde{H} \longrightarrow H \longrightarrow 1 \tag{6.6.1}$$

ここで $\widetilde{K} = \widetilde{H} \cap (\mathbb{Z}/4\mathbb{Z})^6$ であり, また

```
gap> Size(excube)/Size(cube);
2048
gap> 4^6;
4096
```

より, $\widetilde{K}$ は $(\mathbb{Z}/4\mathbb{Z})^6$ の指数 2 の部分群になる. $\widetilde{K}$ の構造を調べる. $(\mathbb{Z}/4\mathbb{Z})^6$ は, 各面での 90 度回転に対応している. 今, 6 個の面の順番を隣り合った面は常に辺を共有するように定め, その生成元を $\varepsilon_1, \ldots, \varepsilon_6$ とする. 順番の決め方から $\varepsilon_1 = (49, 50, 51, 52)$ が x_1 に対応し, $\varepsilon_2 = (57, 58, 59, 60)$ が y_1 に対応するとしてよい. このとき

```
gap> (49,50,51,52) in excube;
false
gap> (49,50,51,52)(57,58,59,60) in excube;
true
gap> (49, 50,51,52)^2 in excube;
true
```

従ってルービック・キューブの対称性から, $\varepsilon_i^2 \in \widetilde{K}, \varepsilon_i\varepsilon_{i+1} \in \widetilde{K}$ が成り立つ. これより

$$\widetilde{K} = \left\{(\varepsilon_1^{\lambda_1}, \ldots, \varepsilon_6^{\lambda_6}) \in (\mathbb{Z}/4\mathbb{Z})^6 \mid \sum_{i=1}^{6} \lambda_i \equiv 0 \pmod 2\right\} \tag{6.6.2}$$

が得られる. これで一応 $\widetilde{H}$ の構造は分かったが, 完全系列 (6.6.1) は分裂しないので, (6.6.1) だけではやはり不満が残る. そこで $H \times (\mathbb{Z}/4\mathbb{Z})^6$ の部分群としての $\widetilde{H}$ のより精密な表示を追求してみる. まず次に注意する.

$$(\mathbb{Z}/3\mathbb{Z})^7 \times (\mathbb{Z}/2\mathbb{Z})^{11} \subset \widetilde{H}. \tag{6.6.3}$$

実際, $(11, 24, 8), (20, 15, 3) \in H_V$ は `[1..24]` $\Leftrightarrow$ `orbits[1]` により $(25, 19, 8), (11, 17, 6) \in S_{24}$ に対応し, $(7, 24), (14, 5) \in H_E$ は `[1..24]` $\Leftrightarrow$ `orbits[2]` により $(7, 18), (4, 10)$ に対応する. このとき

```
gap> (25,19,8)(11,17,6) in cube;
true
gap> (25,19,8)(11,17,6) in excube;
true
gap> (7,18)(4,10) in cube;
true
```

```
gap> (7,18)(4,10) in excube;
true
```

従って, $(25,19,8)(11,17,6),\ (7,18)(4,10) \in \widetilde{H}$ が分かる. ルービック・キューブの対称性から, $(\mathbb{Z}/3\mathbb{Z})^7$, $(\mathbb{Z}/2\mathbb{Z})^{11}$ の生成元はすべて $\widetilde{H}$ に含まれる. よって (6.6.3) が成り立つ.

(6.6.3) より次が導かれる.

$$\widetilde{H} \simeq \widetilde{H}_1 \ltimes ((\mathbb{Z}/3\mathbb{Z})^7 \times (\mathbb{Z}/2\mathbb{Z})^{11})). \tag{6.6.4}$$

ここで $\widetilde{H}_1 = \widetilde{H} \cap (A(S_8 \times S_{12}) \times (\mathbb{Z}/4\mathbb{Z})^6)$ は $A(S_8 \times S_{12}) \times (\mathbb{Z}/4\mathbb{Z})^6$ の指数 2 の部分群である (半直積での $(\mathbb{Z}/4\mathbb{Z})^6$ の作用は自明な作用とする). $\widetilde{H}$ を決めるためには, $\widetilde{H}_1$ の構造を決めればよい. $A_8 \times A_{12}$ は $A(S_8 \times S_{12})$ の部分群であることに注意して次を示す.

$$A_8 \times A_{12} \subset \widetilde{H}_1. \tag{6.6.5}$$

まず $A(S_8 \times S_{12})$ の H への埋め込みを具体的に決めることから始める.

```
gap> Orbits(comp1, [1..24]);
[ [ 1, 2, 3, 4, 5, 13, 7, 8 ], [ 6, 14, 15, 16, 21, 17, 9, 11 ],
  [ 10, 18, 20, 12, 19, 23, 24, 22 ] ]
gap> Orbits(comp2, [1..24]);
[ [ 1, 2, 3, 4, 5, 6, 17, 13, 9, 12, 10, 7 ],
  [ 8, 16, 18, 19, 22, 14, 23, 20, 15, 21, 11, 24 ] ]
```

S_8 は comp1 として H_V の中に実現された. 従って, S_8 は, 次の順序付けられた 8 個の組の置換として得られる.

$$[1,6,22],\ [2,14,18],\ [3,15,20],\ [4,16,12],\ [5,21,10],\ [13,17,19],\ [7,9,23],\ [8,11,24]$$

orbits[1] の上では, 対応表により

$$[1,9,35],\ [14,46,40],\ [17,11,6],\ [3,33,27],\ [48,32,38],\ [24,30,43],\ [22,41,16],\ [19,8,25]$$

となる. 同様に S_{12} は comp2 として H_E の中に実現されるので, S_{12} は順序付けられた 12 個の組の置換として実現される.

$$[1,11],\ [2,16],\ [3,19],\ [4,22],\ [5,14],\ [6,8],\ [17,23],\ [13,20],\ [9,18],\ [12,15],\ [10,21],\ [7,24]$$

orbits[2] の上では

$$[2,34],\ [12,37],\ [5,26],\ [47,39],\ [10,4],\ [36,29],\ [42,23],\ [28,21],\ [44,15],\ [45,31],\ [13,20],\ [7,18]$$

となる. これより, 例えば 3 個の頂点キューブ [1,9,35], [14,46,40], [17,11,6] に関する長さ 3 の巡回置換は H の元として $(1,14,17)(9,46,11)(35,40,6)$ と表される. そこで

```
gap> (1,14,17)(9,46,11)(35,40,6) in cube;
true
gap> (1,14,17)(9,46,11)(35,40,6) in excube;
true
```

同様に 3 個の辺キューブ [2,34], [12,37], [5,26] に関する長さ 3 の巡回置換は H の元として $(2,12,5)(34,37,26)$ と表される. そこで

```
gap> (2,12,5)(34,37,26) in cube;
true
gap> (2,12,5)(34,37,26) in excube;
true
```

上の結果は S_8 と S_{12} の (少なくとも一つの) 長さ 3 の巡回置換が $\widetilde{H}$ に含まれることを意味する. ところで, $A(S_8\times S_{12})\times(\mathbb{Z}/4\mathbb{Z})^6$ は明らかに, $S_8\times S_{12}$ の共役の作用により不変である. $\widetilde{H}_1$ は $A(S_8\times S_{12})\times(\mathbb{Z}/4\mathbb{Z})^6$ の指数 2 の部分群なのでやはり, $S_8\times S_{12}$ の作用で不変になる. したがって S_8 のすべての長さ 3 の巡回置換, S_12 のすべての長さ 3 の巡回置換が $\widetilde{H}_1$ に含まれることが分かる. 交代群は長さ 3 の巡回置換によって生成されるので, これより (6.6.5) が得られる.

さて (6.6.1) と (6.6.5) を合わせて, $\widetilde{H}_1$ は $(A_8\times A_{12})\times\widetilde{K}$ を指数 2 の部分群として含むことが分かる. 残りの剰余類の代表元は, 例えば

$$g=(1,14)(9,46)(35,40)\times(2,12)(34,37)\times(49,50,51,52)$$

で与えられる. 第 1 項が頂点キューブ [1,9,35] と [14,46,40] の互換, 第 2 項が辺キューブ [2,34] と [12,37] の互換, 第 3 項が中心キューブの 90 度回転である. これにより,

$$\widetilde{H}_1=\big(A_8\times A_{12})\times\widetilde{K}\big)\bigcup\big((A_8\times A_{12})\times\widetilde{K}\big)g \tag{6.6.6}$$

と表される. この結果か次のようにすれば, より見やすくなる. 今,

$$\begin{aligned}&\psi_1:A(S_8\times S_{12})\to A(S_8\times S_{12})/(A_8\times A_{12})\simeq\{\pm1\},\\&\psi_2:(\mathbb{Z}/4\mathbb{Z})^6\to(\mathbb{Z}/4\mathbb{Z})^6/\widetilde{K}\simeq\{\pm\}\end{aligned}$$

により 準同型 ψ_1,ψ_2 を定義する. 準同型 $\psi:A(S_8\times S_{12})\times(\mathbb{Z}/4\mathbb{Z})^6\to\{\pm1\}$ を $\psi(a,b)=\psi_1(a)\psi_2(b)$ により定義する. このとき, $\widetilde{H}_1=\mathrm{Ker}\,\psi$ が成り立つ. 以上の結果をまとめると,

(6.6.7) 拡張版ルービック・キューブ群 $\widetilde{H}$ は

$$\widetilde{H}\simeq\widetilde{H}_1\ltimes((\mathbb{Z}/3\mathbb{Z})^7\times(\mathbb{Z}/2\mathbb{Z})^{11})$$

で与えられる. ここに $\widetilde{H}_1 = \mathrm{Ker}\,\psi$ は $A(S_8 \times S_{12}) \times (\mathbb{Z}/4\mathbb{Z})^6$ の指数 2 の部分群である. 特に,

$$|\widetilde{H}| = \frac{1}{24}(8! \times 12! \times 3^8 \times 2^{12} \times 4^6) = 88580102706155225088000.$$

最後に, 拡張版ルービック・キューブ群 $\widetilde{H}$ の中心 $Z(\widetilde{H})$ を求めておく.

```
gap> Center(excube);
<permutation group of size 4096 with 10 generators>
```

従って $Z(\widetilde{H})$ の位数は 4096 である. g を前節のように H の中心の元とするとき

$$Z(\widetilde{H}) \simeq \langle g \rangle \times \widetilde{K} \tag{6.6.8}$$

が成り立つ. 実際

```
gap> g in Center(excube);
true
gap> (49,50,51,52)(57,58,59,60) in Center(excube);
true
gap> (49,50,51,52)^2 in Center(excube);
true
```

2 番目と 3 番目の式から, (6.6.2) を導いたのと同様の議論で $\widetilde{K} \subset Z(\widetilde{H})$ が云える. そこで $\langle g \rangle \times \widetilde{K}$ と $Z(\widetilde{H})$ の位数を比較して (6.6.8) を得る.