Asir

Asir User's Manual
Asir-20030424 (Kobe Distribution)
April 2003

by Masayuki Noro, Takeshi Shimoyama, Taku Takeshima and Risa/Asir committers

1 Introduction

1.1 マニュアルの構成

このマニュアルの構成は次の通りである.

1. **Introduction**
   マニュアルの構成, notation および入手方法
2. **Risa/Asir**
   Asir の概要, インストレーション
3. **型**
   Asir における型
4. **ユーザ言語 Asir**
   Asir のユーザ言語の解説
5. **デバッガ**
   ユーザ言語のデバッガの解説
6. **組み込み函数**
   さまざまな組み込み函数の詳細
7. **分散計算**
   分散計算機能に関する解説, 関数の詳細
8. **グレブナ基底の計算**
   グレブナ基底に関する函数, 演算の解説
9. **代数的数に関する演算**
   代数的数に関する函数, 演算の解説
10. **有限体に関する演算**
    有限体に関する関数, 演算の解説
11. **付録**
    文法の詳細, サンプルファイルの紹介, 入力インタフェース, 変更点, 文献

1.2 Notation

このマニュアルでは, いくつかの記法上の慣習がある. それらについて説明する.

- 函数名は, タイプライタ体で表される.
  `gcd()`, `gr()`
- 函数の説明における函数の引数は, 斜字体で表される.
  int, *poly*
- ファイル名は, シングルクォートつきのタイプライタ体で表される.
  ‘`dbxinit`’, ‘`asir_plot`’
- 例は字下げされ, タイプライタ体で表される.

```
[0] 1;
1
[1] quit;
```

- 文献の参照は, [] つきのタイプライタ体で表される.
  [Boehm,Weiser]
- 函数の引数で, 省略可能なものは, [] で囲って示される. また, 0 個以上の繰り返しは, []* で示される.
  setprec([*n*]), diff(*rat*[,*varn*]*)
- shell (csh) のプロンプトは % で表される. ただし, インストール時など, root として作業している場合のプロンプトは # で表される.

```
% cat afo
afo
bfo
% su
Password:XXXX
# cp asir /usr/local/bin
# exit
%
```

- 有理整数環は **Z**, 有理数体は **Q**, 実数体は **R**, 複素数体は **C** 有限体は **GF(q)** (**q** は素数冪) で表す.

1.3 入手方法

Risa/Asir (‘asir2000.tgz’), **PARI** のソースコード (‘pari.tgz’) および Windows 用バイナリ (‘asirwin-ja.tgz’, ‘asirwin-en.tgz’) は以下から ftp で入手できる.

```
ftp://ftp.math.kobe-u.ac.jp/pub/asir
```

2 Risa/Asir

2.1 Risa および Asir

Risa は, 富士通研で開発中の数式処理システム/ライブラリの総称である. **Risa** の構成は次の通りである.

- 基本演算部
  これは, **Risa** の内部形式に変換されたオブジェクト (数, 多項式など) の間の演算を実行する部分であり, UNIX の 'libc.a' などと同様の, ライブラリとして存在する. エンジンは, C および アセンブラで記述され, 後述する言語インタフェース **Asir** の基本演算部として用いられている.
- メモリ管理部
  Risa では, メモリ管理部として, [Boehm,Weiser] によるフリーソフトウェア (gc-6.1alpha5) を用いている. これはガーベジコレクション (以下 GC と呼ぶ) を自動的に行うメモリ割り当て機構を持ち, **Risa** の各部分はすべてこれにより必要なメモリを得ている.
- **Asir**
  Asir は, **Risa** の計算エンジンの言語インタフェースである. **Risa** では, 比較的容易にユーザ用の言語インタフェースを作ることができる. **Asir** はその一つの例として作ったもので, C 言語に近い文法をもつ. また, C のデバッガとして広く用いられている dbx 風のデバッガも備えている.

2.2 Asir の特徴

Asir は, 前述の通り, 計算エンジンの言語インタフェースである. 通常 **Asir** という名前の実行可能ファイルとして提供される. 現在サポートされている機能は概ね次の通りである.

- C 言語風のユーザ言語
- 数, 多項式, 有理式の加減乗 (除)
- ベクトル, 行列の演算
- 最小限のリスト処理
- 組み込み函数 (因数分解, GCD, グレブナ基底など)
- ユーザ定義函数によるツール (代数体上の因数分解など)
- dbx 風のデバッガ
- 陰函数の描画
- **PARI** (6.1.14 節「pari」p.39 参照) による初等超越函数を含む式の評価
- UNIX 上での分散計算機能 (Open XM)

2.3 Installation

以下の手続きで不明な点, 不都合な点があった場合の問い合わせは

noro@math.kobe-u.ac.jp

宛に e-mail でお願いします.

2.3.1 UNIX binary version

ターゲット CPU/OS に対応する ‘asir.tgz’ が必要である. これらは全て gzip で圧縮してあるので, 入手後 gzip で展開する. まず, インストールするディレクトリを決める. デフォルトでは ‘/usr/local/lib’ に ‘asir’ というディレクトリとしてインストールされることを仮定している. 以下このディレクトリをライブラリディレクトリと呼ぶ.

```
# gzip -dc asir.tgz | ( cd /usr/local/lib; tar xf - )
```

個人的に使用する場合には, ‘$HOME’などに置いてもよい.

```
% gzip -dc asir.tgz | ( cd $HOME; tar xf - )
```

この場合, ライブラリディレクトリの名前を環境変数 ASIR_LIBDIR に設定する必要がある.

```
% setenv ASIR_LIBDIR $HOME/asir
```

Asir 本体は, ライブラリディレクトリの ‘asir’ である. ‘/usr/local/bin’ あるいはユーザの実行ファイルサーチパスのどこかにシンボリックリンクを作ると便利である.

```
# ln -s /usr/local/lib/asir/asir /usr/local/bin/asir
```

これで ‘asir’ が起動できる.

```
% /usr/local/bin/asir
This is Risa/Asir, Version 20000821.
Copyright (C) FUJITSU LABORATORIES LIMITED.
1994-2000. All rights reserved.
[0]
```

2.3.2 UNIX source code version

まず, インストール先のディレクトリを決める必要がある. ここには, 以下のサブディレクトリが置かれる.

- bin
  PARI および Asir の実行可能ファイル
- lib
  PARI および Asir のライブラリ
- include
  PARI のヘッダファイル

これらのサブディレクトリは無ければ自動的に作られる. root 権限がある場合には, ‘/usr/local’ にインストールすることをお勧めする. 以下, このディレクトリを TARGETDIR と書く.

まず PARI ライブラリをインストールする必要がある. ‘pari.tgz’ を入手後, 適当なディレクトリで展開, インストールする.

```
% gzip -dc pari.tgz | tar xvf -
% cd pari
% ./Configure --prefix=TARGETDIR
% make all
% su
# make install
# make install-lib-sta
```

make 中にエラーで止まったら, 以下を実行する.

```
% cd Oxxx
% make lib-sta
% su
# make install-lib-sta
# make install-include
# exit
%
```

上の例で, xxx は現在ターゲットとなっている OS の名前を示す. GP はインストールされないが, asir2000 の作成に必要なファイルはインストールされる.

'asir2000.tgz' を入手後, 適当なディレクトリでで展開し, 以下の手順でインストールする.

```
% gzip -dc asir.tgz | tar xf -
% cd asir2000
% ./configure --prefix=TARGETDIR --with-pari --enable-plot
% make
% su
# make install
# make install-lib
# make install-doc
# exit
```

2.3.3 Windows version

必要なファイルは 'asirwin-ja.tgz' である. 他に, 'gzip.exe', 'tar.exe' が必要だが, 'asirwin-ja.tgz' と同じディレクトリに用意してある. これら 3 つのファイルを同一ディレクトリにおき, コマンドプロンプト (DOS プロンプト) から

```
C:\...> tar xzf asirwin.tgz
```

を実行すれば, 'Asir' というディレクトリ (Asir ルートディレクトリ) ができる. Asir ルートディレクトリのサブディレクト 'bin' に 'asirgui.exe' (GUI), 'engine.exe' (本体) が置かれている. 'asirgui.exe' をダブルクリックすれば Asir が立ち上がる.

2.4 コマンドラインオプション

コマンドラインオプションは次の通り.

`-heap` *number*
: Risa/Asir では, 4KB のブロックをメモリ割り当ての単位として用いている. デフォルトでは, 初期 heap として, 16 ブロック (64KB) 割り当てるが, それを変更する場合, `-heap` を用いる. 単位はブロックである. heap の大きさは, `heap()` 函数で調べることができる (単位はバイト).

`-adj` *number*
: この値が大きいほど, 使用メモリ量は大きくなるが, GC 時間が少なくなる. *number* として 1 以上の整数が指定できる. デフォルトでは 3 である. この値が 1 以下になると GC をしない設定になるので要注意である. heap をなるべく伸ばさずに, GC を主体にしてメモリ管理したい場合には, この値を大きく (例えば 8) 設定する.

`-norc`
: 初期化ファイル '$HOME/.asirrc' を読まない.

`-quiet`　起動時の著作権表示を行わない.

`-f` *file*　標準入力の代わりに, *file* から入力を読み込んで実行する. エラーの際にはただちに終了する.

`-paristack` *number*
PARI (6.1.14 節「pari」p.39 参照) 専用の領域の大きさを指定する. 単位はバイト. デフォルトでは 1 MB.

`-maxheap` *number*
heap 領域の上限を指定する. 単位はバイト. デフォルトでは無制限. UNIX の場合, 実際には `limit` コマンドで表示される `datasize` の値に制限されているため, `-maxheap` の指定がなくても一定量以上に heap を獲得できない場合があるので注意.)

2.5 環境変数

Asir の実行に関するいくつかの環境変数が存在する. UNIX 上では環境変数は shell のコマンドラインから直接設定するか, shell の rc ファイルで設定する. Windows NT では, [設定]->[システム]->[環境] で設定する. Windows 95/98 では, '`c:\autoexec.bat`' に書いて reboot する.

- `ASIR_LIBDIR`
  Asir のライブラリディレクトリ, すなわちユーザ言語で書かれたファイルなどがおかれるディレクトリ. 指定がない場合 UNIX 版では '`/usr/local/lib/asir`', Windows 版では Asir メインディレクトリの下の '`lib`' ディレクトリが用いられる. この環境変数は `ASIRLOADPATH` に統合され廃止される予定.
- `ASIR_CONTRIB_DIR`
  Asir の asir-contrib ディレクトリ, すなわち OpenXM/asir-contrib プロジェクトで書かれたパッケージやデータなどがおかれるディレクトリ. 指定がない場合 UNIX 版では '`/usr/local/lib/asir-contrib`', Windows 版では Asir メインディレクトリの下の '`lib-asir-contrib`' ディレクトリが用いられる. この環境変数は `ASIRLOADPATH` に統合され廃止される予定. This environmental variable will become obsolete.
- `ASIRLOADPATH`
  ロードされるファイルがあるディレクトリを UNIX の場合 ':', Windows の場合 ';' で区切って並べる. ディレクトリは左から順にサーチされる. この指定がない場合, および指定されたファイルが `ASIRLOADPATH` になかった場合, ライブラリディレクトリもサーチされる.
- `HOME`
  -norc オプションつきで起動しない場合, '`$HOME/.asirrc`'があれば, 予めこのファイルを実行する. `HOME` が設定されていない場合, UNIX 版ではなにも読まないが, Windows 版では Asir メインディレクトリ (`get_rootdir()` で返されるディレクトリ) の '`.asirrc`' を探し, あればそれを実行する.

2.6 起動から終了まで

Asir を起動すると,

```
[0]
```

なるプロンプトが表示され, セッションが開始する. '`$HOME/.asirrc`' (Windows 版の場合, `HOME` 設定されていない場合には `get_rootdir()` で返されるディレクトリにある '`.asirrc`')

が存在している場合, このファイルを **Asir** ユーザ言語でかかれたファイルと見なし, 解釈実行する.

プロンプトは入力の番号を表す. セッションは, `end;` または `quit;` を入力することにより終了する. 入力は, ';' または '$' までを一区切りとして評価される. ';' のとき結果は表示され, '$' のとき表示されない.

```
% asir
[0] A;
0
[1] A=(x+y)^5;
x^5+5*y*x^4+10*y^2*x^3+10*y^3*x^2+5*y^4*x+y^5
[2] A;
x^5+5*y*x^4+10*y^2*x^3+10*y^3*x^2+5*y^4*x+y^5
[3] a=(x+y)^5;
evalpv : invalid assignment
return to toplevel
[3] a;
a
[4] fctr(A);
[[1,1],[x+y,5]]
[5] quit;
%
```

この例では, `A`, `a`, `x`, `y` なる文字が使用されている. `A` はプログラムにおける変数で, `a`, `x`, `y` は数学的な意味での不定元である. 一般にプログラム変数は大文字で始まり, 不定元は小文字で始まる. この例でわかるように, プログラム変数は, 数, 式などを格納しておくためのものであり, C 言語などにおける変数に対応する. 一方, 不定元はそれ自身で値を持つことはできず, 従って, 不定元に対する代入は許されない. 後に示すが, 不定元に対する代入は, 組み込み函数 `subst()` により明示的に行われる.

2.7 割り込み

計算を実行中に割り込みをかけたい場合, 割り込みキャラクタ (通常は *C*-*c*, Windows, DOS 版では *C*-x を入力する.

```
@ (x+y)^1000;
C-cinterrupt ?(q/t/c/d/u/w/?)
```

各選択肢の意味は次の通り.

`q`　**Asir** を終了する. (確認あり)

`t`　トップレベルに戻る. (確認あり)

`c`　実行を継続する.

`d`　デバッグモードに入る. デバッガに関しては 第 5 章「デバッガ」p.29 を参照.

`u`　`register_handler()` (7.5.6 節「ox_reset ox_intr register_handler」p.99 参照) で登録された関数を実行後トップレベルに戻る. (確認あり)

`w`　中断点までの函数の呼び出し列を表示する.

`?`　各選択肢の意味を説明する.

2.8 エラー処理

組み込み函数に不正な型の引数を渡した場合などには実行が中断されるが, ユーザ函数の中でエラーが起きた場合にはトップレベルに戻る前に自動的にデバッグモードに入る. この状態でエラーの場所, 直前の引数の値などを調べることができる. 表示されるエラーメッセージはさまざまであり, 内部の函数名に引き続いてメッセージが表示される. これは, 呼び出された組み込み函数と必ずしも対応はしない.

その他, さまざまな原因により内部演算函数においてエラーが生ずることがある. UNIX 版の場合, これは次のいずれかの `internal error` として報告され, 通常のエラーと同様に扱って, デバッグモードに入る.

`SEGV`
`BUS ERROR`
: 組み込み函数によっては, 引数の型を厳密にチェックせずに演算ルーチンに引き渡してしまうものも存在している. このような状況において, 不正なポインタ, あるいは NULL ポインタによるアクセス違反があった場合, これらのエラーとなる.

`BROKEN PIPE`
: プロセス間通信において, 相手先のプロセスとの間のストリームが既に存在していない場合 (例えば既に相手先のプロセスが終了している場合など) に, そのストリームに入出力しようとした場合にこのエラーとなる.

これらは実際には, 組み込み函数の入口において, 引数を完全にチェックすることにより大部分は防げるが, 手間が多くかかることと, 場合によっては効率を落すことにもなるため, あえて引数チェックはユーザ任せにしてある.

2.9 計算結果, 特殊な数

`@` はエスケープ文字として使用される. 現在次のような規定がある.

`@`n　n 番目の計算結果.

`@@`　直前の計算結果.

`@i`　虚数単位.

`@pi`　円周率.

`@e`　自然対数の底.

`@`　2 元体 GF(2) 上の一変数多項式の変数 (不定元).

`@>, @<, @>=, @<=, @==, @&&, @||`
: quantifier elimination における, 一階述語論理演算子

```
[0] fctr(x^10-1);
[[1,1],[x-1,1],[x+1,1],[x^4+x^3+x^2+x+1,1],[x^4-x^3+x^2-x+1,1]]
[1] @@[3];
[x^4+x^3+x^2+x+1,1]
[2] eval(sin(@pi/2));
1.000000000000000000000000000000000000000000000000000000000000
[3] eval(log(@e),20);
0.99999999999999999999999999998
```

```
[4] @0[4][0];
x^4-x^3+x^2-x+1
[5] (1+@i)^5;
(-4-4*@i)
[6] eval(exp(@pi*@i));
-1.00000000000000000000000000000
[7] (@+1)^9;
(@^9+@^8+@+1)
```

トップレベルで計算された値はこのようにヒストリとして取り出し可能であるが, このことは, ガベージコレクタにとっては負担をもたらす可能性がある．特に, 大きな式をトップレベルで計算した場合, その後の GC 時間が急速に増大する可能性がある．このような場合, `delete_history()` (6.14.15 節「`delete_history`」p.89 参照) が有効である．

3 型

3.1 Asir で使用可能な型

Asir においては, 可読な形式で入力されたさまざまな対象は, パーザにより中間言語に変換され, インタプリタにより **Risa** の計算エンジンを呼び出しながら内部形式に変換される. 変換された対象は, 次のいずれかの型を持つ. 各番号は, 組み込み函数 `type()` により返される値に対応している. 各例は, **Asir** のプロンプトに対する入力が可能な形式のいくつかを示す.

0 **0**

実際には 0 を識別子にもつ対象は存在しない. 0 は, C における 0 ポインタにより表現されている. しかし, 便宜上 **Asir** の `type(0)` は値 0 を返す.

1 **数**

```
1 2/3  14.5  3+2*@i
```

数は, さらにいくつかの型に分けられる. これについては下で述べる.

2 **多項式**（数でない）

```
x  afo  (2.3*x+y)^10
```

多項式は, 全て展開され, その時点における変数順序に従って, 再帰的に 1 変数多項式として降冪の順に整理される. (8.1 節「分散表現多項式」p.109 を参照.) この時, その多項式に現れる順序最大の変数を **主変数** と呼ぶ.

3 **有理式**（多項式でない）

```
(x+1)/(y^2-y-x)  x/x
```

有理式は, 分母分子が約分可能でも, 明示的に `red()` が呼ばれない限り約分は行われない. これは, 多項式の GCD 演算が極めて重い演算であるためで, 有理式の演算は注意が必要である.

4 **リスト**

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

リストは読み出し専用である. `[]` は空リストを意味する. リストに対する操作としては, `car()`, `cdr()`, `cons()` などによる操作の他に, 読み出し専用の配列とみなして, [*index*] を必要なだけつけることにより要素の取り出しを行うことができる. 例えば

```
[0] L = [[1,2,3],[4,[5,6]],7]$
[1] L[1][1];
[5,6]
```

注意すべきことは, リスト, 配列 (行列, ベクトル) 共に, インデックスは 0 から始まることと, リストの要素の取り出しをインデックスで行うことは, 結局は先頭からポインタをたどることに相当するため, 配列に対する操作に比較して大きなリストでは時間がかかる場合があるということである.

5 **ベクトル**

```
newvect(3)  newvect(2,[a,1])
```

ベクトルは, `newvect()` で明示的に生成する必要がある. 前者の例では 2 成分の 0 ベクトルが生成され, 後者では, 第 0 成分が `a`, 第 1 成分が 1 のベクトルが生成される. 初期化のための 第 2 引数は, 第 1 引数以下の長さのリストを受け付ける.

リストの要素は左から用いられ, 足りない分は 0 が補われる. 成分は [*index*] により取り出せる. 実際には, 各成分に, ベクトル, 行列, リストを含む任意の型の対象を代入できるので, 多次元配列をベクトルで表現することができる.

```
[0] A3 = newvect(3);
[ 0 0 0 ]
[1] for (I=0;I<3;I++)A3[I] = newvect(3);
[2] for (I=0;I<3;I++)for(J=0;J<3;J++)A3[I][J]=newvect(3);
[3] A3;
[ [ [ 0 0 0 ] [ 0 0 0 ] [ 0 0 0 ] ]
[ [ 0 0 0 ] [ 0 0 0 ] [ 0 0 0 ] ]
[ [ 0 0 0 ] [ 0 0 0 ] [ 0 0 0 ] ] ]
[4] A3[0];
[ [ 0 0 0 ] [ 0 0 0 ] [ 0 0 0 ] ]
[5] A3[0][0];
[ 0 0 0 ]
```

6 行列

```
newmat(2,2)  newmat(2,3,[[x,y],[z]])
```

行列の生成も `newmat()` により明示的に行われる. 初期化も, 引数がリストのリストとなることを除いてはベクトルと同様で, リストの各要素 (これはまたリストである) は, 各行の初期化に使われ, 足りない部分には 0 が埋められる. 行列も, 各要素には任意の対象を代入できる. 行列の各行は, ベクトルとして取り出すことができる.

```
[0] M=newmat(2,3);
[ 0 0 0 ]
[ 0 0 0 ]
[1] M[1];
[ 0 0 0 ]
[2] type(@@);
5
```

7 文字列

```
""  "afo"
```

文字列は, 主にファイル名などに用いられる. 文字列に対しては加算のみが定義されていて, 結果は 2 つの文字列の結合である.

```
[0] "afo"+"take";
afotake
```

8 構造体

```
newstruct(afo)
```

Asir における構造体は, C における構造体を簡易化したものである. 固定長配列の各成分を名前でアクセスできるオブジェクトで, 構造体定義毎に名前をつける.

9 分散表現多項式

```
2*<<0,1,2,3>>-3*<<1,2,3,4>>
```

これは, ほとんどグレブナ基底専用の型で, 通常の計算でこの型が必要となることはまずないが, グレブナ基底計算パッケージ自体がユーザ言語で書かれているため, ユーザが操作できるよう独立した型として **Asir** で使用できるようにしてある. これについては 第 8 章「グレブナ基底の計算」p.109 を参照.

10 **符号なしマシン 32bit 整数**

11 **エラーオブジェクト**

以上二つは, Open XM において用いられる特殊オブジェクトである.

12 **GF(2) 上の行列**

現在, 標数 2 の有限体における基底変換のためのオブジェクトとして用いられる.

13 **MATHCAP オブジェクト**

Open XM において, 実装されている機能を送受信するためのオブジェクトである.

14 **first order formula**

quantifier elimination で用いられる一階述語論理式.

15 **matrix over GF**(p)

小標数有限体上の行列.

16 **byte array**

符号なし byte の配列

-1 **VOID オブジェクト**

型識別子 -1 をもつオブジェクトは関数の戻り値などが無効であることを示す.

3.2 数の型

0 **有理数**
有理数は, 任意多倍長整数 (**bignum**) により実現されている. 有理数は常に既約分数で表現される.

1 **倍精度浮動小数**
マシンの提供する倍精度浮動小数である. **Asir** の起動時には, 通常の形式で入力された浮動小数はこの型に変換される. ただし, `ctrl()` により **bigfloat** が選択されている場合には **bigfloat** に変換される.

```
[0] 1.2;
1.2
[1] 1.2e-1000;
0
[2] ctrl("bigfloat",1);
1
[3] 1.2e-1000;
1.20000000000000000513 E-1000
```

倍精度浮動小数と有理数の演算は, 有理数が浮動小数に変換されて, 浮動小数として演算される.

2 **代数的数**
第 9 章「代数的数に関する演算」p.140 を参照.

3 **bigfloat**
bigfloat は, **Asir** では **PARI** ライブラリにより実現されている. **PARI** においては, **bigfloat** は, 仮数部のみ任意多倍長で, 指数部は 1 ワード以内の整数に限られている. `ctrl()` で **bigfloat** を選択することにより, 以後の浮動小数の入力は **bigfloat** として扱われる. 精度はデフォルトでは 10 進 9 桁程度であるが, `setprec()` により指定可能である.

```
[0] ctrl("bigfloat",1);
1
[1] eval(2^(1/2));
1.41421356237309504876378807303 1
[2] setprec(100);
9
[3] eval(2^(1/2));
1.41421356237309504880168872420969807856967187537694807317...
```

`eval()` は, 引数に含まれる函数値を可能な限り数値化する函数である. `setprec()` で指定された桁数は, 結果の精度を保証するものではなく, **PARI** 内部で用いられる表現のサイズを示すことに注意すべきである. (⟨undefined⟩「eval deval」p.⟨undefined⟩ を参照, 6.1.14 節「pari」p.39.)

4 **複素数**
複素数は, 有理数, 倍精度浮動小数, **bigfloat** を実部, 虚部として `a+b*@i` (`@i` は虚数単位) として与えられる数である. 実部, 虚部はそれぞれ `real()`, `imag()` で取り出せる.

5 **小標数の有限素体の元**
ここで言う小標数とは, 標数が 2^27 未満のもののことである. このような有限体は, 現在のところグレブナ基底計算において内部的に用いられ, 有限体係数の分散表現多項式の係数を取り出すことで得られる. それ自身は属する有限体に関する情報は持たず, `setmod()` で設定されている素数 p を用いて GF(p) 上での演算が適用される.

6 **大標数の有限素体の元**
標数として任意の素数がとれる. この型の数は, 整数に対し `simp_ff` を適用することにより得られる.

7 **標数 2 の有限体の元**
標数 2 の任意の有限体の元を表現する. 標数 2 の有限体 F は, 拡大次数 [F:GF(2)] を n とすれば, GF(2) 上既約な n 次多項式 f(t) により F=GF(2)[t]/(f(t)) とあらわされる. さらに, GF(2)[t] の元 g は, f(t) も含めて自然な仕方でビット列とみなされるため, 形式上は, F の元 g mod f は, g, f をあらわす 2 つのビット列で表現することができる.

F の元を入力するいくつかの方法が用意されている.

- `@`
  `@` はその後ろに数字, 文字を伴って, ヒストリや特殊な数をあらわすが, 単独で現れた場合には, F=GF(2)[t]/(f(t)) における t mod f をあらわす. よって, `@` の多項式として F の元を入力できる. (`@^10+@+1` など)

- ptogf2n
  任意変数の 1 変数多項式を, ptogf2n により対応する F の元に変換する.
- ntogf2n
  任意の自然数を, 自然な仕方で F の元とみなす. 自然数としては, 10 進, 16 進 (0x で始まる), 2 進 (0b で始まる) で入力が可能である.
- その他
  多項式の係数を丸ごと F の元に変換するような場合, simp_ff により変換できる.

8　**位数 $p\hat{}n$ の有限体の元**

位数が $p\hat{}n$ (p は任意の素数, n は正整数) は, 標数 p および GF(p) 上既約な n 次多項式 m(x) を setmod_ff により指定することにより設定する. この体の元は m(x) を法とする GF(p) 上の多項式として表現される.

9　**位数 $p\hat{}n$ の有限体の元 (小位数)**

位数が $p\hat{}n$ の有限体 ($p\hat{}n$ が *2^29* 以下, p が *2^14* 以上なら n は 1) は, 標数 p および拡大次数 n を setmod_ff により指定することにより設定する. この体の 0 でない元は, p が *2^14* 未満の場合, GF($p\hat{}n$) の乗法群の生成元を固定することにより, この元のべきとして表される. これにより, この体の 0 でない元は, このべき指数として表現される. p が *2^14* 以上の場合は通常の剰余による表現となるが, 共通のプログラムで双方の場合を扱えるようにこのような仕様となっている.

小標数有限素体以外の有限体は setmod_ff で設定する. 有限体の元どうしの演算では, 一方が有理数の場合には, その有理数は自動的に現在設定されている有限体の元に変換され, 演算が行われる.

3.3 不定元の型

多項式の変数となり得る対象を**不定元**とよぶ. **Asir** では, 英小文字で始まり, 任意個のアルファベット, 数字, '_' からなる文字列を不定元として扱うが, その他にもシステムにより不定元として扱われるものがいくつかある. **Asir** の内部形式としては, これらは全て多項式としての型を持つが, 数と同様, 不定元の型により区別される.

0　**一般不定元**

英小文字で始まる文字列. 多項式の変数として最も普通に用いられる.

```
[0] [vtype(a),vtype(aA_12)];
[0,0]
```

1　**未定係数**

uc() は, '_' で始まる文字列を名前とする不定元を生成する. これらは, ユーザが入力できないというだけで, 一般不定元と変わらないが, ユーザが入力した不定元と衝突しないという性質を利用して未定係数の自動生成などに用いることができる.

```
[1] U=uc();
_0
[2] vtype(U);
1
```

2 **函数形式**

組み込み函数, ユーザ函数の呼び出しは, 評価されて何らかの **Asir** の内部形式に変換されるが, `sin(x)`, `cos(x+1)` などは, 評価後もそのままの形で存在する. これは函数形式と呼ばれ, それ自身が 1 つの不定元として扱われる. またやや特殊な例として, 円周率 `@pi` や自然対数の底 `@e` も函数形式として扱われる.

```
[3] V=sin(x);
sin(x)
[4] vtype(V);
2
[5] vars(V^2+V+1);
[sin(x)]
```

3 **函数子**

函数呼び出しは, *fname*(*args*) という形で行なわれるが, *fname* の部分を函数子と呼ぶ. 函数子には, 函数の種類により組み込み函数子, ユーザ定義函数子, 初等函数子などがあるが, 函数子は単独で不定元として機能する.

```
[6] vtype(sin);
3
```

4 ユーザ言語 Asir

Asir の組み込み函数は, 因数分解, GCD などの計算を行うもの, ファイル入出力を行うもの, あるいは数式の一部を取り出すものなどさまざまなものが用意されているが, ユーザが実際に行いたいことを実行させるためには一般にはユーザ言語によるプログラムを書く必要がある. ユーザ言語も **Asir** と呼ばれる. 以下では, ユーザ言語の文法規則および実際のユーザ言語プログラムを例としたプログラムの書き方について述べる.

4.1 文法 (C 言語との違い)

Asir の文法は C 言語に準拠している. おもな相違点は次の通りである. 以下で, 変数とは **Asir** におけるプログラム用の変数, すなわち大文字で始まる文字列を意味することとする.

- 変数の型がない.
  既に説明したとおり, **Asir** で扱われる対象自身は全て何らかの型を持っている. しかし, プログラム変数自体は, どのような対象でも代入できるという意味で型がないのである.

```
[0] A = 1;
1
[1] type(A);
1
[2] A = [1,2,3];
[1,2,3]
[3] type(A);
4
```

- 函数内の変数は, デフォルトでは仮引数をこめてすべて局所変数.
  ただし, `extern` 宣言された変数は, トップレベルにおける大域変数となる. すなわち, 変数のスコープは大域変数と局所変数の 2 種類に単純化されている. トップレベル, すなわちプロンプトに対して入力された変数は全て大域変数として登録される. また函数内では次のいずれかとなる.
  1. 函数が定義されるファイルにおいて, その函数定義以前に, ある変数が `extern` 宣言されている場合, 函数内のその変数も大域変数として扱われる.
  2. `extern` 宣言されていない変数はその函数に局所的となる.

```
% cat afo
def afo() { return A;}
extern A$
def bfo() { return A;}
end$
% asir
[0] load("afo")$
[5] A = 1;
1
[6] afo();
0
[7] bfo();
1
```

- プログラム変数は大文字で始まり, 不定元, 函数は小文字で始まる.
  この点は, 既存の数式処理システムのほとんどと異なる点である. **Asir** がこの仕様を採

用したのは, ユーザが不定元のつもりで使用した変数になんらかの値が代入されていた場合に混乱を招く, という, 既存のシステムにありがちな状況を避けるためである.

- `switch` 文, `goto` がない.
  `goto` がないため, 多重ループを一度に抜けるのがやや複雑になる場合がある.
- コンマ式は, `for (A;B;C)` または, `while(A)` の `A`, `B`, `C` にのみ使うことができる.
  これは, リストを正式なオブジェクトとして加えたことによる.

以上は制限であるが, 拡張としては次の点が挙げられる.

- 有理式に対する計算を, 通常の C における計算と同様にできる.
- リストが扱える.
  構造体を用いるまでもない要素の集合体を, リストで表すことができ, C で直接書く場合に比較してプログラムが短く, 読みやすく書ける.
- ユーザ定義函数における一行ヘルプ. Emacs-Lisp に類似した機能である. 詳しくは, 4.2.1 節「ユーザ定義函数」p.17 を参照 を見よ.
- ユーザ定義函数におけるオプション指定.
  これに関しては, 4.2.12 節「オプション指定」p.24 を参照.

Asir では次の語句がキーワードとして定められている.

C 言語に由来:

`break`, `continue`, `do`, `else`, `extern`, `for`, `if`, `return`, `static`, `struct`, `while`

C 言語からの拡張:

`def`, `endmodule`, `function`, `global`, `local`, `localf`, `module`

函数:

`car`, `cdr`, `getopt`, `newstruct`, `map`, `pari`, `quote`, `recmap`, `timer`

4.2 ユーザ定義函数の書き方

4.2.1 ユーザ定義函数

ユーザによる函数の定義は ‘`def`’ 文で行う. 文法エラーは読み込み時にある程度チェックされ, おおよその場所が表示される. 既に (引数の個数に関係なく) 同名の函数が定義されている場合には, その函数は再定義される. `ctrl()` 函数により `verbose` フラグが on になっている場合,

```
afo() redefined.
```

というメッセージが表示される. ある函数の定義において, まだ未定義の函数を呼び出していても, 定義時にはエラーにならない. 実行時に未定義の函数を呼び出そうとした場合にエラーとなる.

```
/* X! */

def f(X) {
    if ( !X )
        return 1;
    else
```

```
        return X * f(X-1);
}


/* iCj ( 0 ≤ i ≤ N, 0 ≤ j ≤ i ) */

def c(N)
{
    A = newvect(N+1); A[0] = B = newvect(1); B[0] = 1;
    for ( K = 1; K <= N; K++ ) {
        A[K] = B = newvect(K+1); B[0] = B[K] = 1;
        for ( P = A[K-1], J = 1; J < K; J++ )
            B[J] = P[J-1]+P[J];
        }
    return A;
}


/* A + B */

def add(A,B)
"add two numbers."
{
    return A+B;
}
```

2 つ目の例では, 長さ N+1 のベクトル (Aとする) が返される. A[I] は長さ I+1 の配列であり, そのそれぞれの要素が ${}_IC_J$ を要素とする配列である.

3 つ目の例では, 引数並びのあとに文字列が置かれているが、これは Emacs-Lisp の関数定義に類似の機能で、ヘルプ用の文字列である。この例の場合、help(add) によってこの文字列が出力される。

参照　6.14.4 節「help」p.84.

以下では, C によるプログラミングの経験がない人のために, **Asir** 言語によるプログラムの書き方を解説する.

4.2.2 変数および不定元

既に述べた通り, **Asir** においてはプログラム変数と不定元を明確に区別している.

変数　大文字で始まり, アルファベット, 数字, '_' からなる文字列

変数あるいはプログラム変数とは, **Asir** のさまざまな型の内部形式を格納するための箱であり, 格納された内部形式が, この変数の値である. 変数が式の要素として評価される時は, そこに収められた値に置き換えられる. すなわち, 内部形式の中にはプログラム変数は現れない. 変数は全て 0 で初期化されている.

```
[0] X^2+X+1;
1
[1] X=2;
2
```

```
[2] X^2+X+1;
7
```

不定元　小文字で始まり, アルファベット, 数字, '_' からなる文字列, またはシングルクオートで囲まれた文字列, もしくは函数形式. 不定元とは, 多項式環を構成する際に添加される変数をいう. **Asir** においては, 不定元は値をもたない超越的な元であり, 不定元への値の代入は許されない.

```
[3] X=x;
x
[4] X^2+X+1;
x^2+x+1
[5] A='Dx'*(x-1)+x*y-y;
(y+Dx)*x-y-Dx
[6] function foo(x,y);
[7] B=foo(x,y)*x^2-1;
foo(x,y)*x^2-1
```

4.2.3 引数

```
def sum(N) {
    for ( I = 1, S = 0; I <= N; I++ )
        S += I;
    return S;
}
```

これは, 1 から N までの自然数の和を求める函数 sum() の定義である. この例における sum(N) の N が引数である. この例は, 1 引数函数の例であるが, 一般に引数の個数は任意であり, 必要なだけの個数を ',' で区切って指定することができる. 引数は値が渡される. すなわち, 引数を受けとった側が, その引数の値を変更しても, 渡した側の変数は変化しない. ただし, 例外がある. それは, ベクトル, 行列を引数に渡した場合である. この場合も, 渡された変数そのものを書き替えることは, その函数に局所的な操作であるが, 要素を書き換えた場合, それは, 呼び出し側のベクトル, 行列の要素を書き換えることになる.

```
def clear_vector(M) {
    /* M is expected to be a vector */
    L = size(M)[0];
    for ( I = 0; I < L; I++ )
        M[I] = 0;
}
```

この函数は, 引数のベクトルを 0 ベクトルに初期化するための函数である. また, ベクトルを引数に渡すことにより, 複数の結果を引数のベクトルに収納して返すことができる. 実際には, このような場合には, 結果をリストにして返すこともできる. 状況に応じて使いわけすることが望ましい.

4.2.4 コメント

C と同様 '/*' と '*/' で囲まれた部分はコメントとして扱われる.

```
/*
 * This is a comment.
 */
```

```
def afo(X) {
```

コメントは複数行に渡っても構わないが, 入れ子にすることはできない. '/*' がいくつあっても最初のもののみが有効となり, 最初に現れた '*/' でコメントは終了したと見なされる. プログラムなどで, コメントを含む可能性がある部分をコメントアウトした場合には, #if 0, #endifを使えばよい. (4.2.11 節「プリプロセッサ」p.23 を参照.)

```
#if 0
def bfo(X) {
/* empty */
}
#endif
```

4.2.5 文

Asir のユーザ函数は,

```
def 名前 (引数, 引数, ..., 引数) {
    文
    文
    ...
    文
}
```

という形で定義される. このように, 文は函数の基本的構成要素であり, プログラムを書くためには, 文がどのようなものであるか知らなければならない. 最も単純な文として, 単文がある. これは,

```
S = sum(N);
```

のように, 式に終端記号 (';' または '$') をつけたものである. この単文及び類似の return 文, break 文などが文の最小構成単位となる. if 文や for 文の定義 (A.1 節「文法の詳細」p.162) を見ればわかる通り, それらの本体は, 単なる一つの文として定義されている. 通常は, 本体には複数の文が書けることが必要となる. このような場合, '{' と '}' で文の並びを括って, 一つの文として扱うことができる. これを複文と呼ぶ.

```
if ( I == 0 ) {
    J = 1;
    K = 2;
    L = 3;
}
```

'}' の後ろには終端記号は必要ない. なぜなら, '{' 文並び '}'が既に文となっていて, if 文の要請を満たしているからである.

4.2.6 return 文

return 文は,

```
return 式;

return;
```

の 2 つの形式がある. いずれも函数から抜けるための文である. 前者は函数の値として 式 を返す. 後者では, 函数の値として何が返されるかはわからない.

4.2.7 if 文

if 文には

```
if ( 式 )              if ( 式 )
    文       及び          文
else
    文
```

の 2 種類がある. これらの動作は明らかであるが, 文の位置に if 文が来た場合に注意を要する. 次の例を考えてみよう.

```
if ( 式 )
    if ( 式 ) 文
else
    文
```

この場合, 字下げからは, else 以下は, 最初の if に対応するように見えるが, パーザは, 自動的に 2 番目の if に対応すると判断する. すなわち, 2 種類の if 文を許したために, 文法に曖昧性が現れ, それを解消するために, else 以下は, 最も近い if に対応するという規則が適用されるのである. 従って, この例は,

```
if ( 式 ) {
    if ( 式 ) 文 else 文
}
```

という意味となる. 字下げに対応させるためには,

```
if ( 式 ) {
    if ( 式 ) 文
} else
    文
```

としなければならない.

関数の中でなく, top level で if 文を用いるときは $ または ; で終了する必要がある. これらがないと次の文がよみとばされる.

4.2.8 ループ, break, return, continue

ループを構成する文は, while 文, for 文, do 文の 3 種類がある.

- while 文
  形式は,

  ```
  while ( 式 ) 文
  ```

  で, これは, 式 を評価して, その値が 0 でない限り 文 を実行するという意味となる. たとえば 式 が 1 ならば, 単純な無限ループとなる.

- for 文
  形式は,

  ```
  for ( 式並び-1; 式; 式並び-2 ) 文
  ```

  で, これは

  ```
  式並び-1 (を単文並びにしたもの)
  while ( 式 ) {
      文
      式並び-2 (を単文並びにしたもの)
  ```

```
    }
```

と等価である.

- do 文

```
do {
    文
} while ( 式 )
```

は, 先に 文を実行してから条件式による判定を行う所が while 文と異なっている.

ループを抜け出す手段として, break 文及び return 文がある. また, ループの制御をある位置に移す手段として continue 文がある.

- break
  break 文は, それを囲むループを一つだけ抜ける.
- return
  return 文は, 一般に函数から抜けるための文であり, ループの中からでも有効である.
- continue
  continue 文は, ループの本体の文の末端に制御を移す. 例えば for 文では, 最後の式並びの実行を行い, while 文では条件式の判定に移る.

4.2.9 構造体定義

構造体とは, 各成分の要素が名前でアクセスできる固定長配列と思ってよい. 各構造体は名前で区別される. 構造体は, struct 文により宣言される. 構造体が宣言されるとき, asir は内部で構造体のそれぞれの型に固有の識別番号をつける. この番号は, 組み込み関数 struct_type により取得できる. ある型の構造体は, 組み込み関数 newstruct により生成される. 構造体の各メンバは, 演算子 -> によりアクセスする. メンバが構造体の場合, -> による指定は入れ子にできる.

```
[1] struct rat {num,denom};
0
[2] A = newstruct(rat);
{0,0}
[3] A->num = 1;
1
[4] A->den = 2;
2
[5] A;
{1,2}
[6] struct_type(A);
1
```

参照　6.7.1 節「newstruct」p.65, 6.7.3 節「struct_type」p.67

4.2.10 さまざまな式

主な式の構成要素としては, 次のようなものがある.

- 加減乗除, 冪
  冪は, '^' により表す. 除算 '/' は, 体としての演算に用いる. 例えば, 2/3 は有理数の

2/3 を表す. 整数除算, 多項式除算 (剰余を含む演算) には別途組み込み函数が用意されている.

```
x+1  A^2*B*afo X/3
```

- インデックスつきの変数
  ベクトル, 行列, リストの要素はインデックスを用いることにより取り出せる. インデックスは 0 から始まることに注意する. 取り出した要素がベクトル, 行列, リストなら, さらにインデックスをつけることも有効である.

  ```
  V[0] M[1][2]
  ```

- 比較演算
  等しい (‘==’), 等しくない (‘!=’), 大小 (‘>’, ‘<’, ‘>=’, ‘<=’) の 2 項演算がある. 真ならば有理数の 1, 偽ならば 0 を値に持つ.
- 論理式
  論理積 (‘&&’), 論理和 (‘||’) の 2 項演算と, 否定 (‘!’) が用意されている. 値はやはり 1, 0 である.
- 代入
  通常の代入は ‘=’ で行う. このほか, 算術演算子と組み合わせて特殊な代入を行うこともできる. (‘+=’, ‘-=’, ‘*=’, ‘/=’, ‘^=’)

  ```
  A = 2  A *= 3 (これは A = A*3 と同じ; その他の演算子も同様)
  ```

- 函数呼び出し
  函数呼び出しも式の一種である.
- ‘++’, ‘--’
  これらは, 変数の前後について, それぞれ次のような操作, 値を表す.

  ```
  A++  値は元の A の値, A = A+1
  A--  値は元の A の値, A = A-1
  ++A  A = A+1, 値は変化後の値
  --A  A = A-1, 値は変化後の値
  ```

4.2.11 プリプロセッサ

Asir のユーザ言語は C 言語を模したものである. C の特徴として, プリプロセッサ cpp によるマクロ展開, ファイルのインクルードがあるが, **Asir** においてもユーザ言語ファイルの読み込みの際 cpp を通してから読み込むこととした. これによりユーザ言語ファイル中で #include, #define, #if などが使える.

- #include
  UNIX では インクルードファイルは, Asir のライブラリディレクトリ (環境変数 **ASIR_LIBDIR** で指定されたディレクトリ) と #include が書かれているファイルと同じディレクトリをサーチする. UNIX 以外では cpp に特に引数を渡さないため, #include が書かれているファイルと同じディレクトリのみをサーチする.
- #define
  これは, C におけるのと全く同様に用いることができる.
- #if
  /*, */ によるコメントは入れ子にできないので, プログラムの大きな部分をコメントアウトする際に, #if 0, #endif を使うと便利である.

次の例は, ‘defs.h’ にあるマクロ定義である.

```
#define ZERO 0
#define NUM 1
#define POLY 2
#define RAT 3
#define LIST 4
#define VECT 5
#define MAT 6
#define STR 7
#define N_Q 0
#define N_R 1
#define N_A 2
#define N_B 3
#define N_C 4
#define V_IND 0
#define V_UC 1
#define V_PF 2
#define V_SR 3
#define isnum(a) (type(a)==NUM)
#define ispoly(a) (type(a)==POLY)
#define israt(a) (type(a)==RAT)
#define islist(a) (type(a)==LIST)
#define isvect(a) (type(a)==VECT)
#define ismat(a) (type(a)==MAT)
#define isstr(a) (type(a)==STR)
#define FIRST(L) (car(L))
#define SECOND(L) (car(cdr(L)))
#define THIRD(L) (car(cdr(cdr(L))))
#define FOURTH(L) (car(cdr(cdr(cdr(L)))))
#define DEG(a) deg(a,var(a))
#define LCOEF(a) coef(a,deg(a,var(a)))
#define LTERM(a) coef(a,deg(a,var(a)))*var(a)^deg(a,var(a))
#define TT(a) car(car(a))
#define TS(a) car(cdr(car(a)))
#define MAX(a,b) ((a)>(b)?(a):(b))
```

C のプリプロセッサを流用しているため, プリプロセッサは $ を正しく処理できない. たとえば LIST が定義されていても LIST$は置換されない. $ の前に空白をおいて LIST $ と書かないといけない.

4.2.12 オプション指定

ユーザ定義関数が N 変数で宣言された場合, その関数は, N 変数での呼び出しのみが許される.

```
[0] def factor(A) { return fctr(A); }
[1] factor(x^5-1,3);
evalf : argument mismatch in factor()
return to toplevel
```

不定個引数の関数をユーザ言語で記述したい場合, リスト, 配列を用いることで可能となるが, 次のようなより分かりやすい方法も可能である.

```
% cat factor
def factor(F)
{
    Mod = getopt(mod);
    ModType = type(Mod);
    if ( ModType == 1 ) /* 'mod' is not specified. */
        return fctr(F);
    else if ( ModType == 0 ) /* 'mod' is a number */
        return modfctr(F,Mod);
}
[0] load("factor")$
[1] factor(x^5-1);
[[1,1],[x-1,1],[x^4+x^3+x^2+x+1,1]]
[2] factor(x^5-1|mod=11);
[[1,1],[x+6,1],[x+2,1],[x+10,1],[x+7,1],[x+8,1]]
```

2 番目の factor() の呼び出しにおいて, 関数定義の際に宣言された引数 x^5-1の後ろに |mod=11 が置かれている. これは, 関数実行時に, mod という keyword に対して 11 という値を割り当てることを指定している. これをオプション指定と呼ぶことにする. この値は getopt(mod) で取り出すことができる. 1 番目の呼び出しのように mod に対するオプション指定がない場合には, getopt(mod) は型識別子 -1 のオブジェクトを返す. これにより, 指定がない場合の動作を if 文により記述できる. '|' の後ろには, 任意個のオプションを, ',' で区切って指定することができる.

```
[100] xxx(1,2,x^2-1,[1,2,3]|proc=1,index=5);
```

さらに, オプションを key1=value1,key2=value2,... のように ',' で区切って渡す代わりに, 特別なキーワード option_list とオプションリスト [["key1",value1],["key2",value2],...] を用いて渡すことも可能である.

```
[101] dp_gr_main([x^2+y^2-1,x*y-1]|option_list=[["v",[x,y]],["order",[[x,5,y,1]]]])
```

特に, 引数なしの getopt() はオプションリストを返すので, オプションをとる関数から, オプションをとる関数を呼び出すときには有用である.

```
% cat foo.rr
def foo(F)
{
    OPTS=getopt();
    return factor(F|option_list=OPTS);
}
[3] load("foo.rr")$
[4] foo(x^5-1|mod=11);
[[1,1],[x+6,1],[x+2,1],[x+10,1],[x+7,1],[x+8,1]]
```

4.2.13 モジュール

ライブラリで定義されている関数, 変数をカプセル化する仕組みがモジュール (module) である. はじめにモジュールを用いたプログラムの例をあげよう.

```
module stack;

static Sp $
Sp = 0$
```

```
static Ssize$
Ssize = 100$
static Stack $
Stack = newvect(Ssize)$
localf push $
localf pop $

def push(A) {
  if (Sp >= Ssize) {print("Warning: Stack overflow\nDiscard the top"); pop();}
  Stack[Sp] = A;
  Sp++;
}
def pop() {
  local A;
  if (Sp <= 0) {print("Stack underflow"); return 0;}
  Sp--;
  A = Stack[Sp];
  return A;
}
endmodule;

def demo() {
  stack.push(1);
  stack.push(2);
  print(stack.pop());
  print(stack.pop());
}
```

モジュールは module モジュール名 〜 endmoduleで囲む．モジュールは入れ子にはできない．モジュールの中だけで使う大域変数は static で宣言する．この変数はモジュールの外からは参照もできないし変更もできない．モジュールの外の大域変数は extern で宣言する．

モジュール内部で定義する関数は localf を用いて宣言しないといけない．上の例では push と pop を宣言している．この宣言は必須である．

モジュール moduleName で定義された関数 functionName をモジュールの外から呼ぶには moduleName.functionName(引数 1, 引数 2, ...) なる形式でよぶ．モジュールの中からは, 関数名のみでよい．次の例では, モジュールの外からモジュール stack で定義された関数 push, pop を呼んでいる．

```
stack.push(2);
print( stack.pop() );
2
```

モジュールで用いる関数名は局所的である．つまりモジュールの外や別のモジュールで定義されている関数名と同じ名前が利用できる．

モジュール機能は大規模ライブラリの開発を想定している．ライブラリを必要に応じて分割ロードするには, 関数 module_definedp を用いるのが便利である．デマンドロードはたとえば次のように行なえば良い．

```
if (!module_definep("stack")) load("stack.rr") $
```

asir では局所変数の宣言は不要であった．しかしモジュール stack の例を見れば分かるように, local A; なる形式で局所変数を宣言できる．キーワード local を用いると, 宣言機能

が有効となる. 宣言機能を有効にすると, 宣言されてない変数はロードの段階でエラーを起こす. 変数名のタイプミスによる予期しないトラブルを防ぐには, 宣言機能を有効にしてプログラムするのがよい.

モジュール内の関数をそのモジュールが定義される前に呼び出すような関数を書くときには, その関数の前でモジュールを次のようにプロトタイプ宣言しておく必要がある.

```
/* Prototype declaration of the module stack */
module stack;
localf push $
localf pop $
endmodule;

def demo() {
  print("----------------");
  stack.push(1);
  print(stack.pop());
  print("---------------");
}

module stack;
  /* The body of the module stack */
endmodule;
```

モジュールの中からトップレベルで定義されている関数を呼ぶには, 下の例のように :: を用いる.

```
def afo() {
  S = "afo, afo";
  return S;
}
module abc;
localf foo,afo $

def foo() {
  G = ::afo();
  return G;
}
def afo() {
  return "afo, afo in abc";
}
endmodule;
end$

[1200] abc.foo();
afo, afo
[1201] abc.afo();
afo, afo in abc
```

参照　6.12.1 節「module_list」p.79, 6.12.2 節「module_definedp」p.79, 6.12.3 節「remove_module」p.80.

5 デバッガ

5.1 デバッガとは

C 言語で書かれたプログラムのためのデバッガ dbx は, ソースレベルでのブレークポイントの設定, ステップ実行, 変数の参照などが可能な強力なデバッガである. **Asir** では, dbx 風のデバッガを用意している. デバッグモードに入るには, トップレベルで debug; と入力する.

```
[10] debug;
(debug)
```

その他, 次の方法, あるいは状況でデバッグモードに入る.

- 実行中ブレークポイントに達した場合
- 割り込みで 'd' を選択した場合
- 実行中エラーを起こした場合

  この場合, 実行の継続は不可能であるが, 直接のエラーの原因となったユーザ定義函数の文を表示してデバッグモードに入るため, エラー時における変数の値を参照でき, デバッグに役立たせることができる.
- error() が呼び出された場合

5.2 コマンドの解説

コマンドは dbx のコマンドの内必要最小限のものを採用した. 更に, gdb のコマンドからもいくつか便利なものを採用した. 実際の機能は dbx とほぼ同様であるが, step, next は, 次の行ではなく次の文を実行する. 従って, 1 行に複数の文がある場合は, その文の数だけ next を実行しなければ次の行に進めない. また, dbx と同様 '.dbxinit' を読み込むので, dbx と同じ alias を使うことができる.

step　次の文を実行する. 次の文が函数を含むとき, その函数に入る.

next　次の文を実行する.

finish　現在実行中の函数の実行が終了した時点で再びデバッグモードに入る. 誤って step を実行した場合に有効である.

cont
quit　デバッグモードから抜け, 実行を継続する.

up [*n*]　スタックフレームを 1 段 (引数 *n* がある時は *n* 段) 上がる. これにより, そのスタックフレームに属する変数の値の参照, 変更ができる.

down [*n*]　スタックフレームを 1 段 (引数 *n* がある時は *n* 段) 下がる.

frame [*n*]　引数がないとき, 現在実行中の函数を表示する. 引数があるとき, スタックフレームを番号 *n* のものに設定する. ここでスタックフレームの番号とは where により表示される呼び出し列において, 先頭に表示される番号のことである.

list [*startline*]
list *function*
　現在行, または *startline*, または *function* の先頭から 10 行ソースファイルを表示する.

`print` *expr*
: *expr* を表示する.

`func` *function*
: 対象函数を *function* に設定する.

`stop at` *sourceline* [`if` *cond*]
`stop in` *function*
: *sourceline* 行目, または *function* の先頭にブレークポイントを設定する. ブレークポイントは, 函数が再定義された場合自動的に取り消される. `if` が続く場合, *cond* が評価され, それが 0 でない場合に実行が中断し, デバッグモードに入る.

`trace` *expr* `at` *sourceline* [`if` *cond*]

`trace` *expr* `in` *function*
: `stop` と同様であるが, `trace` では単に *expr* を表示するのみで, デバッグモードには入らない.

`delete` n　ブレークポイント n を取り消す.

`status`　ブレークポイントの一覧を表示する.

`where`　現在の停止点までの呼び出し列を表示する.

`alias` *alias command*
: *command* に *alias* の別名を与える.

`print` の引数として, トップレベルにおけるほとんどすべての式がとれる. 通常は, 変数の内容の表示が主であるが, 必要に応じて次のような使い方ができる.

- **変数の書き換え**

  実行中のブレークポイントにおいて, 変数の値を変更して実行を継続させたい場合, 次のような操作を行えばよい.

```
(debug) print A
A = 2
(debug) print A=1
A=1 = 1
(debug) print A
A = 1
```

- **函数の呼び出し**

  函数呼び出しも式であるから, `print` の引数としてとれる.

```
(debug) print length(List)
length(List) = 14
```

  この例では, 変数 `List` に格納されているリストの長さを `length()` により調べている.

```
(debug) print ctrl("cputime",1)
ctrl("cputime",1) = 1
```

  この例は, 計算開始時に CPU 時間の表示の指定をし忘れた場合などに, 計算途中でデバッグモードから指定を行えることを示している.

  また, 止むを得ず計算を中断しなければならない場合, デバッグモードから `bsave()` などのコマンドにより途中結果をファイルに保存することもできる.

```
(debug) print bsave(A,"savefile")
bsave(A,"savefile") = 1
```

デバッグモードからの函数呼び出しで注意すべきことは, print の引数がユーザ定義函数の呼び出しを含む場合, その函数呼び出しでエラーが起こった場合に元の函数の実行継続が不可能になる場合があるということである.

5.3 デバッガの使用例

ここでは, 階乗を再帰的に計算させるユーザ定義函数を例として, デバッガの実際の使用法を示す.

```
% asir
[0] load("fac")$
[3] debug$
(debug) list factorial
1   def factorial(X) {
2       if ( !X )
3           return 1;
4       else
5           return X * factorial(X - 1);
6   }
7   end$
(debug) stop at 5                       <-- ブレークポイントの設定
(0) stop at "./fac":5
(debug) quit                            <-- デバッグモードを抜ける
[4] factorial(6);                       <-- factorial(6) の呼び出し
stopped in factorial at line 5 in file "./fac"
5           return X * factorial(X - 1);
(debug) where                           <-- ブレークポイントまでの呼び出し列
の表示
factorial(), line 5 in "./fac"
(debug) print X                         <-- X の値の表示
X = 6
(debug) step                            <-- ステップ実行（函数に入る）
stopped in factorial at line 2 in file "./fac"
2 if ( !X )
(debug) where
factorial(), line 2 in "./fac"
factorial(), line 5 in "./fac"
(debug) print X
X = 5
(debug) delete 0                        <-- ブレークポイント 0 の消去
(debug) cont                            <-- 実行継続
720                                     <-- 結果 = 6!
[5] quit;
```

5.4 デバッガの初期化ファイルの例

前に述べた通り, **Asir** は, 起動時に '$HOME/.dbxinit' を読み込む. このファイルは, dbx のさまざまな初期設定用のコマンドを記述しておくファイルであるが, **Asir** は, alias 行のみを認識する. 例えば,

```
% cat ~/.dbxinit
alias n next
alias c cont
alias p print
alias s step
alias d delete
alias r run
alias l list
alias q quit
```

なる設定により, print, cont など, デバッグモードにおいて頻繁に用いられるコマンドが, それぞれ p, c など, 短い文字列で代用できる. また, デバッグモードにおいて, alias コマンドにより alias の追加ができる.

```
lex_hensel(La,[a,b,c],0,[a,b,c],0);
stopped in gennf at line 226 in file "/home/usr3/noro/asir/gr"
226         N = length(V); Len = length(G); dp_ord(O); PS = newvect(Len);
(debug) p V
V = [a,b,c]
(debug) c
...
```

6 組み込み函数

6.1 数の演算

6.1.1 idiv, irem

idiv(*i1*,*i2*)
:: 整数除算による商.

irem(*i1*,*i2*)
:: 整数除算による剰余.

return　整数

i1 i2　整数

- *i1* の *i2* による整数除算による商, 剰余を求める.
- *i2* は 0 であってはならない.
- 被除数が負の場合, 絶対値に対する値にマイナスをつけた値を返す.
- *i1* % *i2* は, 結果が正に正規化されることを除けば irem() の代わりに用いることができる.
- 多項式の場合は sdiv, srem を用いる.

```
[0] idiv(100,7);
14
[0] idiv(-100,7);
-14
[1] irem(100,7);
2
[1] irem(-100,7);
-2
```

参照　6.3.8 節「sdiv sdivm srem sremm sqr sqrm」p.45, 6.3.10 節「%」p.47.

6.1.2 fac

fac(*i*)　:: *i* の階乗.

return　整数

i　整数

- *i* の階乗を計算する.
- *i* が負の場合は 0 を返す.

```
[0] fac(50);
30414093201713378043612608166064768844377641568960512000000000000
```

6.1.3 igcd,igcdcntl

igcd(*i1*,*i2*)
:: 整数の GCD (最大公約数)

igcdcntl([*i*])
:: 整数 GCD のアルゴリズム選択

return　整数

i1 i2 i　整数

- igcd は *i1* と *i2* の GCD を求める.
- 引数が整数でない場合は, エラーまたは無意味な結果を返す.
- 多項式の場合は, gcd, gcdz を用いる.
- 整数 GCD にはさまざまな方法があり, igcdcntl で設定できる.

  0　Euclid 互除法 (default)

  1　binary GCD

  2　bmod GCD

  3　accelerated integer GCD

  2, 3 は [Weber] による.
  おおむね 3 が高速だが, 例外もある.

```
[0] A=lrandom(10^4)$
[1] B=lrandom(10^4)$
[2] C=lrandom(10^4)$
[3] D=A*C$
[4] E=A*B$
[5] cputime(1)$
[6] igcd(D,E)$
0.6sec + gc : 1.93sec(2.531sec)
[7] igcdcntl(1)$
[8] igcd(D,E)$
0.27sec(0.2635sec)
[9] igcdcntl(2)$
[10] igcd(D,E)$
0.19sec(0.1928sec)
[11] igcdcntl(3)$
[12] igcd(D,E)$
0.08sec(0.08023sec)
```

参照　6.3.20 節「gcd gcdz」p.53.

6.1.4 ilcm

ilcm(*i1*,*i2*)
:: 最小公倍数を求める.

return　整数

i1 i2　　整数

- 整数 *i1*, *i2* の最小公倍数を求める.
- 一方が 0 の場合 0 を返す.

参照　　6.1.3 節「`igcd igcdcntl`」p.34, 6.1.10 節「`mt_save mt_load`」p.36.

6.1.5 isqrt

`isqrt(`*n*`)`　:: 平方根を越えない最大の整数を求める.

return　　非負整数

n　　非負整数

6.1.6 inv

`inv(`*i*`,`*m*`)`　:: *m* を法とする *i* の逆数

return　　整数

i m　　整数

- $ia \equiv 1 \bmod (m)$ なる整数 *a* を求める.
- *i* と *m* は互いに素でなければならないが, `inv()` はそのチェックは行わない.

```
[71] igcd(1234,4321);
1
[72] inv(1234,4321);
3239
[73] irem(3239*1234,4321);
1
```

参照　　6.1.3 節「`igcd igcdcntl`」p.34.

6.1.7 prime, lprime

`prime(`*index*`)`
`lprime(`*index*`)`
:: 素数を返す

return　　整数

index　　整数

- `prime()`, `lprime()` いずれもシステムが内部に持つ素数表の要素を返す. `index` は 0 以上の整数で, 素数表のインデックスに用いられる. `prime()` は 16381 までの素数を小さい順に 1900 個, `lprime()` は, 10 進 8 桁で最大の素数から大きい順に 999 個返す. それ以外のインデックスに対しては 0 を返す.
- より一般的な素数生成函数としては, `pari(nextprime,`*number*`)` がある.

```
[95] prime(0);
2
[96] prime(1228);
9973
```

```
[97] lprime(0);
99999989
[98] lprime(999);
0
```

参照　　6.1.14 節「pari」p.39.

6.1.8 random

random([*seed*])
:: 乱数を生成する.

seed
return　　自然数

- 最大 2^32-1 の非負整数の乱数を生成する.
- 0 でない引数がある時, その値を seed として設定してから, 乱数を生成する.
- default の seed は固定のため, 種を設定しなければ, 生成される乱数の系列は起動毎に一定である.
- 松本眞-西村拓士による Mersenne Twister (http://www.math.keio.ac.jp/matsumoto/mt.html) アルゴリズムの, 彼ら自身による実装を用いている.
- 周期は 2^19937-1 と非常に長い.
- mt_save により state をファイルに save できる. これを mt_load で読み込むことにより, 異る Asir セッション間で一つの乱数の系列を辿ることができる.

参照　　6.1.9 節「lrandom」p.36, 6.1.10 節「mt_save mt_load」p.36.

6.1.9 lrandom

lrandom(*bit*)
:: 多倍長乱数を生成する.

bit

return　　自然数

- 高々 *bit* の非負整数の乱数を生成する.
- random を複数回呼び出して結合し, 指定の bit 長にマスクしている.

参照　　6.1.8 節「random」p.36, 6.1.10 節「mt_save mt_load」p.36.

6.1.10 mt_save, mt_load

mt_save(*fname*)
:: 乱数生成器の現在の状態をファイルにセーブする.

mt_load(*fname*)
:: ファイルにセーブされた乱数生成器の状態をロードする.

return　　0 または 1

fname　　文字列

- ある状態をセーブし, その状態をロードすることで, 一つの疑似乱数系列を, 新規の Asir セッションで続けてたどることができる.

```
[340] random();
3510405877
[341] mt_save("/tmp/mt_state");
1
[342] random();
4290933890
[343] quit;
% asir
This is Asir, Version 991108.
Copyright (C) FUJITSU LABORATORIES LIMITED.
3 March 1994. All rights reserved.
[340] mt_load("/tmp/mt_state");
1
[341] random();
4290933890
```

参照　6.1.8 節「random」p.36, 6.1.9 節「lrandom」p.36.

6.1.11 nm, dn

nm(*rat*)　:: *rat* の分子.

dn(*rat*)　:: *rat* の分母.

return　整数または多項式

rat　有理数または有理式

- 与えられた有理数また有理式の分子及び分母を返す.
- 有理数の場合, 分母は常に正で, 符号は分子が持つ.
- 有理式の場合, 単に分母, 分子を取り出すだけである. 有理式に対しては, 約分は自動的には行われない. red() を明示的に呼び出す必要がある.

```
[2] [nm(-43/8),dn(-43/8)];
[-43,8]
[3] dn((x*z)/(x*y));
y*x
[3] dn(red((x*z)/(x*y)));
y
```

参照　6.3.21 節「red」p.54.

6.1.12 conj, real, imag

real(*comp*)
:: *comp* の実数部分.

imag(*comp*)
:: *comp* の虚数部分.

`conj(`*comp*`)`
:: *comp* の共役複素数.

return comp
複素数

- 複素数に対し, 実部, 虚部, 共役を求める.
- これらは, 多項式に対しても働く.

```
[111] A=(2+@i)^3;
(2+11*@i)
[112] [real(A),imag(A),conj(A)];
[2,11,(2-11*@i)]
```

6.1.13 eval, deval

`eval(`*obj*`[,`*prec*`])`
`deval(`*obj*`)`
:: *obj* の値の評価.

return　数あるいは式

obj　一般の式

prec　整数

- *obj* に含まれる函数の値を可能な限り評価する.
- `deval` は倍精度浮動小数を結果として `eval` の場合, 有理数はそのまま残る.
- `eval` においては, 計算は **PARI** (6.1.14 節「pari」p.39) が行う. `deval` においては, 計算は C 数学ライブラリの関数を用いて行う.
- `deval` は複素数は扱えない.
- `eval` においては, *prec* を指定した場合, 計算は, 10 進 *prec* 桁程度で行われる. *prec* の指定がない場合, 現在設定されている精度で行われる. (6.1.15 節「setprec」p.40 を参照.)
- 扱える函数は，次の通り.
  `sin`, `cos`, `tan`,
  `asin`, `acos`, `atan`,
  `sinh`, `cosh`, `tanh`,
  `asinh`, `acosh`, `atanh`,
  `exp`, `log`, `pow(a,b)` (`a^b`)
- 以下の記号を数として評価できる. ただし `@i` を扱えるのは `eval`, `deval` のみである.

  `@i`　虚数単位

  `@pi`　円周率

  `@e`　自然対数の底

```
[118] eval(exp(@pi*@i));
-1.00000000000000000000000000000
[119] eval(2^(1/2));
```

```
1.4142135623730950488763788073031
[120] eval(sin(@pi/3));
0.866025403784438864674620506632
[121] eval(sin(@pi/3)-3^(1/2)/2,50);
-2.78791084448179148471 E-58
[122] eval(1/2);
1/2
[123] deval(sin(1)^2+cos(1)^2);
1
```

参照　6.14.1 節「ctrl」p.82, 6.1.15 節「setprec」p.40, 6.1.14 節「pari」p.39.

6.1.14 pari

pari(*func*,*arg*,*prec*)
:: **PARI** の函数 *func* を呼び出す.

return　*func* 毎に異なる.

func　**PARI** の函数名

arg　*func* の引数

prec　整数

- **PARI** の函数を呼び出す.
- **PARI** [Batut et al.] は Bordeaux 大学で開発されフリーソフトウェアとして公開されている. **PARI** は数式処理的な機能を有してはいるが, 主なターゲットは整数論に関連した数 (**bignum**, **bigfloat**) の演算で, 四則演算に限らず **bigfloat** によるさまざまな函数値の評価を高速に行うことができる. **PARI** は他のプログラムからサブルーチンライブラリとして用いることができ, また, 'gp' という **PARI** ライブラリのインタフェースにより UNIX のアプリケーションとして利用することもできる. 現在のバージョンは **2.0.17beta** でいくつかの ftp site (たとえば ftp://megrez.ceremab.u-bordeaux.fr/pub/pari) から anonymous ftp できる.
- 最後の引数 *prec* で計算精度を指定できる. *prec* を省略した場合 setprec() で指定した精度となる.
- 現時点で実行できる **PARI** の函数は次の通りである. いずれも 1 引数で **Asir** が対応できる型の引数をとる函数である. なお各々の機能については **PARI** のマニュアルを参照のこと.

  abs, adj, arg, bigomega, binary, ceil, centerlift, cf, classno, classno2, conj, content, denom, det, det2, detr, dilog, disc, discf, divisors, eigen, eintg1, erfc, eta, floor, frac, galois, galoisconj, gamh, gamma, hclassno, hermite, hess, imag, image, image2, indexrank, indsort, initalg, isfund, isprime, ispsp, isqrt, issqfree, issquare, jacobi, jell, ker, keri, kerint, kerintg1, kerint2, kerr, length, lexsort, lift, lindep, lll, lllg1, lllgen, lllgram, lllgramg1, lllgramgen, lllgramint, lllgramkerim, lllgramkerimgen, lllint, lllkerim, lllkerimgen, lllrat, lngamma, logagm, mat, matrixqz2, matrixqz3, matsize, modreverse, mu, nextprime, norm, norml2, numdiv, numer, omega, order, ordred, phi, pnqn, polred, polred2, primroot, psi, quadgen, quadpoly, real, recip, redcomp, redreal, regula, reorder, reverse, rhoreal,

roots, rootslong, round, sigma, signat, simplify, smalldiscf, smallfact, smallpolred, smallpolred2, smith, smith2, sort, sqr, sqred, sqrt, supplement, trace, trans, trunc, type, unit, vec, wf, wf2, zeta

- **Asir** で用いているのは **PARI** のほんの一部の機能であるが, 今後より多くの機能が利用できるよう改良する予定である.

```
/* 行列の固有ベクトルを求める. */
[0] pari(eigen,newmat(2,2,[[1,1],[1,2]]));
[ -1.61803398874989484819771921990 0.61803398874989484826 ]
[ 1 1 ]
/* 1 変数多項式の根を求める. */
[1] pari(roots,t^2-2);
[ -1.41421356237309504876 1.41421356237309504876 ]
```

参照　6.1.15 節「setprec」p.40.

6.1.15 setprec

setprec([*n*])
:: **bigfloat** の桁数を *n* 桁に設定する.

return　整数

n　整数

- 引数がある場合, **bigfloat** の桁数を *n* 桁に設定する. 引数のあるなしにかかわらず, 以前に設定されていた値を返す.
- **bigfloat** の計算は **PARI** (6.1.14 節「pari」p.39) によって行われる.
- **bigfloat** での計算に対し有効である. **bigfloat** の flag を on にする方法は, ctrl を参照.
- 設定できる桁数に上限はないが, 指定した桁数に設定されるとは限らない. 大きめの値を設定するのが安全である.

```
[1] setprec();
9
[2] setprec(100);
9
[3] setprec(100);
96
```

参照　6.14.1 節「ctrl」p.82, ⟨undefined⟩「eval deval」p.⟨undefined⟩, 6.1.14 節「pari」p.39.

6.1.16 setmod

setmod([*p*])
:: 有限体を GF(p) に設定する.

return　整数

n　2^27 未満の素数

- 有限体を GF(p) に設定する. 設定値を返す.

- 有限体の元の型を持つ数は, それ自身はどの有限体に属するかの情報を持たず, 現在設定されている素数 p により GF(p) 上での演算が適用される.
- 位数の大きな有限体に関しては 第 10 章「有限体に関する演算」p.150 参照.

```
[0] A=dp_mod(dp_ptod(2*x,[x]),3,[]);
(2)*<<1>>
[1] A+A;
addmi : invalid modulus
return to toplevel
[1] setmod(3);
3
[2] A+A;
(1)*<<1>>
```

参照　8.10.13 節「dp_mod dp_rat」p.126, 3.2 節「数の型」p.12.

6.1.17 ntoint32, int32ton

ntoint32(*n*)
int32ton(*int32*)
:: 非負整数と符号なし 32bit 整数の間の型変換.

return　符号なし 32bit 整数または非負整数

n　2^32 未満の非負整数

int32　符号なし 32bit 整数

- 非負整数 (識別子 1) の符号なし 32bit 整数 (識別子 10) への変換, またはその逆変換を行う.
- 32bit 整数は **OpenXM** の基本構成要素であり, 整数をその型で送信する必要がある場合に用いる.

参照　第 7 章「分散計算」p.91, 3.2 節「数の型」p.12.

6.2 bit 演算

6.2.1 iand, ior, ixor

iand(*i1*,*i2*)
:: bit ごとの and

ior(*i1*,*i2*)
:: bit ごとの or

ixor(*i1*,*i2*)
:: bit ごとの xor

return　整数

i1 i2　整数

- 整数 *i1*, *i2* の絶対値を bit 列とみて演算する.

- 引数の符号は無視し, 非負の値を返す.

```
[0] ctrl("hex",1);
0x1
[1] iand(0xeeeeeeeeeeeeeeee,0x2984723234812312312);
0x4622224802202202
[2] ior(0xa0a0a0a0a0a0a0a0,0xb0c0b0b0b0b0b0b);
0xabacababababababab
[3] ixor(0xfffffffffff,0x234234234234);
0x2cbdcbdcbdcb
```

参照　6.2.2 節「ishift」p.42.

6.2.2 ishift

ishift(*i*,*count*)
:: bit shift

return　整数

i count　整数

- 整数 *i* の絶対値を bit 列とみて shift する.
- *i* の符号は無視し, 非負の値を返す.
- *count* が正ならば右 shift, 負ならば左 shift を行う.

```
[0] ctrl("hex",1);
0x1
[1] ishift(0x1000000,12);
0x1000
[2] ishift(0x1000,-12);
0x1000000
[3] ixor(0x1248,ishift(1,-16)-1);
```

参照　6.2.1 節「iand ior ixor」p.41.

6.3 多項式, 有理式の演算

6.3.1 var

var(*rat*)　:: *rat* の主変数.

return　不定元

rat　有理式

- 主変数に関しては, 3.1 節「Asir で使用可能な型」p.10 を参照.
- デフォルトの変数順序は次のようになっている.

  x, y, z, u, v, w, p, q, r, s, t, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, 以後は変数の現れた順.

```
[0] var(x^2+y^2+a^2);
x
[1] var(a*b*c*d*e);
a
[2] var(3/abc+2*xy/efg);
abc
```

参照　6.3.7 節「ord」p.45, 6.3.2 節「vars」p.43.

6.3.2 vars

vars(*obj*) :: *obj* に含まれる変数のリスト.

return　リスト

obj　任意

- 与えられた式に含まれる変数のリストを返す.
- 変数順序の高いものから順に並べる.

```
[0] vars(x^2+y^2+a^2);
[x,y,a]
[1] vars(3/abc+2*xy/efg);
[abc,xy,efg]
[2] vars([x,y,z]);
[x,y,z]
```

参照　6.3.1 節「var」p.42, 6.3.3 節「uc」p.43, 6.3.7 節「ord」p.45.

6.3.3 uc

uc()　:: 未定係数法のための不定元を生成する.

return　vtype が 1 の不定元

- uc() を実行するたびに, _0, _1, _2,... という不定元を生成する.
- uc() で生成された不定元は, 直接キーボードから入力することができない. これは, プログラム中で未定係数を自動生成する場合, 入力などに含まれる不定元と同一のものが生成されることを防ぐためである.
- 通常の不定元 (vtype が 0) の自動生成には rtostr(), strtov() を用いる.
- uc() で生成された不定元の不定元としての型 (vtype) は 1 である. (3.3 節「不定元の型」p.14 を参照.)

```
[0] A=uc();
_0
[1] B=uc();
_1
[2] (uc()+uc())^2;
_2^2+2*_3*_2+_3^2
[3] (A+B)^2;
_0^2+2*_1*_0+_1^2
```

参照　6.8.3 節「vtype」p.69, 6.10.1 節「rtostr」p.71, 6.10.2 節「strtov」p.72.

6.3.4 coef

coef(*poly*,*deg*[,*var*])
:: *poly* の *var* (省略時は主変数) に関する *deg* 次の係数.

return　多項式

poly　多項式

var　不定元

deg　自然数

- *poly* の *var* に関する *deg* 次の係数を出力する.
- *var* は, 省略すると主変数 var(*poly*) だとみなされる.
- *var* が主変数でない時, *var* が主変数の場合に比較して効率が落ちる.

```
[0] A = (x+y+z)^3;
x^3+(3*y+3*z)*x^2+(3*y^2+6*z*y+3*z^2)*x+y^3+3*z*y^2+3*z^2*y+z^3
[1] coef(A,1,y);
3*x^2+6*z*x+3*z^2
[2] coef(A,0);
y^3+3*z*y^2+3*z^2*y+z^3
```

参照　6.3.1 節「var」p.42, 6.3.5 節「deg mindeg」p.44.

6.3.5 deg, mindeg

deg(*poly*,*var*)
:: *poly* の, 変数 *var* に関する最高次数.

mindeg(*poly*,*var*)
:: *poly* の, 変数 *var* に関する最低次数.

return　自然数

poly　多項式

var　不定元

- 与えられた多項式の変数 *var* に関する最高次数, 最低次数を出力する.
- 変数 *var* を省略することは出来ない.

```
[0] deg((x+y+z)^10,x);
10
[1] deg((x+y+z)^10,w);
0
[75] mindeg(x^2+3*x*y,x);
1
```

6.3.6 nmono

nmono(*rat*)
:: *rat* の単項式の項数.

return　自然数

rat　　有理式

- 多項式を展開した状態での 0 でない係数を持つ単項式の項数を求める.
- 有理式の場合は, 分子と分母の項数の和が返される.
- 函数形式 (3.3 節「不定元の型」p.14) は, 引数が何であっても単項とみなされる. (1 個の不定元と同じ.)

```
[0] nmono((x+y)^10);
11
[1] nmono((x+y)^10/(x+z)^10);
22
[2] nmono(sin((x+y)^10));
1
```

参照　　6.8.3 節「vtype」p.69.

6.3.7 ord

ord([*varlist*])
:: 変数順序の設定

return　　変数のリスト

varlist　　変数のリスト

- 引数があるとき, 引数の変数リストを先頭に出し, 残りの変数がその後に続くように変数順序を設定する. 引数のあるなしに関わらず, ord() の終了時における変数順序リストを返す.
- この函数による変数順序の変更を行っても, 既にプログラム変数などに代入されている式の内部形式は新しい順序に従っては変更されない. 従って, この函数による順序の変更は, **Asir** の起動直後, あるいは, 新たな変数が現れた時点に行われるべきである. 異なる変数順序のもとで生成された式どうしの演算が行われた場合, 予期せぬ結果が生ずることもあり得る.

```
[0] ord();
[x,y,z,u,v,w,p,q,r,s,t,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,_x,_y,_z,_u,_v,
_w,_p,_q,_r,_s,_t,_a,_b,_c,_d,_e,_f,_g,_h,_i,_j,_k,_l,_m,_n,_o,
exp(_x),(_x)^(_y),log(_x),(_x)^(_y-1),cos(_x),sin(_x),tan(_x),
(-_x^2+1)^(-1/2),cosh(_x),sinh(_x),tanh(_x),
(_x^2+1)^(-1/2),(_x^2-1)^(-1/2)]
[1] ord([dx,dy,dz,a,b,c]);
[dx,dy,dz,a,b,c,x,y,z,u,v,w,p,q,r,s,t,d,e,f,g,h,i,j,k,l,m,n,o,_x,_y,
_z,_u,_v,_w,_p,_q,_r,_s,_t,_a,_b,_c,_d,_e,_f,_g,_h,_i,_j,_k,_l,_m,_n,
_o,exp(_x),(_x)^(_y),log(_x),(_x)^(_y-1),cos(_x),sin(_x),tan(_x),
(-_x^2+1)^(-1/2),cosh(_x),sinh(_x),tanh(_x),
(_x^2+1)^(-1/2),(_x^2-1)^(-1/2)]
```

6.3.8 sdiv, sdivm, srem, sremm, sqr, sqrm

sdiv(*poly1*,*poly2*[,*v*])
sdivm(*poly1*,*poly2*,*mod*[,*v*])
:: *poly1* を *poly2* で割る除算が最後まで実行できる場合に商を求める.

srem(*poly1*,*poly2*[,*v*])

sremm(*poly1*,*poly2*,*mod*[,*v*])

:: *poly1* を *poly2* で割る除算が最後まで実行できる場合に剰余を求める.

sqr(*poly1*,*poly2*[,*v*])

sqrm(*poly1*,*poly2*,*mod*[,*v*])

:: *poly1* を *poly2* で割る除算が最後まで実行できる場合に商, 剰余を求める.

return　sdiv(), sdivm(), srem(), sremm() : 多項式, sqr(), sqrm() : [商, 剰余] なるリスト

poly1 poly2　多項式

v　不定元

mod　素数

- *poly1* を *poly2* の主変数 var(*poly2*) (引数 *v* がある場合には *v*) に関する多項式と見て, *poly2* で, 割り算を行う.
- sdivm(), sremm(), sqrm() は GF(*mod*) 上で計算する.
- 多項式の除算は, 主係数どうしの割算により得られた商と, 主変数の適当な冪の積を *poly2* に掛けて, *poly1* から引くという操作を *poly1* の次数が *poly2* の次数より小さくなるまで繰り返して行う. この操作が, 多項式の範囲内で行われるためには, 各ステップにおいて主係数どうしの除算が, 多項式としての整除である必要がある. これが, 「除算が最後まで実行できる」ことの意味である.
- 典型的な場合として, *poly2* の主係数が, 有理数である場合, あるいは, *poly2* が *poly1* の因子であることがわかっている場合などがある.
- sqr() は商と剰余を同時に求めたい時に用いる.
- 整数除算の商, 剰余は idiv, irem を用いる.
- 係数に対する剰余演算は % を用いる.

```
[0] sdiv((x+y+z)^3,x^2+y+a);
x+3*y+3*z
[1] srem((x+y+z)^2,x^2+y+a);
(2*y+2*z)*x+y^2+(2*z-1)*y+z^2-a
[2] X=(x+y+z)*(x-y-z)^2;
x^3+(-y-z)*x^2+(-y^2-2*z*y-z^2)*x+y^3+3*z*y^2+3*z^2*y+z^3
[3] Y=(x+y+z)^2*(x-y-z);
x^3+(y+z)*x^2+(-y^2-2*z*y-z^2)*x-y^3-3*z*y^2-3*z^2*y-z^3
[4] G=gcd(X,Y);
x^2-y^2-2*z*y-z^2
[5] sqr(X,G);
[x-y-z,0]
[6] sqr(Y,G);
[x+y+z,0]
[7] sdiv(y*x^3+x+1,y*x+1);
divsp: cannot happen
return to toplevel
```

参照　6.1.1 節「idiv irem」p.33, 6.3.10 節「%」p.47.

6.3.9 tdiv

tdiv(*poly1*,*poly2*)
:: *poly1* が *poly2* で割り切れるかどうか調べる.

return　割り切れるならば商, 割り切れなければ 0

poly1 poly2
多項式

- *poly2* が *poly1* を多項式として割り切るかどうか調べる.
- ある多項式が既約因子であることはわかっているが, その重複度がわからない場合に, tdiv() を繰り返し呼ぶことにより重複度がわかる.

```
[11] Y=(x+y+z)^5*(x-y-z)^3;
x^8+(2*y+2*z)*x^7+(-2*y^2-4*z*y-2*z^2)*x^6
+(-6*y^3-18*z*y^2-18*z^2*y-6*z^3)*x^5
+(6*y^5+30*z*y^4+60*z^2*y^3+60*z^3*y^2+30*z^4*y+6*z^5)*x^3
+(2*y^6+12*z*y^5+30*z^2*y^4+40*z^3*y^3+30*z^4*y^2+12*z^5*y+2*z^6)*x^2
+(-2*y^7-14*z*y^6-42*z^2*y^5-70*z^3*y^4-70*z^4*y^3-42*z^5*y^2
-14*z^6*y-2*z^7)*x-y^8-8*z*y^7-28*z^2*y^6-56*z^3*y^5-70*z^4*y^4
-56*z^5*y^3-28*z^6*y^2-8*z^7*y-z^8
[12] for(I=0,F=x+y+z,T=Y; T=tdiv(T,F); I++);
[13] I;
5
```

参照　6.3.8 節「sdiv sdivm srem sremm sqr sqrm」p.45.

6.3.10 %

poly % *m*　:: 整数による剰余

return　整数または多項式

poly　整数または整数係数多項式

m　整数

- *poly* の各係数を *m* で割った剰余で置き換えた多項式を返す.
- 結果の係数は全て正の整数となる.
- *poly* は整数でもよい. この場合, 結果が正に正規化されることを除けば irem() と同様に用いることができる.
- *poly* の係数, *m* とも整数である必要があるが, チェックは行なわれない.

```
[0] (x+2)^5 % 3;
x^5+x^4+x^3+2*x^2+2*x+2
[1] (x-2)^5 % 3;
x^5+2*x^4+x^3+x^2+2*x+1
[2] (-5) % 4;
3
[3] irem(-5,4);
-1
```

参照　6.1.1 節「idiv irem」p.33.

6.3.11 subst, psubst

subst(*rat*[,*varn*,*ratn*]*)

psubst(*rat*[,*var*,*rat*]*)

:: *rat* の *varn* に *ratn* を代入 (*n*=1,2,... で左から右に順次代入する).

return　有理式

rat ratn　有理式

varn　不定元

- 有理式の特定の不定元に, 定数あるいは多項式, 有理式などを代入するのに用いる.
- subst(*rat*,*var1*,*rat1*,*var2*,*rat2*,...) は, subst(subst(*rat*,*var1*,*rat1*),*var2*,*rat2*,...) と同じ意味である.
- 入力の左側から順に代入を繰り返すために, 入力の順によって結果が変わることがある.
- subst() は, sin() などの函数の引数に対しても代入を行う. psubst() は, このような函数を一つの独立した不定元と見なして, その引数には代入は行わない. (partial substitution のつもり)
- **Asir** では, 有理式の約分は自動的には行わないため, 有理式の代入は, 思わぬ計算時間の増大を引き起こす場合がある. 有理式を代入する場合には, 問題に応じた独自の函数を書いて, なるべく分母, 分子が大きくならないように配慮することもしばしば必要となる.
- 分数を代入する場合も同様である.
- substの引数 *rat* がリスト, 配列, 行列, あるいは分散表現多項式であった場合には, それぞれの要素または係数に対して再帰的に substを行う.

```
[0] subst(x^3-3*y*x^2+3*y^2*x-y^3,y,2);
x^3-6*x^2+12*x-8
[1] subst(@@,x,-1);
-27
[2] subst(x^3-3*y*x^2+3*y^2*x-y^3,y,2,x,-1);
-27
[3] subst(x*y^3,x,y,y,x);
x^4
[4] subst(x*y^3,y,x,x,y);
y^4
[5] subst(x*y^3,x,t,y,x,t,y);
y*x^3
[6] subst(x*sin(x),x,t);
sint(t)*t
[7] psubst(x*sin(x),x,t);
sin(x)*t
```

6.3.12 diff

diff(*rat*[,*varn*]*)

diff(*rat*,*varlist*)

:: *rat* を *varn* あるいは *varlist* の中の変数で順次微分する.

return　式

rat　　有理式 (初等函数を含んでもよい)

varn　　不定元

varlist　　不定元のリスト

- 与えられた初等函数を *varn* あるいは *varlist* の中の変数で順次微分する.
- 左側の不定元より, 順に微分していく. つまり, `diff`(*rat*,`x,y`) は, `diff`(`diff`(*rat*,`x`),`y`) と同じである.

```
[0] diff((x+2*y)^2,x);
2*x+4*y
[1] diff((x+2*y)^2,x,y);
4
[2] diff(x/sin(log(x)+1),x);
(sin(log(x)+1)-cos(log(x)+1))/(sin(log(x)+1)^2)
[3] diff(sin(x),[x,x,x,x]);
sin(x)
```

6.3.13 ediff

`ediff`(*poly*`[,`*varn*`]*`)

`ediff`(*poly*`,`*varlist*`)`

:: *poly* を *varn* あるいは *varlist* の中の変数で順次オイラー微分する.

return　　多項式

poly　　多項式

varn　　不定元

varlist　　不定元のリスト

- 左側の不定元より, 順にオイラー微分していく. つまり, `ediff`(*poly*,`x,y`) は, `ediff`(`ediff`(*poly*,`x`),`y`) と同じである.

```
[0] ediff((x+2*y)^2,x);
2*x^2+4*y*x
[1] ediff((x+2*y)^2,x,y);
4*y*x
```

6.3.14 res

`res`(*var*`,`*poly1*`,`*poly2*`[,`*mod*`]`)

:: *var* に関する *poly1* と *poly2* の終結式.

return　　多項式

var　　不定元

poly1 poly2

多項式

mod　　素数

- 二つの多項式 *poly1* と *poly2* の, 変数 *var* に関する終結式を求める.

- 部分終結式アルゴリズムによる.
- 引数 *mod* がある時, GF(*mod*) 上での計算を行う.

```
[0] res(t,(t^3+1)*x+1,(t^3+1)*y+t);
-x^3-x^2-y^3
```

6.3.15 fctr, sqfr

`fctr(`*poly*`)`
:: *poly* を既約因子に分解する.

`sqfr(`*poly*`)`
:: *poly* を無平方分解する.

return　リスト

poly　有理数係数の多項式

- 有理数係数の多項式 *poly* を因数分解する. `fctr()` は既約因子分解, `sqfr()` は無平方因子分解.
- 結果は [[**数係数**,1],[**因子**, **重複度**],...] なるリスト.
- **数係数** と 全ての **因子^重複度** の積が *poly* と等しい.
- **数係数** は, (*poly*/**数係数**) が, 整数係数で, 係数の GCD が 1 となるような多項式になるように選ばれている. (`ptozp()` 参照)

```
[0] fctr(x^10-1);
[[1,1],[x-1,1],[x+1,1],[x^4+x^3+x^2+x+1,1],[x^4-x^3+x^2-x+1,1]]
[1] fctr(x^3+y^3+(z/3)^3-x*y*z);
[[1/27,1],[9*x^2+(-9*y-3*z)*x+9*y^2-3*z*y+z^2,1],[3*x+3*y+z,1]]
[2] A=(a+b+c+d)^2;
a^2+(2*b+2*c+2*d)*a+b^2+(2*c+2*d)*b+c^2+2*d*c+d^2
[3] fctr(A);
[[1,1],[a+b+c+d,2]]
[4] A=(x+1)*(x^2-y^2)^2;
x^5+x^4-2*y^2*x^3-2*y^2*x^2+y^4*x+y^4
[5] sqfr(A);
[[1,1],[x+1,1],[-x^2+y^2,2]]
[6] fctr(A);
[[1,1],[x+1,1],[-x-y,2],[x-y,2]]
```

参照　6.3.16 節「`ufctrhint`」p.50.

6.3.16 ufctrhint

`ufctrhint(`*poly*`,`*hint*`)`
:: 次数情報を用いた 1 変数多項式の因数分解

return　リスト

poly　有理数係数の 1 変数多項式

hint　自然数

- 各既約因子の次数が *hint* の倍数であることがわかっている場合に *poly* の既約因子分解を `fctr()` より効率良く行う. *poly* が, d 次の拡大体上におけるある多項式のノルム (第 9 章「代数的数に関する演算」p.140) で無平方である場合, 各既約因子の次数は d の倍数となる. このような場合に用いられる.

```
[10] A=t^9-15*t^6-87*t^3-125;
t^9-15*t^6-87*t^3-125
0msec
[11] N=res(t,subst(A,t,x-2*t),A);
-x^81+1215*x^78-567405*x^75+139519665*x^72-19360343142*x^69
+1720634125410*x^66-88249977024390*x^63-4856095669551930*x^60
+1999385245240571421*x^57-1557968995259025151 5*x^54
+15956967531741971462865*x^51
...
+14039558872035397353552612361266144455065 9875*x^6
+10122324287343155430042768923500799484375*x^3
+13926274344440731013345902118273331445312 5
980msec + gc : 250msec
[12] sqfr(N);
[[-1,1],[x^81-1215*x^78+567405*x^75-139519665*x^72+19360343142*x^69
-1720634125410*x^66+88249977024390*x^63+4856095669551930*x^60
-1999385245240571421*x^57+1557968995259025151 5*x^54
...
-10122324287343155430042768923500799484375*x^3
-13926274344440731013345902118273331445312 5,1]]
20msec
[13] fctr(N);
[[-1,1],[x^9-405*x^6-63423*x^3-2460375,1],
[x^18-486*x^15+98739*x^12-9316620*x^9+945468531*x^6-12368049246*x^3
+296607516309,1],[x^18-8667*x^12+19842651*x^6+19683,1],
[x^18-324*x^15+44469*x^12-1180980*x^9+427455711*x^6+2793253896*x^3
+31524548679,1],
[x^18+10773*x^12+2784051*x^6+307546875,1]]
167.050sec + gc : 1.890sec
[14] ufctrhint(N,9);
[[-1,1],[x^9-405*x^6-63423*x^3-2460375,1],
[x^18-486*x^15+98739*x^12-9316620*x^9+945468531*x^6-12368049246*x^3
+296607516309,1],[x^18-8667*x^12+19842651*x^6+19683,1],
[x^18-324*x^15+44469*x^12-1180980*x^9+427455711*x^6+2793253896*x^3
+31524548679,1],
[x^18+10773*x^12+2784051*x^6+307546875,1]]
119.340sec + gc : 1.300sec
```

参照　　6.3.15 節「`fctr sqfr`」p.50.

6.3.17 `modfctr`

`modfctr`(*poly*,*mod*)
:: 有限体上での多項式の因数分解

return　　リスト

poly　　整数係数の多項式

mod　　自然数

- 2^29 未満の自然数 *mod* を標数とする素体上で多項式 *poly* を既約因子に分解する.
- 結果は [[**数係数**,1],[**因子**, **重複度**],...] なるリスト.
- **数係数** と 全ての **因子^重複度** の積が *poly* と等しい.
- 大きな位数を持つ有限体上の因数分解には `fctr_ff` を用いる. (第 10 章「有限体に関する演算」p.150,10.5.16 節「fctr_ff」p.159 参照).

```
[0] modfctr(x^10+x^2+1,2147483647);
[[1,1],[x+1513477736,1],[x+2055628767,1],[x+91854880,1],
[x+634005911,1],[x+1513477735,1],[x+634005912,1],
[x^4+1759639395*x^2+2045307031,1]]
[1] modfctr(2*x^6+(y^2+z*y)*x^4+2*z*y^3*x^2+(2*z^2*y^2+z^3*y)*x+z^4,3);
[[2,1],[2*x^3+z*y*x+z^2,1],[2*x^3+y^2*x+2*z^2,1]]
```

参照　　6.3.15 節「`fctr sqfr`」p.50.

6.3.18 ptozp

`ptozp(`*poly*`)`
:: *poly* を有理数倍して整数係数多項式にする.

return　　多項式

poly　　多項式

- 与えられた多項式 *poly* に適当な有理数を掛けて, 整数係数かつ係数の GCD が 1 になるようにする.
- 分数の四則演算は, 整数の演算に比較して遅いため, 種々の多項式演算の前に, 多項式を整数係数にしておくことが望ましい.
- 有理式を約分する `red()` で分数係数有理式を約分しても, 分子多項式の係数は有理数のままであり, 有理式の分子を求める `nm()` では, 分数係数多項式は, 分数係数のままの形で出力されるため, 直ちに整数係数多項式を得る事は出来ない.
- オプション factor が設定された場合の戻り値はリスト [g,c] である. ここで c は有理数であり, g がオプションのない場合の戻り値であり, *poly* = c*g となる.

```
[0] ptozp(2*x+5/3);
6*x+5
[1] nm(2*x+5/3);
2*x+5/3
```

参照　　6.1.11 節「`nm dn`」p.37.

6.3.19 prim, cont

`prim(`*poly*`[,`*v*`])`
:: *poly* の原始的部分 (primitive part).

`cont(`*poly*`[,`*v*`])`
:: *poly* の容量 (content).

return poly
　　有理数係数多項式

v　　不定元

- *poly* の主変数 (引数 *v* がある場合には *v*) に関する原始的部分, 容量を求める.

```
[0] E=(y-z)*(x+y)*(x-z)*(2*x-y);
(2*y-2*z)*x^3+(y^2-3*z*y+2*z^2)*x^2+(-y^3+z^2*y)*x+z*y^3-z^2*y^2
[1] prim(E);
2*x^3+(y-2*z)*x^2+(-y^2-z*y)*x+z*y^2
[2] cont(E);
y-z
[3] prim(E,z);
(y-z)*x-z*y+z^2
```

参照　　6.3.1 節「var」p.42, 6.3.7 節「ord」p.45.

6.3.20 gcd, gcdz

gcd(*poly1*,*poly2*[,*mod*])

gcdz(*poly1*,*poly2*)
　　:: *poly1* と *poly2* の gcd.

return　　多項式

poly1 poly2
　　多項式

mod　　素数

- 二つの多項式の最大公約式 (GCD) を求める.
- gcd() は有理数体上の多項式としての GCD を返す. すなわち, 結果は整数係数で, かつ係数の GCD が 1 になるような多項式, または, 互いに素の場合は 1 を返す.
- gcdz() は *poly1*, *poly2* ともに整数係数の場合に, 整数環上の多項式としての GCD を返す. すなわち, gcd() の値に, 係数全体の整数 GCD の値を掛けたものを返す.
- 引数 *mod* がある時, gcd() は GF(*mod*) 上での GCD を返す.
- gcd(), gcdz() Extended Zassenhaus アルゴリズムによる. 有限体上の GCD は PRS アルゴリズムによっているため, 大きな問題, GCD が 1 の場合などにおいて効率が悪い.

```
[0] gcd(12*(x^2+2*x+1)^2,18*(x^2+(y+1)*x+y)^3);
x^3+3*x^2+3*x+1
[1] gcdz(12*(x^2+2*x+1)^2,18*(x^2+(y+1)*x+y)^3);
6*x^3+18*x^2+18*x+6
[2] gcd((x+y)*(x-y)^2,(x+y)^2*(x-y));
x^2-y^2
[3] gcd((x+y)*(x-y)^2,(x+y)^2*(x-y),2);
x^3+y*x^2+y^2*x+y^3
```

参照　　6.1.3 節「igcd igcdcntl」p.34.

6.3.21 red

`red`(*rat*)　:: *rat* を約分したもの.

return　有理式

rat　有理式

- **Asir** は有理数の約分を常に自動的に行う. しかし, 有理式については通分は行うが, 約分はユーザーが指定しない限り行わない. この約分を行うコマンドが `red` である.
- EZGCD により *rat* の分子, 分母を約分する.
- 出力される有理式の分母の多項式は, 各係数の GCD が 1 の整数係数多項式である. 分子については整数係数多項式となるとは限らない.
- GCD は大変重い演算なので, 他の方法で除ける共通因子は可能な限り除くのが望ましい. また, 分母, 分子が大きくなってからのこの函数の呼び出しは, 非常に時間が掛かる場合が多い. 有理式演算を行う場合は, ある程度頻繁に, 約分を行う必要がある.

```
[0] (x^3-1)/(x-1);
(x^3-1)/(x-1)
[1] red((x^3-1)/(x-1));
x^2+x+1
[2] red((x^3+y^3+z^3-3*x*y*z)/(x+y+z));
x^2+(-y-z)*x+y^2-z*y+z^2
[3] red((3*x*y)/(12*x^2+21*y^3*x));
(y)/(4*x+7*y^3)
[4] red((3/4*x^2+5/6*x)/(2*y*x+4/3*x));
(9/8*x+5/4)/(3*y+2)
```

参照　6.1.11 節「`nm dn`」p.37, 6.3.20 節「`gcd gcdz`」p.53, 6.3.18 節「`ptozp`」p.52.

6.4 一変数多項式の演算

6.4.1 umul, umul_ff, usquare, usquare_ff, utmul, utmul_ff

`umul`(*p1*,*p2*)
`umul_ff`(*p1*,*p2*)
:: 一変数多項式の高速乗算

`usquare`(*p1*)
`usquare_ff`(*p1*)
:: 一変数多項式の高速 2 乗算

`utmul`(*p1*,*p2*,*d*)
`utmul_ff`(*p1*,*p2*,*d*)
:: 一変数多項式の高速乗算 (打ち切り次数指定)

return　一変数多項式

p1 p2　一変数多項式

d　非負整数

- 一変数多項式の乗算を, 次数に応じて決まるアルゴリズムを用いて高速に行う.

- umul(), usquare(), utmul() は係数を整数と見なして, 整数係数の多項式として積を求める. 係数が有限体 GF(p) の元の場合には, 係数は 0 以上 p 未満の整数と見なされる.
- umul_ff(), usquare_ff(), utmul_ff() は, 係数を有限体の元と見なして, 有限体上の多項式として積を求める. ただし, 引数の係数が整数の場合, 整数係数の多項式を返す場合もあるので, これらを呼び出した結果が有限体係数であることを保証するためにはあらかじめ simp_ff() で係数を有限体の元に変換しておくとよい.
- umul_ff(), usquare_ff(), utmul_ff() は, GF(2^n) 係数の多項式を引数に取れない.
- umul(), umul_ff() の結果は *p1*, *p2* の積, usquare(), usquare_ff() の結果は *p1* の 2 乗, utmul(), utmul_ff() の結果は *p1*, *p2* の積の, *d* 次以下の部分となる.
- いずれも, set_upkara() (utmul, utmul_ff については set_uptkara()) で返される値以下の次数に対しては通常の筆算形式の方法, set_upfft() で返される値以下の次数に対しては Karatsuba 法, それ以上では FFT および中国剰余定理が用いられる. すなわち, 整数に対する FFT ではなく, 十分多くの 1 ワード以内の法 *mi* を用意し, *p1*, *p2* の係数を *mi* で割った余りとしたものの積を, FFT で計算し, 最後に中国剰余定理で合成する. その際, 有限体版の関数においては, 最後に基礎体を表す法で各係数の剰余を計算するが, ここでは Shoup によるトリック [Shoup] を用いて高速化してある.

```
[176] load("fff")$
[177] cputime(1)$
0sec(1.407e-05sec)
[178] setmod_ff(2^160-47);
1461501637330902918203684832716283019655932542929
0sec(0.00028sec)
[179] A=randpoly_ff(100,x)$
0sec(0.001422sec)
[180] B=randpoly_ff(100,x)$
0sec(0.00107sec)
[181] for(I=0;I<100;I++)A*B;
7.77sec + gc : 8.38sec(16.15sec)
[182] for(I=0;I<100;I++)umul(A,B);
2.24sec + gc : 1.52sec(3.767sec)
[183] for(I=0;I<100;I++)umul_ff(A,B);
1.42sec + gc : 0.24sec(1.653sec)
[184] for(I=0;I<100;I++)usquare_ff(A);
1.08sec + gc : 0.21sec(1.297sec)
[185] for(I=0;I<100;I++)utmul_ff(A,B,100);
1.2sec + gc : 0.17sec(1.366sec)
[186] deg(utmul_ff(A,B,100),x);
100
```

参照　6.4.3 節「set_upkara set_uptkara set_upfft」p.56, 6.4.2 節「kmul ksquare ktmul」p.55.

6.4.2 kmul, ksquare, ktmul

kmul(*p1*,*p2*)
:: 一変数多項式の高速乗算

ksquare(*p1*)
:: 一変数多項式の高速 2 乗算

`ktmul(`*p1*`,`*p2*`,`*d*`)`
:: 一変数多項式の高速乗算 (打ち切り次数指定)

return　一変数多項式

p1 p2　一変数多項式

d　非負整数

- 一変数多項式の乗算を Karatsuba 法で行う.
- 基本的には `umul` と同様だが, 次数が大きくなっても FFT を用いた高速化は行わない.
- GF(2^n) 係数の多項式にも用いることができる.

```
[0] load("code/fff");
1
[34] setmod_ff(defpoly_mod2(160));
x^160+x^5+x^3+x^2+1
[35] A=randpoly_ff(100,x)$
[36] B=randpoly_ff(100,x)$
[37] umul(A,B)$
umul : invalid argument
return to toplevel
[37] kmul(A,B)$
```

6.4.3 set_upkara, set_uptkara, set_upfft

`set_upkara([`*threshold*`])`
`set_uptkara([`*threshold*`])`
`set_upfft([`*threshold*`])`
:: 1 変数多項式の積演算における N^2 , Karatsuba, FFT アルゴリズムの切替えの閾値

return　設定されている値

threshold　非負整数

- いずれも, 一変数多項式の積の計算における, アルゴリズム切替えの閾値を設定する.
- 一変数多項式の積は, 次数 N が小さい範囲では通常の N^2 アルゴリズム, 中程度の場合 Karatsuba アルゴリズム, 大きい場合には FFT アルゴリズムで計算される. この切替えの次数を設定する.
- 詳細は, それぞれの積関数の項を参照のこと.

参照　6.4.2 節「`kmul ksquare ktmul`」p.55,　6.4.1 節「`umul umul_ff usquare usquare_ff utmul utmul_ff`」p.54.

6.4.4 utrunc, udecomp, ureverse

`utrunc(`*p*`,`*d*`)`
`udecomp(`*p*`,`*d*`)`
`ureverse(`*p*`)`
:: 多項式に対する操作

return　一変数多項式あるいは一変数多項式のリスト

p　　一変数多項式

d　　非負整数

- p の変数を x とする. このとき $p = p1$+x^(d+1)$p2$ ($p1$ の次数は d 以下) と分解できる. utrunc() は $p1$ を返し, udecomp() は [$p1$,$p2$] を返す.
- p の次数を e とし, i 次の係数を $p[i]$ とすれば, ureverse() は $p[e]$+p[e-1]x+... を返す.

```
[132] utrunc((x+1)^10,5);
252*x^5+210*x^4+120*x^3+45*x^2+10*x+1
[133] udecomp((x+1)^10,5);
[252*x^5+210*x^4+120*x^3+45*x^2+10*x+1,x^4+10*x^3+45*x^2+120*x+210]
[134] ureverse(3*x^3+x^2+2*x);
2*x^2+x+3
```

参照　　6.4.6 節「udiv urem urembymul urembymul_precomp ugcd」p.58.

6.4.5 uinv_as_power_series, ureverse_inv_as_power_series

uinv_as_power_series(p,d)
ureverse_inv_as_power_series(p,d)
:: 多項式を冪級数とみて, 逆元計算

return　　一変数多項式

p　　一変数多項式

d　　非負整数

- uinv_as_power_series(p,d) は, 定数項が 0 でない多項式 p に対し, pr-1 の最低次数が d+1 以上になるような 高々 d 次の多項式 r を求める.
- ureverse_inv_as_power_series(p,d) は p の次数を e とするとき, $p1$=ureverse(p,e) に対して uinv_as_power_series($p1$,d) を計算する.
- rembymul_precomp() の引数として用いる場合, ureverse_inv_as_power_series() の結果をそのまま用いることができる.

```
[123] A=(x+1)^5;
x^5+5*x^4+10*x^3+10*x^2+5*x+1
[124] uinv_as_power_series(A,5);
-126*x^5+70*x^4-35*x^3+15*x^2-5*x+1
[126] A*R;
-126*x^10-560*x^9-945*x^8-720*x^7-210*x^6+1
[127] A=x^10+x^9;
x^10+x^9
[128] R=ureverse_inv_as_power_series(A,5);
-x^5+x^4-x^3+x^2-x+1
[129] ureverse(A)*R;
-x^6+1
```

参照　　6.4.4 節「utrunc udecomp ureverse」p.56, 6.4.6 節「udiv urem urembymul urembymul_precomp ugcd」p.58.

6.4.6 udiv, urem, urembymul, urembymul_precomp, ugcd

udiv(*p1*,*p2*)

urem(*p1*,*p2*)

urembymul(*p1*,*p2*)

urembymul_precomp(*p1*,*p2*,*inv*)

ugcd(*p1*,*p2*)

:: 一変数多項式の除算, GCD

return　一変数多項式

p1 p2 inv　一変数多項式

- 一変数多項式 *p1*, *p2* に対し, udiv は商, urem, urembymul は剰余, ugcd は GCD を返す. これらは, 密な一変数多項式に対する高速化を図ったものである. urembymul は, *p2* による剰余計算を, *p2* の冪級数としての逆元計算および, 乗算 2 回に置き換えたもので, 次数が大きい場合に有効である.
- urembymul_precomp は, 固定された多項式による剰余計算を多数行う場合などに効果を発揮する. 第 3 引数は, あらかじめ ureverse_inv_as_power_series() により計算しておく.

```
[177] setmod_ff(2^160-47);
1461501637330902918203684832716283019655932542929
[178] A=randpoly_ff(200,x)$
[179] B=randpoly_ff(101,x)$
[180] cputime(1)$
0sec(1.597e-05sec)
[181] srem(A,B)$
0.15sec + gc : 0.15sec(0.3035sec)
[182] urem(A,B)$
0.11sec + gc : 0.12sec(0.2347sec)
[183] urembymul(A,B)$
0.08sec + gc : 0.09sec(0.1651sec)
[184] R=ureverse_inv_as_power_series(B,101)$
0.04sec + gc : 0.03sec(0.063sec)
[185] urembymul_precomp(A,B,R)$
0.03sec(0.02501sec)
```

参照　6.4.5 節「uinv_as_power_series ureverse_inv_as_power_series」p.57.

6.5 リストの演算

6.5.1 car, cdr, cons, append, reverse, length

car(*list*)　:: 空でない *list* の先頭要素.

cdr(*list*)　:: 空でない *list* から先頭要素を取り除いたリスト.

cons(*obj*,*list*)

:: *list* の先頭に *obj* を付け加えたリスト.

append(*list1*,*list2*)
:: *list1* と *list2* をこの順に 1 つにしたリスト.

reverse(*list*)
:: *list* を逆順にしたリスト.

length(*list*|*vect*)
:: *list* の長さ, または *vect* の長さ.

return　car() : 任意, cdr(), cons(), append(), reverse() : リスト, length() : 自然数

list list1 list2
リスト

obj　任意

- リストは [*obj1*,*obj2*,...] と表される. *obj1* が先頭要素である.
- car() は, 空でない *list* の先頭要素を出力する. 空リストが入力された場合は, 空リストが出力される.
- cdr() は, 空でない *list* から先頭要素を取り除いたリストを出力する. 空リストが入力された場合は, 空リストが出力される.
- cons() は, *list* の先頭に *obj* を付け加えたリストを出力する.
- append() は, *list1* の要素と *list2* のすべての要素を結合させたリスト [*list1* の要素の並び,*list2* の要素の並び] を出力する.
- reverse() は, *list* を逆順にしたリストを出力する.
- length() は, *list* または *vect* の長さを出力する. 行列の要素の個数は, size() を用いる.
- リストは読み出し専用で, 要素の入れ替えはできない.
- リストの n 番目の要素の取り出しは, cdr() を n 回適用した後 car() を適用することにより可能であるが, 便法として, ベクトル, 行列などの配列と同様, インデックス [n] を後ろに付けることにより取り出すことができる. ただし, システム内部では, 実際にポインタを n 回たどるので, 後ろの要素ほど取り出しに時間がかかる.
- cdr() は新しいセルを生成しないが, append() は, 実際には第 1 引数のリストの長さだけの cons() の繰り返しとなるため, 第 1 引数のリストが長い場合には多くのメモリを消費することになる. reverse() に関しても同様である.

```
[0] L = [[1,2,3],4,[5,6]];
[[1,2,3],4,[5,6]]
[1] car(L);
[1,2,3]
[2] cdr(L);
[4,[5,6]]
[3] cons(x*y,L);
[y*x,[1,2,3],4,[5,6]]
[4] append([a,b,c],[d]);
[a,b,c,d]
[5] reverse([a,b,c,d]);
[d,c,b,a]
[6] length(L);
3
```

```
[7] length(ltov(L));
3
[8] L[2][0];
5
```

6.6 配列

6.6.1 newvect

newvect(*len*[,*list*])
:: 長さ *len* のベクトルを生成する.

return ベクトル

len 自然数

list リスト

- 長さ *len* のベクトルを生成する. 第 2 引数がない場合, 各成分は 0 に初期化される. 第 2 引数がある場合, インデックスの小さい成分から, リストの各要素により初期化される. 各要素は, 先頭から順に使われ, 足りない分は 0 が埋められる.
- ベクトルの成分は, 第 0 成分から第 *len*-1 成分となる. (第 1 成分からではない事に注意.)
- リストは各成分が, ポインタを辿る事によってシーケンシャルに呼び出されるのに対し, ベクトルは各成分が第一成分からのメモリ上の displacement (変位) によってランダムアクセスで呼び出され, その結果, 成分のアクセス時間に大きな差が出てくる. 成分アクセスは, リストでは, 成分の量が増えるに従って時間がかかるようになるが, ベクトルでは, 成分の量に依存せずほぼ一定である.
- **Asir** では, 縦ベクトル, 横ベクトルの区別はない. 行列を左から掛ければ縦ベクトルとみなされるし, 右から掛ければ横ベクトルとみなされる.
- ベクトルの長さは size() によって得られる.
- 函数の引数としてベクトルを渡した場合, 渡された函数は, そのベクトルの成分を書き換えることができる.

```
[0] A=newvect(5);
[ 0 0 0 0 0 ]
[1] A=newvect(5,[1,2,3,4,[5,6]]);
[ 1 2 3 4 [5,6] ]
[2] A[0];
1
[3] A[4];
[5,6]
[4] size(A);
[5]
[5] def afo(V) { V[0] = x; }
[6] afo(A)$
[7] A;
[ x 2 3 4 [5,6] ]
```

参照 6.5.5 節「newmat」p.62, 6.6.6 節「size」p.63, 6.6.2 節「ltov」p.61, 6.6.3 節「vtol」p.61.

6.6.2 ltov

`ltov(`*list*`)` :: リストをベクトルに変換する.

return　ベクトル

list　リスト

- リスト *list* を同じ長さのベクトルに変換する.
- この関数は `newvect(length(`*list*`),` *list*`)` に等しい.

```
[3] A=[1,2,3];
[4] ltov(A);
[ 1 2 3 ]
```

参照　6.6.1 節「newvect」p.60, 6.6.3 節「vtol」p.61.

6.6.3 vtol

`vtol(`*vect*`)`
:: ベクトルをリストに変換する.

return　リスト

vect　ベクトル

- 長さ *n* のベクトル *vect* を `[`*vect*`[0],...,`*vect*`[`*n*`-1]]` なるリストに変換する.
- リストからベクトルへの変換は `newvect()` で行う.

```
[3] A=newvect(3,[1,2,3]);
[ 1 2 3 ]
[4] vtol(A);
[1,2,3]
```

参照　6.6.1 節「newvect」p.60, 6.6.2 節「ltov」p.61.

6.6.4 newbytearray

`newbytearray(`*len*`,[`*listorstring*`])`
:: 長さ *len* の byte array を生成する.

return　byte array

len　自然数

listorstring
リストまたは文字列

- `newvect` と同様にして byte array を生成する. similar to that of `newvect`.
- 文字列で初期値を指定することも可能である.
- byte array の要素のアクセスは配列と同様である.

```
[182] A=newbytearray(3);
|00 00 00|
[183] A=newbytearray(3,[1,2,3]);
|01 02 03|
```

```
[184] A=newbytearray(3,"abc");
|61 62 63|
[185] A[0];
97
[186] A[1]=123;
123
[187] A;
|61 7b 63|
```

参照　6.6.1 節「newvect」p.60.

6.6.5 newmat

newmat(*row*,*col* [,[[*a*,*b*,...],[*c*,*d*,...],...]])
:: *row* 行 *col* 列の行列を生成する.

return　行列

row col　自然数

a b c d　任意

- *row* 行 *col* 列の行列を生成する. 第 3 引数がない場合, 各成分は 0 に初期化される. 第 3 引数がある場合, インデックスの小さい成分から, 各行が, リストの各要素 (これはまたリストである) により初期化される. 各要素は, 先頭から順に使われ, 足りない分は 0 が埋められる.
- 行列のサイズは size() で得られる.
- M が行列のとき, M[I] により第 I 行をベクトルとして取り出すことができる. このベクトルは, もとの行列と成分を共有しており, いずれかの成分を書き換えれば, 他の対応する成分も書き換わることになる.
- 函数の引数として行列を渡した場合, 渡された函数は, その行列の成分を書き換えることができる.

```
[0] A = newmat(3,3,[[1,1,1],[x,y],[x^2]]);
[ 1 1 1 ]
[ x y 0 ]
[ x^2 0 0 ]
[1] det(A);
-y*x^2
[2] size(A);
[3,3]
[3] A[1];
[ x y 0 ]
[4] A[1][3];
getarray : Out of range
return to toplevel
```

参照　6.6.1 節「newvect」p.60, 6.6.6 節「size」p.63, 6.6.7 節「det nd_det invmat」p.63.

6.6.6 size

size(*vect*|*mat*)
:: [*vect* の長さ] または [*mat* の行数,*mat* の列数].

return　リスト

vect　ベクトル

mat　行列

- *vect* の長さ, または *mat* の大きさをリストで出力する.
- *vect* の長さは length() で求めることもできる.
- *list* の長さは length()を, 有理式に現れる単項式の数は nmono() を用いる.

```
[0] A = newvect(4);
[ 0 0 0 0 ]
[1] size(A);
[4]
[2] length(A);
4
[3] B = newmat(2,3,[[1,2,3],[4,5,6]]);
[ 1 2 3 ]
[ 4 5 6 ]
[4] size(B);
[2,3]
```

参照　6.5.1 節「car cdr cons append reverse length」p.58, 6.3.6 節「nmono」p.44.

6.6.7 det,invmat

det(*mat*[,*mod*])
nd_det(*mat*[,*mod*])
:: *mat* の行列式を求める.

invmat(*mat*)
:: *mat* の逆行列を求める.

return　det: 式, invmat: リスト

mat　行列

mod　素数

- det および nd_det は行列 *mat* の行列式を求める. invmat は行列 *mat* の逆行列を求める. 逆行列は [分母, 分子] の形で返され, 分母が行列, 分母/分子 が逆行列となる.
- 引数 *mod* がある時, GF(*mod*) 上での行列式を求める.
- 分数なしのガウス消去法によっているため, 多変数多項式を成分とする行列に対しては小行列式展開による方法のほうが効率がよい場合もある.
- nd_det は有理数または有限体上の多項式行列の行列式計算専用である. アルゴリズムはやはり分数なしのガウス消去法だが, データ構造および乗除算の工夫により, 一般に det より高速に計算できる.

```
[91] A=newmat(5,5)$
[92] V=[x,y,z,u,v];
[x,y,z,u,v]
[93] for(I=0;I<5;I++)for(J=0,B=A[I],W=V[I];J<5;J++)B[J]=W^J;
[94] A;
[ 1 x x^2 x^3 x^4 ]
[ 1 y y^2 y^3 y^4 ]
[ 1 z z^2 z^3 z^4 ]
[ 1 u u^2 u^3 u^4 ]
[ 1 v v^2 v^3 v^4 ]
[95] fctr(det(A));
[[1,1],[u-v,1],[-z+v,1],[-z+u,1],[-y+u,1],[y-v,1],[-y+z,1],[-x+u,1],
[-x+z,1],[-x+v,1],[-x+y,1]]
[96] A = newmat(3,3)$
[97] for(I=0;I<3;I++)for(J=0,B=A[I],W=V[I];J<3;J++)B[J]=W^J;
[98] A;
[ 1 x x^2 ]
[ 1 y y^2 ]
[ 1 z z^2 ]
[99] invmat(A);
[[ -z*y^2+z^2*y z*x^2-z^2*x -y*x^2+y^2*x ]
[ y^2-z^2 -x^2+z^2 x^2-y^2 ]
[ -y+z x-z -x+y ],(-y+z)*x^2+(y^2-z^2)*x-z*y^2+z^2*y]
[100] A*B[0];
[ (-y+z)*x^2+(y^2-z^2)*x-z*y^2+z^2*y 0 0 ]
[ 0 (-y+z)*x^2+(y^2-z^2)*x-z*y^2+z^2*y 0 ]
[ 0 0 (-y+z)*x^2+(y^2-z^2)*x-z*y^2+z^2*y ]
[101] map(red,A*B[0]/B[1]);
[ 1 0 0 ]
[ 0 1 0 ]
[ 0 0 1 ]
```

参照　6.6.5 節「newmat」p.62.

6.6.8 qsort

qsort(*array*[,*func*])
:: 一次元配列 *array* をソートする.

return　*array* (入力と同じ; 要素のみ入れ替わる)

array　一次元配列

func　比較用関数

- 一次元配列を quick sort でソートする.
- 比較用関数が指定されていない場合, オブジェクトどうしの比較結果で順序が下のものから順に並べ換えられる.
- 0, 1, -1 を返す 2 引数関数が *func* として与えられた場合, *func*(A,B)=1 の場合に A<B として, 順序が下のものから順に並べ換えられる.
- 配列は新たに生成されず, 引数の配列の要素のみ入れ替わる.

```
[0] qsort(newvect(10,[1,4,6,7,3,2,9,6,0,-1]));
[ -1 0 1 2 3 4 6 6 7 9 ]
[1] def rev(A,B) { return A>B?-1:(A<B?1:0); }
[2] qsort(newvect(10,[1,4,6,7,3,2,9,6,0,-1]),rev);
[ 9 7 6 6 4 3 2 1 0 -1 ]
```

参照　6.3.7 節「ord」p.45, 6.3.2 節「vars」p.43.

6.7 構造体

6.7.1 newstruct

newstruct(*name*)
:: 構造体名が *name* の構造体を生成する.

return　構造体

name　文字列

- 名前が *name* である構造体を生成する.
- あらかじめ, *name* なる構造体が定義されていなければならない.
- 構造体の各メンバは演算子 -> により名前でアクセスする. メンバが構造体の場合, 更に -> による指定を続けることができる.

```
[0] struct list {h,t};
0
[1] A=newstruct(list);
{0,0}
[2] A->t = newstruct(list);
{0,0}
[3] A;
{0,{0,0}}
[4] A->h = 1;
1
[5] A->t->h = 2;
2
[6] A->t->t = 3;
3
[7] A;
{1,{2,3}}
```

参照　6.7.2 節「arfreg」p.65, 4.2.9 節「構造体定義」p.22

6.7.2 arfreg

arfreg(*name*,*add*,*sub*,*mul*,*div*,*pwr*,*chsgn*,*comp*)
:: 構造体に体する基本演算を登録する.

return　1

name　文字列

add sub mul div pwr chsgn comp
　　ユーザ定義関数

- *name* なる構造体型に対する基本演算を登録する.
- 登録したくない基本演算に対しては引数に 0 を与える. これによって一部の演算のみを利用することができる.
- それぞれの関数の仕様は次の通りである.

`add(A,B)` A+B

`sub(A,B)` A-B

`mul(A,B)` A*B

`div(A,B)` A/B

`pwr(A,B)` A^B

`chsgn(A)` -A

`comp(A,B)`
　　1,0,-1 according to the result of a comparison between A and B.

```
% cat test
struct a {id,body}$

def add(A,B)
{
  C = newstruct(a);
  C->id = A->id; C->body = A->body+B->body;
  return C;
}

def sub(A,B)
{
  C = newstruct(a);
  C->id = A->id; C->body = A->body-B->body;
  return C;
}

def mul(A,B)
{
  C = newstruct(a);
  C->id = A->id; C->body = A->body*B->body;
  return C;
}

def div(A,B)
{
  C = newstruct(a);
  C->id = A->id; C->body = A->body/B->body;
  return C;
}
```

```
def pwr(A,B)
{
  C = newstruct(a);
  C->id = A->id; C->body = A->body^B;
  return C;
}

def chsgn(A)
{
  C = newstruct(a);
  C->id = A->id; C->body = -A->body;
  return C;
}

def comp(A,B)
{
  if ( A->body > B->body )
    return 1;
  else if ( A->body < B->body )
    return -1;
  else
    return 0;
}

arfreg("a",add,sub,mul,div,pwr,chsgn,comp)$
end$
% asir
This is Risa/Asir, Version 20000908.
Copyright (C) FUJITSU LABORATORIES LIMITED.
1994-2000. All rights reserved.
[0] load("./test")$
[11] A=newstruct(a);
{0,0}
[12] B=newstruct(a);
{0,0}
[13] A->body = 3;
3
[14] B->body = 4;
4
[15] A*B;
{0,12}
```

参照　　6.7.1 節「newstruct」p.65, 4.2.9 節「構造体定義」p.22

6.7.3 struct_type

struct_type(*name*|*object*)
　　:: 構造体の識別番号を取得する.

return　　整数

name　　文字列

object　　構造体

- 名前が *name* である構造体, または *object* の指す構造体の識別番号を取得する. エラーのときは -1 を返す.

```
[10] struct list {h,t};
0
[11] A=newstruct(list);
{0,0}
[12] struct_type(A);
3
[13] struct_type("list");
3
```

参照　　6.7.1 節「newstruct」p.65, 4.2.9 節「構造体定義」p.22

6.8 型を求める函数

6.8.1 type

type(*obj*) :: *obj* の 型 (整数) を返す.

return　　整数

obj　　任意

- *obj* の 型の値は次の通り.

0	0
1	数
2	多項式 (数でない)
3	有理式 (多項式でない)
4	リスト
5	ベクトル
6	行列
7	文字列
8	構造体
9	分散表現多項式
10	32bit 符号なし整数
11	エラーオブジェクト
12	GF(2) 上の行列
13	MATHCAP オブジェクト
14	一階述語論理式

-1　　VOID オブジェクト

- 数の型を求めるには, ntype を用いる. 不定元の型を求めるには, vtype を用いる.

参照　　6.8.2 節「ntype」p.69, 6.8.3 節「vtype」p.69.

6.8.2 ntype

ntype(*num*)
:: *num* (数) の 型 (整数) を返す.

return　　自然数

obj　　数

- 数の型の値は次の通り.

0　　有理数

1　　倍精度浮動小数

2　　代数的数

3　　任意精度浮動小数 (**bigfloat**)

4　　複素数

5　　有限体の元

6　　大標数素体の元

7　　標数 2 の有限体の元

- newalg(x^2+1) で生成した数と, 虚数単位 @i は, 異なるものとして扱われる.
- 代数的数に関しては, 第 9 章「代数的数に関する演算」p.140 を参照.

```
[0] [10/37,ntype(10/37)];
[10/37,0]
[1] [10.0/37.0,ntype(10.0/37.0)];
[0.27027,1]
[2] [newalg(x^2+1)+1,ntype(newalg(x^2+1)+1)];
[(#0+1),2]
[3] [eval(sin(@pi/6)),ntype(eval(sin(@pi/6)))];
[0.49999999999999999991,3]
[4] [@i+1,ntype(@i+1)];
[(1+1*@i),4]
```

参照　　6.8.1 節「type」p.68.

6.8.3 vtype

vtype(*var*)
:: *var* (不定元) の 型 (整数) を返す.

return　　整数

var　　不定元

- *var* (不定元) の型の値は次の通り. 詳細は 3.3 節「不定元の型」p.14 を参照.

0　一般不定元 (a,b,x,afo,bfo,...,etc)

1　uc() で生成された不定元 (_0, _1, _2, ... etc)

2　函数形式 (sin(x), log(a+1), acosh(1), @pi, @e, ... etc)

3　函数子 (組み込み函数子, ユーザ定義函数子, 初等函数子 : sin, log, ... etc)

- a(); を実行 (通常ではエラー) しただけでも ntype(a) は 3 となる. すなわち a はユーザ定義函数子とみなされる.
- ユーザが函数形式を定義するためにはキーワード function を用いる.
- @pi, @e は不定元として扱われるが, eval(), pari() においては数として扱われる.

参照　6.8.1 節「type」p.68, 6.8.2 節「ntype」p.69, 6.3.3 節「uc」p.43.

6.9 函数に対する操作

6.9.1 call

call(*name*, *args*)
:: 函数 *name* を呼び出す.

return　函数 name() の返り値.

name　函数名を表す不定元 (函数子とは限らない)

args　引数のリスト

- 函数子に関しては, vtype() を参照.
- 引数の個数があらかじめ分かっているときは (*name)() の形式, 分からないときは call() を使うとよい.

```
[0] A=igcd;
igcd
[1] call(A,[4,6]);
2
[2] (*A)(4,6);
2
```

参照　6.8.3 節「vtype」p.69.

6.9.2 functor, args, funargs

functor(*func*)
:: *func* の函数子を取り出す.

args(*func*)
:: *func* の引数リストを取り出す.

funargs(*func*)
:: cons(functor(*func*),args(*func*)) を返す.

return　functor() : 不定元, args(), funargs() : リスト

func　函数形式を表す不定元

- 函数形式に関しては, vtype() を参照.
- 函数形式 *func* の函数子, 引数リストを取り出す.
- 逆に, 取り出した函数子を値に持つプログラム変数を F とすれば (*F)(x) で x を引数とする函数呼び出しまたは函数形式が入力できる.

```
[0] functor(sin(x));
sin
[0] args(sin(x));
[x]
[0] funargs(sin(3*cos(y)));
[sin,3*cos(y)]
[1] for (L=[sin,cos,tan];L!=[];L=cdr(L)) {A=car(L);
print(eval((*A)(@pi/3)));}
0.86602540349122136831
0.5000000002
1.7320508058
```

参照　6.8.3 節「vtype」p.69.

6.10 文字列に関する演算

6.10.1 rtostr

rtostr(*obj*)
:: *obj* を文字列に変える.

return　文字列

obj　任意

- 任意のオブジェクト *obj* を文字列に変える.
- 整数などを文字列に変換して変数名と結合することにより, 添字付きの不定元を生成する場合に多く用いられる.
- 逆に, 文字列を不定元に変換する時には, strtov を用いる.

```
[0] A=afo;
afo
[1] type(A);
2
[2] B=rtostr(A);
afo
[3] type(B);
7
[4] B+"1";
afo1
```

参照　6.10.2 節「strtov」p.72, 6.8.1 節「type」p.68.

6.10.2 strtov

strtov(*str*)
:: *str* (文字列) を不定元に変える.

return　不定元

str　不定元として変換可能な文字列

- 不定元として変換可能な文字列を不定元に変える.
- 不定元として変換可能な文字列とは, 英小文字で始まり, 英字, 数字および記号 _ で作られる文字列である.
- rtostr() と組合せて, プログラム中で自動的に不定元を生成したい時に用いられる.

```
[0] A="afo";
afo
[1] for (I=0;I<3;I++) {B=strtov(A+rtostr(I)); print([B,type(B)]);}
[afo0,2]
[afo1,2]
[afo2,2]
```

参照　6.10.1 節「rtostr」p.71, 6.8.1 節「type」p.68, 6.3.3 節「uc」p.43.

6.10.3 eval_str

eval_str(*str*)
:: *str* (文字列) を評価する.

return　オブジェクト

str　**Asir** の parser が受理可能な文字列

- **Asir** の parser が受理可能な文字列を評価してその結果を返す.
- 評価可能な文字列は, 式を表すものに限る.
- 論理的には rtostr() の逆関数となる.

```
[0] eval_str("1+2");
3
[1] fctr(eval_str(rtostr((x+y)^10)));
[[1,1],[x+y,10]]
```

参照　6.10.1 節「rtostr」p.71

6.10.4 strtoascii, asciitostr

strtoascii(*str*)
:: 文字列をアスキーコードで表す.

asciitostr(*list*)
:: アスキーコードの列を文字列に変換する.

return　strtoascii():リスト; asciitostr():文字列

str　文字列

list　　1 以上 256 未満の整数からなるリスト

- strtoascii() は文字列を整数のリストに変換する．各整数は文字列のアスキーコードを表す．
- asciitostr() は asciitostr() の逆関数である．

```
[0] strtoascii("abcxyz");
[97,98,99,120,121,122]
[1] asciitostr(@);
abcxyz
[2] asciitostr([256]);
asciitostr : argument out of range
return to toplevel
```

6.10.5 str_len, str_chr, sub_str

str_len(*str*)
:: 文字列の長さを返す．

str_chr(*str*,*start*,*c*)
:: 文字が最初に現れる位置を返す．

sub_str(*str*,*start*,*end*)
:: 部分文字列を返す．

return　　str_len(), str_chr():整数; sub_str():文字列

str c　　文字列

start end　　非負整数

- str_len() は文字列の長さを返す．
- str_chr() は *str* の *start* 番目の文字からスキャンして最初に *c* の最初の文字が現れた位置を返す．文字列の先頭は 0 番目とする．指定された文字が現れない場合には -1 を返す．
- sub_str() は, *str* の *start* 番目から *end* 番目までの部分文字列を生成し返す．

```
[185] Line="123 456 (x+y)^3";
123 456 (x+y)^3
[186] Sp1 = str_chr(Line,0," ");
3
[187] D0 = eval_str(sub_str(Line,0,Sp1-1));
123
[188] Sp2 = str_chr(Line,Sp1+1," ");
7
[189] D1 = eval_str(sub_str(Line,Sp1+1,Sp2-1));
456
[190] C = eval_str(sub_str(Line,Sp2+1,str_len(Line)-1));
x^3+3*y*x^2+3*y^2*x+y^3
```

6.11 入出力

6.11.1 end, quit

`end, quit`
:: 現在読み込み中のファイルを閉じる. トップレベルにおいてはセッションを終了することになる.

- `end`, `quit` ともに無引数の函数であるが, '`()`' なしで呼び出すことができる. いずれも現在読み込み中のファイルを閉じる. これは, トップレベルにおいてはセッションを終了させることになる.
- ファイルの場合, ファイルの終端まで読めば, 自動的にファイルは閉じられるが, トップレベルの場合プロンプトが出ないまま, 入力待ちになるので, ファイルの終端には `end$` を書くのが望ましい.

```
[6] quit;
%
```

参照　6.11.2 節「load」p.74.

6.11.2 load

`load("`*filename*`")`
:: *filename* を読み込む.

return　(1|0)

filename　ファイル名 (パス名)

- 実際のプログラムの書き方は, 第 4 章「ユーザ言語 Asir」p.16 参照. テキストファイルを読み込む場合, `cpp` を通すので, C のプログラム同様 `#include`, `#define` を使うことができる.
- 指定したファイルが存在した時には 1 を返し, 存在しなかった時は 0 を返す.
- ファイル名が '`/`' で始まる場合は絶対パス, '`.`' で始まる場合はカレントディレクトリからの相対パスと見なされる. それ以外の場合, 環境変数 `ASIRLOADPATH` に設定されているディレクトリを左から順にサーチする. それらに該当するファイルが存在しない場合, 標準ライブラリディレクトリ (あるいは環境変数 `ASIR_LIBDIR` に設定されているディレクトリ) もサーチする. Windows 版の場合, `ASIR_LIBDIR` が設定されていない場合には, `get_rootdir()/lib` をサーチする.
- 読み込むファイルの最後に, `end$` がないと `load()` 終了後にプロンプトがでないが, 実際には入力を受け付ける. しかし, 混乱を招くおそれがあるのでファイルの最後に `end$` を書いておくことが望ましい. (`end;` でもよいが, `end` が返す値 0 が表示されるため, `end$` をお勧めする.)
- Windows 版もディレクトリのセパレータとして '`/`' を用いる.

参照　6.11.1 節「end quit」p.74, 6.11.3 節「which」p.74, 6.14.16 節「get_rootdir」p.90.

6.11.3 which

`which("`*filename*`")`
:: 引数 *filename* に対し, `load()` が読み込むパス名を返す.

return　　パス名

filename　　ファイル名 (パス名) または 0

- `load()` がファイルをサーチする手順に従ってサーチし, ファイルが存在する場合にはパス名を文字列として, 存在しない場合には 0 を返す.
- サーチの手順については `load()` を参照.
- Windows 版もディレクトリのセパレータとして '/' を用いる.

```
[0] which("gr");
./gb/gr
[1] which("/usr/local/lib/gr");
0
[2] which("/usr/local/lib/asir/gr");
/usr/local/lib/asir/gr
```

参照　　6.11.2 節「load」p.74.

6.11.4 output

`output(["`*filename*`"])`
:: 以降の出力先を *filename* または標準出力に切替える.

return　　1

filename　　ファイル名

- **Asir** の出力を標準出力から, ファイルへの出力に切替える. なお, ファイル出力の間は, 標準出力にはキーボードからの入力以外, 出力されない.
- 別のファイル出力に切替える時には, 再び `output("`*filename*`")` を実行する. 又, ファイル出力を終了し標準出力に戻りたい時には, 引数なしで `output()` を実行する.
- 指定したファイル *filename* が存在した時は, そのファイルの末尾に追書きされ, 存在しなかった時には, 新たにファイルを作成し, そこに書き込まれる.
- ファイルネームを "" ダブルクォートなしで指定をしたり, ユーザが, 書き込めないファイルを指定したりすると, エラーによりトップレベルに戻る.
- 入力したものも込めてファイルに出力したい場合には, `ctrl("echo",1)` を実行した後でファイル出力に切替えれば良い.
- 計算時間など, 標準エラー出力に書き出されるものはファイルには書き出されない.
- 函数形式, 未定係数 (`vtype()` 参照) を含まない数式のファイルへの読み書きは, `bload()`, `bsave()` を使うのが, 時間, 空間ともに効率がよい.
- Windows 版もディレクトリのセパレータとして '/' を用いる.

```
[83] output("afo");
fctr(x^2-y^2);
print("afo");
output();
1
[87] quit;
% cat afo
1
[84] [[1,1],[x+y,1],[x-y,1]]
```

```
[85] afo
0
[86]
```

参照　6.14.1 節「ctrl」p.82, 6.11.5 節「bsave bload」p.76.

6.11.5 bsave, bload

bsave(*obj*,"*filename*")
:: *filename* に *obj* をバイナリ形式で書き込む.

bload("*filename*")
:: *filename* から数式をバイナリ形式で読み込む.

return　bsave() : 1, bload() : 読み込んだ数式

obj　函数形式, 未定係数を含まない任意の数式

filename　ファイル名

- bsave() は内部形式をほぼそのままバイナリ形式でファイルに書き込む. bload() は, bsave() で書き込んだ数式を読み込んで内部形式に変換する. 現在のインプリメンテーションの制限により, 函数形式, 未定係数 (vtype() 参照) を含まないリスト, 配列などを含む任意の数式をファイルに保存することができる.
- output() などで保存した場合, 読み込み時にパーザが起動されるが, bsave() で保存したものを bload() で読む場合, 直接内部形式が構成できるため, 時間的, 空間的に効率がよい.
- 多項式の場合, 書き込み時と読み込み時で変数順序が異なる場合があるが, その場合には, 自動的に現在の変数順序における内部形式に変換される.
- Windows 版もディレクトリのセパレータとして '/' を用いる.

```
[0] A=(x+y+z+u+v+w)^20$
[1] bsave(A,"afo");
1
[2] B = bload("afo")$
[3] A == B;
1
[4] X=(x+y)^2;
x^2+2*y*x+y^2
[5] bsave(X,"afo")$
[6] quit;
% asir
[0] ord([y,x])$
[1] bload("afo");
y^2+2*x*y+x^2
```

参照　6.11.4 節「output」p.75.

6.11.6 bload27

bload27("*filename*")
:: 旧版で作られた bsave file の読み込み

return　　読み込んだ数式

filename　　ファイル名

- 旧版では, 多倍長整数が, 1 ワード 27 bit で表現されていたが, 新版では 1 ワード 32 bit に変更された. このため, 旧版で `bsave` されたバイナリファイルはそのままでは読み込めない. このようなファイルを読み込むために `bload27` を用いる.
- Windows 版もディレクトリのセパレータとして '/' を用いる.

参照　　6.11.5 節「`bsave bload`」p.76.

6.11.7 print

`print(`*obj* `[,`*nl*`])`
:: *obj* を表示する.

return　　0

obj　　任意

nl　　フラグ (任意)

- *obj* を評価して表示する.
- 第 2 引数がないか, または 0, 2 以外の場合, 改行する. 第 2 引数が 1 の場合, 改行せず, 出力はバッファに書き込まれ, バッファはフラッシュされない. 第 2 引数が 2 の場合, 改行しないがバッファはフラッシュされる.
- この函数の戻り値は 0 であるから, `print();` で実行すると, 出力の後に 0 が返される. `print()$` とすれば, 最後の 0 は出力されない.
- 複数の *obj* を同時に出力したい時は *obj* をリストにするとよい.

```
[8] def cat(L) { while ( L != [] ) { print(car(L),0); L = cdr(L);}
print(""); }
[9] cat([xyz,123,"gahaha"])$
xyz123gahaha
```

6.11.8 access

`access(`*file*`)`
:: *file* の存在をテストする.

return　　(1|0)

file　　ファイル名

- *file* が存在すれば 1, 存在しなければ 0 を返す.

6.11.9 remove_file

`remove_file(`*file*`)`
:: *file* を消去する.

return　　1

file　　ファイル名

6.11.10 open_file, close_file, get_line, get_byte, put_byte, purge_stdin

`open_file("`*filename*`"[,"`*mode*`"])`
:: *filename* をオープンする.

`close_file(`*num*`)`
:: 識別子 *num* のファイルをクローズする.

`get_line([`*num*`])`
:: 識別子 *num* のファイルから 1 行読む.

`get_byte(`*num*`)`
:: 識別子 *num* のファイルから 1 バイト読む.

`put_byte(`*num*`,`*c*`)`
:: 識別子 *num* のファイルに 1 バイト *c* を書く.

`purge_stdin()`
:: 標準入力のバッファをクリアする.

return　`open_file()`：整数 (識別子); `close_file()`：1; `get_line()`：文字列; `get_byte()`, `put_byte()`：整数

filename　ファイル名 (パス名)

mode　文字列

num　非負整数 (ファイル識別子)

- `open_file()` はファイルをオープンする. *mode* 指定がない場合読み出し用, *mode* 指定がある場合には, C の標準入出力関数 `fopen()` に対するモード指定とみなす. たとえば新規書き込み用の場合 `"w"`, 末尾追加の場合 `"a"` など. 成功した場合, ファイル識別子として非負整数を返す. 失敗の場合エラーとなる. 不要になったファイルは `close_file()` でクローズする. 特別なファイル名 unix://stdin, unix://stdout, unix://stderr を与えるとそれぞれ標準入力, 標準出力, 標準エラー出力をオープンする. この場合モード指定は無視される.
- `get_line()` は現在オープンしているファイルから 1 行読み, 文字列として返す. 引数がない場合, 標準入力から 1 行読む.
- `get_byte()` は現在オープンしているファイルから 1 バイト読み整数として返す.
- `put_byte()` は現在オープンしているファイルに 1 バイト書き, そのバイトを整数として返す.
- ファイルの終りまで読んだ後に `get_line()` が呼ばれた場合, 整数の 0 を返す.
- 読み出した文字列は, 必要があれば `sub_str()` などの文字列処理関数で加工したのち `eval_str()` により内部形式に変換できる.
- `purge_stdin()` は, 標準入力バッファを空にする. 関数内で `get_line()` により標準入力から文字列を受け取る場合, 既にバッファ内に存在する文字列による誤動作を防ぐためにあらかじめ呼び出す.

```
[185] Id = open_file("test");
0
[186] get_line(Id);
12345
```

```
[187] get_line(Id);
67890

[188] get_line(Id);
0
[189] type(@@);
0
[190] close_file(Id);
1
[191] open_file("test");
1
[192] get_line(1);
12345

[193] get_byte(1);
54                    /* the ASCII code of '6' */
[194] get_line(1);
7890                  /* the rest of the last line */
[195] def test() { return get_line(); }
[196] def test1() { purge_stdin(); return get_line(); }
[197] test();
                      /* a remaining newline character has been read */
                      /* returns immediately */
[198] test1();
123;                  /* input from a keyboard */
123;                  /* returned value */

[199]
```

参照　6.10.3 節「eval_str」p.72, 6.10.5 節「str_len str_chr sub_str」p.73.

6.12 モジュールに対する操作

6.12.1 module_list

module_list()
:: 定義済みのモジュールのリストを得る.

return　定義済みのモジュールのリスト.

```
[1040] module_list();
[gr,primdec,bfct,sm1,gnuplot,tigers,phc]
```

参照　4.2.13 節「モジュール」p.25 を参照

6.12.2 module_definedp

module_definedp(*name*)
:: モジュール *name* の存在をテストする.

return　(1|0)

name　モジュール名

- モジュール *name* が存在すれば 1, 存在しなければ 0 を返す.

```
[100] module_definedp("gr");
1
```

参照　6.12.1 節「module_list」p.79, 4.2.13 節「モジュール」p.25 を参照.

6.12.3 remove_module

remove_module(*name*)
:: モジュール *name* を削除する.

return　(1|0)

name　モジュール名

- 削除に成功すれば 1, 失敗すれば 0 を返す.

```
[100] remove_module("gr");
1
```

参照　4.2.13 節「モジュール」p.25 を参照.

6.13 数値函数

6.13.1 dacos, dasin, datan, dcos, dsin, dtan

dacos(*num*)
:: 函数値 Arccos(*num*) を求める.

dasin(*num*)
:: 函数値 Arcsin(*num*) を求める.

datan(*num*)
:: 函数値 Arctan(*num*) を求める.

dcos(*num*)
:: 函数値 cos(*num*) を求める.

dsin(*num*)
:: 函数値 sin(*num*) を求める.

dtan(*num*)
:: 函数値 tan(*num*) を求める.

return　倍精度浮動小数

num　数

- 三角関数、逆三角関数を数値的に計算する.
- これらの函数は C 言語の標準数学ライブラリを用いる. したがって, 計算結果はオペレーティングシステムとコンパイラに依存する.

- 複素数に対しては正しくない結果を返すので注意しなければならない.
- @pi などのシンボルを引数に与えることはできない.

```
[0] 4*datan(1);
3.14159
```

6.13.2 dabs, dexp, dlog, dsqrt

`dabs(`*num*`)`
:: 絶対値 |*num*| を求める.

`dexp(`*num*`)`
:: 函数値 exp(*num*) を求める.

`dlog(`*num*`)`
:: 対数値 log(*num*) を求める.

`dsqrt(`*num*`)`
:: 平行根 sqrt(*num*) を求める.

return　倍精度浮動小数

num　数

- 初等函数を数値的に計算する.
- これらの函数は C 言語の標準数学ライブラリを用いる. したがって, 計算結果はオペレーティングシステムとコンパイラに依存する.
- `dabs()`と `dsqrt()`を除き、複素数に対しては正しくない結果を返すので注意しなければならない.
- @pi などのシンボルを引数に与えることはできない.

```
[0] dexp(1);
2.71828
```

6.13.3 ceil, floor, rint, dceil, dfloor, drint

`ceil(`*num*`)`
`dceil(`*num*`)`
:: *num* より大きい最小の整数を求める.

`floor(`*num*`)`
`dfloor(`*num*`)`
:: *num* より小さい最大の整数を求める.

`rint(`*num*`)`
`drint(`*num*`)`
:: *num* を整数にまるめる.

return　整数

num　数

- `dceil`, `dfloor`, `drint` は `ceil`, `floor`, `rint` の別名である.
- 浮動小数を整数にまるめる方法は, オペレーティングシステムとコンパイラに依存する.

- 複素数に対しては正しくない結果を返すので注意しなければならない.
- @pi などのシンボルを引数に与えることはできない.

```
[0] dceil(1.1);
1
```

6.14 その他

6.14.1 ctrl

`ctrl("`*switch*`"[,`*obj*`])`
:: 環境設定

return　設定されている値

switch　スイッチ名

obj　パラメタ

- **Asir** の実行環境の設定変更, 参照を行う.
- *switch* のみの場合, そのスイッチの現在の状態を返す.
- *obj* が与えられているとき, その値を設定する.
- スイッチは文字列として入力する. すなわちダブルクォートで囲む.
- スイッチは次の通り. 以下で, on は 1, off は 0 を意味する.

`cputime`　on の時 CPU time および GC time を表示, off の時 表示しない. `cputime()` を参照. `ctrl("cputime",`*onoff*`)` は `cputime(`*onoff*`)` と同じである.

`nez`　EZGCD のアルゴリズムの切替え. デフォルトで 1 であり, とくに切替える必要はない.

`echo`　on の時は標準入力を繰り返して出力し, off の時は標準入力を繰り返さない. `output` コマンドを用いる際に有効である.

`bigfloat`　on の時, 入力された浮動小数は **bigfloat** に変換され, 浮動小数演算は **PARI** (6.1.14 節「pari」p.39) により行われる. デフォルトの有効桁数は 9 桁である. 有効桁数を増やしたい時には `setprec()` を用いる. off の時, 入力された浮動小数は, 倍精度浮動小数に変換される.

`adj`　ガーベッジコレクションの頻度の変更. 1 以上の有理数が指定できる. デフォルト値は 3. 1 に近い程, ガーベッジコレクションせずにヒープを大きくとるようになる. 整数値はコマンドラインで指定できる. 2.4 節「コマンドラインオプション」p.5 を参照.

`verbose`　on の時, 函数の再定義時にメッセージを表示する.

`quiet_mode`
1 のとき, 起動時に著作権表示を行わない. 2.4 節「コマンドラインオプション」p.5 を参照.

`prompt`　0 のときプロンプトを表示しない. 1 のとき標準プロンプトを表示. C スタイルのフォマット文字列をもちいるとユーザ定義のプロンプト. 例 (asirgui では不可): `ctrl("prompt","\033[32m[%d]:= \033[0m")`

`hex`　1 のとき, 整数は 0x で始まる 16 進数として表示される. -1 のとき, 16 進数は, 間に '|' をはさんで 8 桁ごとに区切って表示される.

`real_digit`
倍精度浮動小数の表示の桁数を指定する.

`double_output`
1 のとき, 倍精度浮動小数はつねに ddd.ddd の形で表示される.

`fortran_output`
1 のとき, 多項式の表示が FORTRAN スタイルになる. すなわち冪が '^' の代わりに '**' で表される. (デフォルト値は 0.)

`ox_batch`　1 のとき, 送信バッファがいっぱいになった時のみ自動的に flush. 0 のとき, データ, コマンド送信毎に flush. (デフォルト値は 0.) 第 7 章「分散計算」p.91 を参照.

`ox_check`　1 のとき, 送信データを相手プロセスが受け取れるかどうかチェックする. 0 のときしない. (デフォルト値は 1.) 第 7 章「分散計算」p.91 を参照.

`ox_exchange_mathcap`
1 のとき, OX server との接続開始時に, 自動的に mathcap の交換を行う. (デフォルト値は 1.) 第 7 章「分散計算」p.91 を参照.

参照　6.14.6 節「`cputime tstart tstop`」p.85, 6.11.4 節「`output`」p.75, 6.1.14 節「`pari`」p.39, 6.1.15 節「`setprec`」p.40, ⟨undefined⟩「`eval deval`」p.⟨undefined⟩.

6.14.2 debug

`debug`　:: デバッグモードに入る.

- `debug` は無引数の函数であるが, '`()`' なしで呼び出せる.
- デバッグモードに入るとプロンプトが `(debug)` となり, コマンド受け付け状態となる. `quit` を入力するとデバッガから抜ける.
- デバッグモードについての詳細は 第 5 章「デバッガ」p.29 を参照.

```
[1] debug;
(debug) quit
0
[2]
```

6.14.3 error

`error(`*message*`)`
:: プログラム中で強制的にエラーを発生させる.

message　文字列

- 一般に, 引数の間違いなど, 続行不可能なエラーが組み込み函数において発生した時, トップレベルに戻る前に, 可能ならばそのエラーの時点でデバッグモードに入る. `error()` は, ユーザ函数の内部でこの動作と同様の動作を行わせるための函数である.
- 引数は, `error()` が呼び出される際に表示されるメッセージで, 文字列である.

- ユーザ函数において, 変数をチェックして, あり得ない値の場合に error() を呼び出すようにしておけば, その時点で自動的にデバッグモードに入れる.

```
% cat mod3
def mod3(A) {
        if ( type(A) >= 2 )
                error("invalid argument");
        else
                return A % 3;
}
end$
% asir
[0] load("mod3");
1
[3] mod3(5);
2
[4] mod3(x);
invalid argument
stopped in mod3 at line 3 in file "./mod3"
3                       error("invalid argument");
(debug) print A
A = x
(debug) quit
return to toplevel
[4]
```

参照　6.14.2 節「debug」p.83.

6.14.4 help

help(["*function*"])
:: 函数の説明を表示する.

return　0

function　函数名

- 無引数の時, 最小限の説明が表示される. 函数名が引数として与えられたとき, 標準ライブラリディレクトリにある 'help' というディレクトリに同名のファイルがあれば, 環境変数 PAGER に設定されているコマンド, あるいは 'more' を呼び出してそのファイルを表示する.
- 環境変数 LANG が設定されている場合, その値が "japan" または"ja_JP" で始まるなら, 'help' の代わりに 'help-jp' にあるファイルが表示される. そうでない場合, 'help-eg' にあるファイルが表示される.
- Windows 版では, コマンドラインからのヘルプ呼び出しは未サポートだが, メニューから HTML 形式のものを呼び出し用いることができる.

6.14.5 time

time()　:: セッション開始から現在までの CPU 時間および GC 時間を表示する

return　リスト

- CPU 時間および GC 時間の表示に関するコマンドである.
- GC 時間とは, ガーベジコレクタにより消費されたと見なされる時間, CPU 時間は, 全体の CPU 時間から GC 時間を引いた残りで, 単位は秒である.
- time() は引数なしで, セッション開始から現在までの CPU 時間, GC 時間, 現在までに要求されたメモリののべ容量, およびセッション開始から現在までの経過時間の表示をする. すなわち, [CPU 時間 (秒), GC 時間 (秒), メモリ量 (ワード), 経過時間 (秒)] なるリストを返す. 1 ワードは通常 4 バイトである.
- 計算の実行開始時, 終了時の time() から, その計算に対する CPU 時間, GC 時間がわかる.
- メモリ量は多倍長数ではないため, ある値を越えると無意味な値となるためあくまでも目安として用いるべきである.
- ctrl() や cputime() により cputime スイッチが on になっている場合には, トップレベルの文を一つの単位として, その実行時間が表示される. しかし, プログラムの内部などで, 特定の計算に対する計算時間を知りたい時には, time() などを使う必要がある.
- getrusage() が使える UNIX 上では time() は信頼性のある値を返すが, Windows 95, 98 上では時刻を用いるほか方法がないため経過時間そのものが表示される. よって, 待ち状態があると, それも経過時間に加算される.

```
[72] T0=time();
[2.390885,0.484358,46560,9.157768]
[73] G=hgr(katsura(4),[u4,u3,u2,u1,u0],2)$
[74] T1=time();
[8.968048,7.705907,1514833,63.359717]
[75] ["CPU",T1[0]-T0[0],"GC",T1[1]-T0[1]];
[CPU,6.577163,GC,7.221549]
```

参照　6.14.6 節「cputime tstart tstop」p.85, 6.14.8 節「currenttime」p.86.

6.14.6 cputime, tstart, tstop

cputime(*onoff*)
:: 引数が 0 ならば cputime の表示を止める. それ以外ならば表示を行う.

tstart()　:: CPU time 計測開始.

tstop()　:: CPU time 計測終了および表示.

return　0

onoff　フラグ (任意)

- cputime() は, 引数が 0 ならば CPU time の表示を止める. それ以外ならば表示を行う.
- tsart は引数なし, '()' なしで, CPU time 計測を開始する.
- tstop は引数なし, '()' なしで, CPU time 計測を終了, および表示する.
- cputime(*onoff*) は ctrl("cputime",*onoff*) と同じである.
- tstart, tstop は, 入れ子にして使われることは想定していないため, そのような可能性がある場合には, time() による計測を行う必要がある.
- cputime() による on, off は, 単に表示の on, off であり, トップレベルの一つの文に対する計測は常に行われている. よって, 計算を始めてからでも, 計算終了前にデバッガに入って cputime(1) を実行させれば計算時間は表示される.

```
[49] tstart$
[50] fctr(x^10-y^10);
[[1,1],[x+y,1],[x^4-y*x^3+y^2*x^2-y^3*x+y^4,1],[x-y,1],
[x^4+y*x^3+y^2*x^2+y^3*x+y^4,1]]
[51] tstop$
80msec + gc : 40msec
```

参照　6.14.5 節「time」p.85, 6.14.8 節「currenttime」p.86, 6.14.1 節「ctrl」p.82.

6.14.7 timer

timer(*interval*,*expr*,*val*)
:: 制限時間つきで計算を実行する.

return　結果

interval　制限時間 (秒)

expr　計算する式

val　タイマによる中断時の戻り値

- timer() は, 時間を指定して計算を実行する. 指定時間内に計算が完了した場合その値を返す. 指定時間内に計算が完了しなかった場合, 第 3 引数を返す.
- 第 3 引数の値は, 計算が完了した場合の値と区別できる必要がある.

```
[0] load("cyclic");
1
[10] timer(10,dp_gr_main(cyclic(7),[c0,c1,c2,c3,c4,c5,c6],1,1,0),0);
interval timer expired (VTALRM)
0
[11]
```

6.14.8 currenttime

currenttime()
:: 現在時刻を取得

return　1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数.

- currenttime() は現在時刻を返す. UNIX の場合, time(3) を呼んでいるだけである.

```
[0] currenttime();
1071639228
[1]
```

6.14.9 sleep

sleep(*interval*)
:: プロセスの実行を停止

return　1

interval　停止時間 (マイクロ秒)

- sleep() は, プロセスの実行を停止する. UNIX の場合, usleep を呼んでいるだけである.

```
[0] sleep(1000);
1
[1]
```

6.14.10 heap

`heap()`　:: 現在のヒープの大きさを返す. (単位:バイト)

return　自然数

- 現在のヒープの大きさ (単位 : バイト) を返す. ヒープとは, **Asir** のさまざまな数式や, ユーザプログラムなどがおかれるメモリの領域で, ガーベジコレクタにより管理されている. プログラムの動作中は, ヒープの大きさは単調非減少であり, 実メモリの量をこえて大きくなった場合には, OS によるスワップエリアへの読み書きがほとんどの計算時間を占めることになる.
- 実メモリが少ない場合には, 起動時の `-adj` オプションにより, GC 主体の設定を行っておく必要がある.

```
% asir -adj 16
[0] load("fctrdata")$
0
[97] cputime(1)$
0msec
[98] heap();
524288
0msec
[99] fctr(Wang[8])$
3.190sec + gc : 3.420sec
[100] heap();
1118208
0msec
[101] quit;
% asir
[0] load("fctrdata")$
0
[97] cputime(1)$
0msec
[98] heap();
827392
0msec
[99] fctr(Wang[8])$
3.000sec + gc : 1.180sec
[100] heap();
1626112
0msec
[101] quit;
```

参照　2.4 節「コマンドラインオプション」p.5.

6.14.11 version

`version()`
:: **Asir** のバージョンを返す.

return　自然数

- **Asir** のバージョンを自然数で返す.

```
[0] version();
991214
```

6.14.12 shell

`shell(`*command*`)`
:: *command* をシェルコマンドとして実行する.

return　自然数

command　文字列

- *command* を C の `system()` 函数によりシェルコマンドとして実行する. シェルの終了ステータスを返す.

```
[0] shell("ls");
alg             da              katsura         ralg            suit
algt            defs.h          kimura          ratint          test
alpi            edet            kimura3         robot           texput.log
asir.o          fee             mfee            sasa            wang
asir_symtab     gr              mksym           shira           wang_data
base            gr.h            mp              snf1            wt
bgk             help            msubst          solve
chou            hom             p               sp
const           ifplot          proot           strum
cyclic          is              r               sugar
0
[1]
```

6.14.13 map

`map(`*function*`,`*arg0*`,`*arg1*`,...)`
:: リスト, 配列の各要素に函数を適用する.

return　*arg0* と同じ型のオブジェクト

function　函数名

arg0　リスト, ベクトル, 行列

arg1 ...　任意 (残りの引数)

- *arg0* の各要素を最初の引数, *arg1* 以下の残りの引数として函数 *function* を呼び出し, *arg0* の対応する要素の位置に函数呼び出しの結果が入った同じ型のオブジェクトを生成して返す.
- *function* は, ダブルクォートのない函数名を用いる.
- *function* にプログラム変数は使えない.

- *arg0* がリスト, ベクトル, 行列以外の場合, 単に *arg0*, *arg1*, ... を引数として *function* を呼び出しその結果を返す.
- `map` の引数 *function* で与えられる函数は, 内部的にも函数として実装されていなければならない. そうでなければ parse error になる. 例えば `map` 自身や `car`, `cdr` などは内部的には函数ではなく, Asir の文法におけるキーワードとして実装されている. したがって `map` の引数に `map` をとることはできない.

```
[82] def afo(X) { return X^3; }
[83] map(afo,[1,2,3]);
[1,8,27]
```

6.14.14 flist

`flist()`　:: 現在定義されている函数名を文字列リストとして返す.

return　文字列のリスト

- 現在定義されている組み込み函数, ユーザ定義函数の函数名を文字列リストとして返す.
- システム函数の後にユーザ定義函数が続く.

```
[77] flist();
[defpoly,newalg,mainalg,algtorat,rattoalg,getalg,alg,algv,...]
```

6.14.15 delete_history

`delete_history([`*index*`])`
:: ヒストリを消去する.

return　0

index　消去したいヒストリの番号

- 引数がないとき, これまで計算したヒストリを全て消去する.
- 引数があるとき, その番号の結果のみ消去する.
- ここでヒストリとは, 番号つきのプロンプトに対しての入力を評価して得られた式で, この式は `@`*number* により取り出すことができる. このことは, ガーベッジコレクションの際にもこの式が生き残ることを意味する.
- 大きな式がヒストリとして残った場合, 以降のメモリ管理に支障を来す場合が多いため, `bsave()` などでファイルにセーブして, `delete_history()` によりヒストリを消去しておくのが有効である.

```
[0] (x+y+z)^100$
[1] @0;
...
[2] delete_history(0);
[3] @0;
0
```

6.14.16 get_rootdir

`get_rootdir()`
:: Asir のルートディレクトリ名を取り出す

return　　文字列

- UNIX 版の場合, 環境変数 ASIR_LIBDIR が定義されている場合にはその値, されていない場合には '/usr/local/lib/asir' を返す.
- Windows 版の場合, 'asirgui.exe' のあるディレクトリ ('bin' という名前のはずである) の親ディレクトリが返される.
- この関数が返すディレクトリ名を基準とした相対パス名を指定することにより, インストールされた場所によらないファイル読み込みプログラムを書くことができる.

6.14.17 getopt

getopt([*key*])
:: オプションの値を返す.

return　　オブジェクト

- ユーザ定義関数は, 固定個数引数でしか宣言できない. ユーザ定義関数で可変個引数を実現する方法の一つとして, オプションによる引数の指定がある (4.2.12 節「オプション指定」p.24 参照). 指定されたオプションを関数内で受け取るためにこの関数を用いる.
- 無引数で呼び出された場合, getopt() は [[key1,value1],[key2,value2],...] なるリストを返す. ここで, key は関数呼び出し時に指定されたオプション, value はその値である.
- 関数呼び出しの際に *key* がオプションとして指定されている場合には, その値を返す. もし指定がない場合には, VOID 型オブジェクト (型識別子 -1) を返す. getopt() が返した値の型を type() で調べることで, そのオプションが指定されたかどうか調べることができる.
- 関数呼び出しにおけるオプションの指定は, 正規の引数ならびの後ろに,

  ```
  xxx(A,B,C,D|x=X,y=Y,z=Z)
  ```

  という風に, '|' に続く, *key*=*value* の ',' で区切られた並びを置くことで行う.

参照　　4.2.12 節「オプション指定」p.24, 6.8.1 節「type」p.68.

6.14.18 getenv

getenv(*name*)
:: 環境変数の値を返す.

return
name　　文字列

- 環境変数 *name* の値を返す.

```
[0] getenv("HOME");
/home/pcrf/noro
```

7 分散計算

7.1 OpenXM

Asir は, 分散計算における通信プロトコルとして, **OpenXM** (Open message eXchange for Mathematics) プロトコルを採用している. **OpenXM** プロジェクトについては, `http://www.math.sci.kobe-u.ac.jp/OpenXM/` を参照してほしい.

OpenXM プロトコルは, 主として数学オブジェクトをプロセス間でやりとりするための規約である. **OpenXM** においては

1. client が server に対して計算実行依頼のメッセージを送る.
2. server が計算を実行する.
3. client が server に結果送付依頼のメッセージを送る.
4. server は結果を返し, client は結果を受け取る

という形で分散計算が行われる. server はスタックマシンである. すなわち, client から送られたデータオブジェクトは, 指定がない限り server のスタックに積まれ, コマンドが送られた時に, 必要なだけスタックからデータを取り出して, 関数呼び出しの引数とする.

OpenXM において特徴的なことは, 計算結果は単に server のスタックに積まれるだけで, client からの依頼がない限り, 通信路にデータは流れないという点である.

プロトコルには, オブジェクトの共通フォーマットを規定する **CMO** (Common Mathematical Object format), プロセスに対する動作を指定する **SM** (Stack Machine command) が含まれる. これらは, データを送る際に, データの種類を指定するための **OX** expression としてラッピングされる.

OpenXM による分散計算を行う場合には, まず, server を立ち上げて, 通信を成立させる必要がある. このために, `ox_launch()`, `ox_launch_nox()`, `ox_launch_generic()` などの関数が用意されている. さらに, 通信の成立した server に対して以下のような操作が関数として用意されている.

`ox_push_cmo()`
: データを server のスタックに積む

`ox_pop_cmo()`
: データを server のスタックから取り出す.

`ox_cmo_rpc()`
: server の関数を呼び出し, 結果をスタックに積む.

`ox_execute_string()`
: server 固有のユーザ言語 (**Asir** なら Asir 言語) で書かれた文字列を server が実行し, 結果をスタックに積む.

`ox_push_cmd()`
: **SM** コマンドの送信.

`ox_get()`
: 既に通信路にあるデータの取り出し.

7.2 Mathcap

server, client ともに, **OpenXM** で規定されている全ての **CMO** フォーマット, **SM** コマンドを実装しているとは限らない. 相手の知らないデータ, コマンドを送った場合, 現状では結果は予想できない. このため, **OpenXM** では, あらかじめ互いのサポートする **CMO**, **SM** のリストを交換しあって, 相手の知らないデータを送らないようにする仕組みを提唱している. このためのデータが Mathcap である. Mathcap は **CMO** としてはリストであり, その要素は 32 bit 整数または文字列である. 現在の規定では, Mathcap は長さが 3 のリストで,

[[version 番号, server 名],**SM**taglist,[[**OX**tag,**CMO**taglist], [**OX**tag,**CMO**taglist],...]]

という形をしている. [**OX**tag,**CMO**taglist] は, **OX**tag で示されるカテゴリのデータに対して, どのような **CMO** が使用可能かを示すものである. この指定を複数許すことにより, 例えば 'ox_asir' のように, **CMO** データ以外に, **Asir** 固有のデータ形式により, **CMO** より多くの種類のデータ送受信を行えることを示せる.

データ送信の際に, 相手プロセスの Mathcap が既に登録されている場合, Mathcap によるチェックを行うか否かは, ctrl コマンドの "ox_check" スイッチにより決まる. このスイッチの初期値は 1 で, チェックを行うことを意味する. ctrl("ox_check",0) によりチェックを行わないようにできる.

7.3 スタックマシンコマンド

スタックマシンコマンドは, スタックマシンである server に何らかの操作を行わせるために用意されている. いくつかのコマンドは, よく用いられる形で, 他のコマンド, データとともに, **Asir** の組み込み関数により送られるが, ユーザが明示的にあるコマンドを送る必要がしばしば生ずる. スタックマシンコマンドは 32 bit 以下の整数であり, ox_push_cmd() コマンドで送信できる. 以下で, 代表的なスタックマシンコマンドについて解説する. **SM_xxx=yyy** で, **SM_xxx** が mnemonic, **yyy** が値である.

以下で, スタックからデータを取り出すとは, スタックの一番上からデータを取り除くことを言う.

SM_popSerializedLocalObject=258
: server が 'ox_asir' の場合に, 必ずしも **CMO** で定義されていないオブジェクトをスタックから取り出し, 通信路に流す.

SM_popCMO=262
: **CMO** オブジェクトをスタックから取り出し, 通信路に流す.

SM_popString=263
: スタックからデータを取り出し, 可読形式の文字列に変換して通信路に流す.

SM_mathcap=264
: server の mathcap をスタックに積む.

SM_pops=265
: スタックから取り出したデータを個数として, その個数分スタックからデータを取り除く.

SM_setName=266
: スタックからデータを変数名として取り出し, 次に取り出したデータをその変数に割り当てる. この割り当ては, server 固有の処理として行われる.

SM_evalName=267
スタックから取り出したデータを変数名として, その値をスタックに載せる.

SM_executeStringByLocalParser=268
スタックから取り出したデータを, server 固有の parser, evaluator で処理し, 結果をスタックに載せる.

SM_executeFunction=269
スタックから, 関数名, 引数の個数, 個数分の引数を取り出し, 関数を呼び出し結果をスタックに載せる.

SM_beginBlock=270
データブロックのはじまり.

SM_endBlock=271
データブロックの終り.

SM_shutdown=272
server との交信を切断し, server を終了させる.

SM_setMathcap=273
スタックのデータを client の mathcap として, server に登録を要求する.

SM_getsp=275
現在スタックに積まれているデータの数をスタックに載せる.

SM_dupErrors=276
現在スタックに積まれているオブジェクトの内, エラーオブジェクトのみをリストにして, スタックに載せる.

SM_nop=300
なにもしない.

7.4 デバッグ

分散計算においては, 一般にデバッグが困難となる. 'ox_asir' においては, デバッグのためのいくつかの機能を提供している.

7.4.1 エラーオブジェクト

OpenXM server が実行中にエラーを起こした場合, 結果のかわりに **CMO** エラーオブジェクトをスタックに積む. エラーオブジェクトは, 対応する **SM** コマンドのシリアル番号と, エラーメッセージからなり, それによってどの **SM** コマンドがどのようなエラーを起こしたがある程度判明する.

```
[340] ox_launch();
0
[341] ox_rpc(0,"fctr",1.2*x);
0
[342] ox_pop_cmo(0);
error([8,fctrp : invalid argument])
```

7.4.2 リセット

`ox_reset()` は現在実行中の server をリセットして, コマンド受け付け状態に戻す. この機能は, 通常の **Asir** セッションにおけるキーボード割り込みとほぼ同様に, **OpenXM** server をリセットできる. また, 何らかの原因で, 通信路のデータが載ったままの状態で `ox_rpc()` などを実行すると, `ox_pop_cmo()` など, スタックからの取り出しと, 実際に読まれるデータの対応が不正になる. そのような場合にも有効である.

7.4.3 デバッグ用ポップアップウィンドウ

server には, client におけるキーボードに相当する入力機能がないため, server 側で動作しているユーザ言語プログラムのデバッグが困難になる. このため, server 側でのユーザ言語プログラム実行中のエラーおよび, client からの `ox_rpc(`*id*`,"debug")` 実行により, server にデバッグコマンドを入力するための小さなウィンドウがポップアップする. このウィンドウからの入力に対する出力は, log 用の '`xterm`' に表示される. このウィンドウを閉じるには, `quit` を入力すればよい.

7.5 分散計算に関する関数

7.5.1 ox_launch, ox_launch_nox, ox_shutdown

`ox_launch(`[*host*`[,`*dir*`],`*command*`])`
`ox_launch_nox(`[*host*`[,`*dir*`],`*command*`])`
:: 遠隔プロセスの起動および通信を開始する.

`ox_shutdown(`*id*`)`
:: 遠隔プロセスを終了させ, 通信を終了する.

return　整数

host　文字列または 0

dir command
文字列

id　整数

- `ox_launch()` は, ホスト *host* 上でコマンド *command* を起動し, このプロセスと通信を開始する. 引数が 3 つの場合, *host* 上で, *dir* にある '`ox_launch`' というサーバ起動用プログラムを立ち上げる. '`ox_launch`' は *command* を起動する. *host* が 0 の時, **Asir** が動作しているマシン上でコマンドを起動する. 無引数の場合, *host* は 0, *dir* は `get_rootdir()` で返されるディレクトリ, *command* は同じディレクトリの '`ox_asir`' を意味する.
- *host* が 0, すなわちサーバを local に起動する場合には, *dir* を省略できる. この場合, *dir* は `get_rootdir()` で返されるディレクトリとなる.
- *command* が '`/`' で始まる文字列の場合, 絶対パスと解釈される. それ以外の場合, *dir* からの相対パスと解釈される.
- UNIX 版においては, `ox_launch()` は, *command* の標準出力, 標準エラー出力を表示するための '`xterm`' を起動する. `ox_launch_nox()` は, `X` なしの環境の場合, あるいは

‘xterm’ を起動せずにサーバを立ち上げる場合に用いる．この場合, *command* の出力は ‘/dev/null’ に接続される．ox_launch() の場合でも, 環境変数 DISPLAY が設定されていない場合には, ox_launch_nox() と同じ動作をする．

- 返される整数は通信のための識別子となる．
- **Asir** と通信するプロセスは同一のマシン上で動作している必要はない．また, 通信におけるバイトオーダは server, client 間での最初の negotiation で決まるため, 相手先のマシンとバイトオーダが異なっていても構わない．
- *host* にマシン名を指定する場合, 以下の準備が必要である．ここで, **Asir** の動いているホストを A, 通信相手のプロセスが起動されるホストを B とする．
  1. ホスト B の ‘~/.rhosts’ に, ホスト A のホスト名を登録する．
  2. ‘ox_plot’ など, X とのコネクションも用いられる場合, Xserver に対し, 必要なホストを authorize させる．xhost で必要なホスト名を追加すればよい．
  3. *command* によっては, スタックを大量に使用するものもあるため, ‘.cshrc’ でスタックサイズを大きめ (16MB 程度) に指定しておくのが安全である．スタックサイズは limit stacksize 16m などと指定する．
- *command* が, X 上にウインドウを開ける場合, *display* が指定されればその文字列を, 省略時には環境変数 DISPLAY の値を用いる．
- 環境変数 ASIR_RSH がセットされている場合, サーバの立ち上げプログラムとして ‘rsh’ の代わりにこの変数の値が用いられる．例えば,

  ```
  % setenv ASIR_RSH "ssh -f -X -A "
  ```

  により, サーバの立ち上げに ‘ssh’ が用いられ, X11 の通信が forwarding される．詳しくは ‘ssh’ のマニュアルを参照．
- ox_shutdown() は識別子 *id* に対応する遠隔プロセスを終了させる．
- **Asir** が正常した場合には全ての入出力ストリームは自動的に閉じられ, 起動されているプロセスは全て終了するが, 異常終了した場合, 遠隔プロセスが終了しない場合もある．**Asir** が異常終了した場合, 遠隔プロセスを起動したマシン上で ps などを起動して, もし **Asir** から起動したプロセスが残っている場合, kill する必要がある．
- log 表示用 ‘xterm’ は ‘-name ox_term’ オプションで起動される．よって, ‘ox_term’ なるリソース名に対して ‘xterm’ のリソース設定を行えば, log 用 ‘xterm’ の挙動のみを変えることができる．例えば,

  ```
  ox_xterm*iconic:on
  ox_xterm*scrollBar:on
  ox_xterm*saveLines:1000
  ```

  により, icon で起動, scrollbar つき, scrollbar で参照できる行数が最大 1000 行, という指定ができる．

```
[219] ox_launch();
0
[220] ox_rpc(0,"fctr",x^10-y^10);
0
[221] ox_pop_local(0);
[[1,1],[x^4+y*x^3+y^2*x^2+y^3*x+y^4,1],
[x^4-y*x^3+y^2*x^2-y^3*x+y^4,1],[x-y,1],[x+y,1]]
[222] ox_shutdown(0);
0
```

参照　7.5.5 節「ox_rpc ox_cmo_rpc ox_execute_string」p.98, 7.5.8 節「ox_pop_cmo ox_pop_local」p.101, 7.5.15 節「ifplot conplot plot polarplot plotover」p.105

7.5.2 ox_launch_generic

ox_launch_generic(*host*,*launch*,*server*,*use_unix*,*use_ssh*,*use_x*,*conn_to_serv*)
:: 遠隔プロセスの起動および通信を開始する.

return　整数

host　文字列または 0

launcher server
文字列

use_unix use_ssh use_x conn_to_serv
整数

- ox_launch_generic() は, ホスト *host* 上で, コントロールプロセス *launch* およびサーバプロセス *server* を起動する. その他の引数は, 使用する protocol の種類, X の使用/不使用, rsh/ssh によるプロセス起動, connect 方法の指定などを行うスイッチである.
- *host* が 0 の場合, **Asir** が動作しているマシン上に, *launch*, *server* を立ち上げる. この場合, *use_unix* の値にかかわらず, UNIX internal protocol が用いられる.
- *use_unix* が 1 の場合, UNIX internal protocol を用いる. 0 の場合, Internet protocol を用いる.
- *use_ssh* が 1 の場合, 'ssh' (Secure Shell) によりコントロール, サーバプロセスを立ち上げる. 'ssh-agent' などを利用していな場合, パスワードの入力が必要となる. 相手先で 'sshd' が動いていない場合, 自動的に 'rsh' が用いられるが, パスワードが必要となる場合には, その場で起動に失敗する.
- *use_x* が 1 の場合, X 上での動作を仮定し, 設定されている DISPLAY 変数を用いて, log 表示用 'xterm' のもとで *server* が起動される. DISPLAY 変数がセットされていない場合には, 自動的に X なしの設定となる. DISPLAY が不適切にセットされている場合には, コントロール, サーバがハングするので要注意である.
- *conn_to_serv* が 1 の場合, **Asir** (client) が生成したポートに対し, client が bind,listen し, 起動されたプロセスが connect する. *conn_to_serv* が 0 の場合, 起動されたプロセスが bind, listen し, client が connect する.

```
[342] LIB=get_rootdir();
/export/home/noro/ca/Kobe/build/OpenXM/lib/asir
[343] ox_launch_generic(0,LIB+"/ox_launch",LIB+"/ox_asir",0,0,0,0);
1
[344] ox_launch_generic(0,LIB+"/ox_launch",LIB+"/ox_asir",1,0,0,0);
2
[345] ox_launch_generic(0,LIB+"/ox_launch",LIB+"/ox_asir",1,1,0,0);
3
[346] ox_launch_generic(0,LIB+"/ox_launch",LIB+"/ox_asir",1,1,1,0);
4
[347] ox_launch_generic(0,LIB+"/ox_launch",LIB+"/ox_asir",1,1,1,1);
5
```

```
[348] ox_launch_generic(0,LIB+"/ox_launch",LIB+"/ox_asir",1,1,0,1);
6
```

参照　7.5.1 節「ox_launch ox_launch_nox ox_shutdown」p.94, 7.5.2 節「ox_launch_generic」p.96

7.5.3 generate_port, try_bind_listen, try_connect, try_accept, register_server

`generate_port([`*use_unix*`])`
:: port の生成

`try_bind_listen(`*port*`)`
:: port に対して bind, listen

`try_connect(`*host*`,`*port*`)`
:: port に対して connect

`try_accept(`*socket*`,`*port*`)`
:: connect 要求を accept

`register_server(`*control_socket*`,`*control_port*`,`*server_socket*`,`*server_port*`)`
:: connection の成立した control socket, server socket の登録

return　`generate_port()` のみ整数または文字列. その他は整数.

use_unix　0 または 1

host　文字列

port control_port server_port
整数または文字列

socket control_socket server_socket
整数

- これらの関数は, 遠隔プロセスと通信を成立させるためのプリミティブである.
- `generate_port()` は通信のための port を生成する. 無引数あるいは引数が 0 の場合, Internet domain の socket のための port 番号, それ以外の場合には, UNIX domain (host-internal protocol) のための, ファイル名を生成する. port 番号は random に生成されるが, その port が使用中でない保証はない.
- `try_bind_listen()` は, 与えられた port に対し, その protocol に対応した socket を生成し, bind, listen する. 成功した場合, socket 識別子を返す. 失敗した場合, -1 が返る.
- `try_connect()` は, ホスト *host* の port *port* に対し connect を試みる. 成功した場合, socket 識別子を返す. 失敗した場合 -1 が返る.
- `try_accept()` は, *socket* に対する connect 要求を accept し, 新たに生成された socket を返す. 失敗した場合 -1 が返る. いずれの場合にも, *socket* は自動的に close される. 引数 *port* は, *socket* の protocol を判別するために与える.
- `register_server()` は, control, server それぞれの socket を一組にして, server list に登録し, `ox_push_cmo()` などで用いるプロセス識別子を返す.
- 遠隔プロセスの起動は, `shell()` または手動で行う.

```
[340] CPort=generate_port();
39896
```

```
[341] SPort=generate_port();
37222
[342] CSocket=try_bind_listen(CPort);
3
[343] SSocket=try_bind_listen(SPort);
5

/*
ここで, ox_launch を起動 :
%  ox_launch "127.1" 0 39716 37043 ox_asir "shio:0"
*/

[344] CSocket=try_accept(CSocket,CPort);
6
[345] SSocket=try_accept(SSocket,SPort);
3
[346] register_server(CSocket,CPort,SSocket,SPort);
0
```

参照　7.5.1 節「ox_launch ox_launch_nox ox_shutdown」p.94, 7.5.2 節「ox_launch_generic」p.96, 6.14.12 節「shell」p.88, 7.5.7 節「ox_push_cmo ox_push_local」p.100

7.5.4 'ox_asir'

'ox_asir' は, **Asir** のほぼ全ての機能を **OpenXM** サーバとして提供する. 'ox_asir' は, ox_launch または ox_launch_nox で起動する. 後者は X 環境を用いない場合のために用意されている.

```
[5] ox_launch();
0
[5] ox_launch_nox("127.0.0.1","/usr/local/lib/asir",
"/usr/local/lib/asir/ox_asir");
0
[7] RemoteLibDir = "/usr/local/lib/asir/"$
[8] Machines = ["sumire","rokkaku","genkotsu","shinpuku"];
[sumire,rokkaku,genkotsu,shinpuku]
[9] Servers = map(ox_launch,Machines,RemoteLibDir,
RemoteLibDir+"ox_asir");
[0,1,2,3]
```

参照　7.5.1 節「ox_launch ox_launch_nox ox_shutdown」p.94

7.5.5 ox_rpc, ox_cmo_rpc, ox_execute_string

ox_rpc(*number*,"*func*",*arg0*,...)
ox_cmo_rpc(*number*,"*func*",*arg0*,...)
ox_execute_string(*number*,"*command*",...)
:: プロセスの函数呼び出し

return　0

number　数 (プロセス識別子)

func　函数名

command　文字列

arg0 ...　任意 (引数)

- 識別子 *number* のプロセスの函数を呼び出す.
- 函数の計算終了を待たず, 直ちに 0 を返す.
- `ox_rpc()` は, サーバが '`ox_asir`' の場合のみ用いることができる. それ以外の場合は, `ox_cmo_rpc()` を用いる.
- 函数が返す値は `ox_pop_local()`, `ox_pop_cmo()` により取り出す.
- サーバが '`ox_asir`' 以外のもの (例えば Kan サーバ '`ox_sm1`'など) の場合には, **Open_XM** プロトコルでサポートされているデータのみを送ることができる.
- `ox_execute_string` は, 送った文字列 *command* をサーバが自らのユーザ言語パーザで解析し, 評価した結果をサーバのスタックに置くように指示する.

```
[234] ox_cmo_rpc(0,"dp_ht",dp_ptod((x+y)^10,[x,y]));
0
[235] ox_pop_cmo(0);
(1)*<<10,0>>
[236] ox_execute_string(0,"12345 % 678;");
0
[237] ox_pop_cmo(0);
141
```

参照　7.5.8 節「`ox_pop_cmo ox_pop_local`」p.101

7.5.6 ox_reset,ox_intr,register_handler

`ox_reset(`*number*`)`
:: プロセスのリセット

`ox_intr(`*number*`)`
:: プロセスのに `SIGINT` 送付

`register_handler(`*func*`)`
:: プロセスのリセットのための関数登録

return　1

number　数 (プロセス識別子)

func　関数子または 0

- `ox_reset()` は, 識別子 *number* のプロセスをリセットし, コマンド受け付け状態にする.
- そのプロセスが既に書き出した, あるいは現在書き出し中のデータがある場合, それを全部読み出し, 出力バッファを空にした時点で戻る.
- 子プロセスが RUN 状態の場合でも, 割り込みにより強制的に計算を終了させる.
- 分散計算を行う函数の先頭で, 使用するプロセスに対して実行する. あるいは計算途中での強制中断に用いる.

- ox_intr() は, 識別子 *number* のプロセスをに対して SIGINT を送付する. SIGINT に対するプロセスの動作は規定されていないが, 'ox_asir' の場合, ただちに debug mode に入る. X 上で動作している場合, デバッグコマンド入力用のウィンドウがポップアップする.
- register_handler() は, *C-c* などによる割り込みの際に, *u* を指定することで, 無引数ユーザ定義関数 *func*() が呼び出されるように設定する. この関数に, ox_reset() を呼び出させることで, 割り込みの際に自動的に **OpenXM** server のリセットを行うことができる.
- *func* に 0 を指定することで, 設定を解除できる.

```
[10] ox_launch();
0
[11] ox_rpc(0,"fctr",x^100-y^100);
0
[12] ox_reset(0); /* xterm のウィンドウには                                    */
1                 /* usr1 : return to toplevel by SIGUSR1 が表示される. */
[340] Procs=[ox_launch(),ox_launch()];
[0,1]
[341] def reset() { extern Procs; map(ox_reset,Procs);}
[342] map(ox_rpc,Procs,"fctr",x^100-y^100);
[0,0]
[343] register_handler(reset);
1
[344] interrupt ?(q/t/c/d/u/w/?) u
Abort this computation? (y or n) y
Calling the registered exception handler...done.
return to toplevel
```

参照　7.5.5 節「ox_rpc ox_cmo_rpc ox_execute_string」p.98

7.5.7 ox_push_cmo, ox_push_local

ox_push_cmo(*number*,*obj*)
ox_push_local(*number*,*obj*)
:: *obj* を識別子 *number* のプロセスに送信

return　0

number　数 (プロセス識別子)

obj　オブジェクト

- 識別子 *number* のプロセスに *obj* を送信する.
- ox_push_cmo は, Asir 以外の **Open_XM** サーバに送信する際に用いる.
- ox_push_local は, 'ox_asir', 'ox_plot' にデータを送る場合に用いることができる.
- バッファがいっぱいにならない限り, ただちに復帰する.

参照　7.5.5 節「ox_rpc ox_cmo_rpc ox_execute_string」p.98, 7.5.8 節「ox_pop_cmo ox_pop_local」p.101

7.5.8 ox_pop_cmo, ox_pop_local

`ox_pop_local(`*number*`)`
:: プロセス識別子 *number* からデータを受信する.

return　受信データ

number　数 (プロセス識別子)

- プロセス識別子 *number* のプロセスからデータを受信する.
- `ox_pop_cmo` は, Asir 以外の **Open_XM** サーバから受信する際に用いる.
- `ox_pop_local` は, 'ox_asir', 'ox_plot' からデータを受け取る場合に用いることができる.
- サーバが計算中の場合ブロックする. これを避けるためには, `ox_push_cmd` で `SM_popCMO` (262) または `SM_popSerializedLocalObject` (258) を送っておき, `ox_select` でプロセスが ready になっていることを確かめてから `ox_get` すればよい.

```
[341] ox_cmo_rpc(0,"fctr",x^2-1);
0
[342] ox_pop_cmo(0);
[[1,1],[x-1,1],[x+1,1]]
[343] ox_cmo_rpc(0,"newvect",3);
0
[344] ox_pop_cmo(0);
error([41,cannot convert to CMO object])
[345] ox_pop_local(0);
[ 0 0 0 ]
```

参照　7.5.5 節「ox_rpc ox_cmo_rpc ox_execute_string」p.98, 7.5.9 節「ox_push_cmd ox_sync」p.101, 7.5.12 節「ox_select」p.103, 7.5.10 節「ox_get」p.102

7.5.9 ox_push_cmd, ox_sync

`ox_push_cmd(`*number*`,`*command*`)`
:: プロセス識別子 *number* のプロセスにコマンド *command* を送信する.

`ox_sync(`*number*`)`
:: プロセス識別子 *number* のプロセスに **OX_SYNC_BALL** を送信する.

return　0

number　数 (プロセス識別子)

command　数 (コマンド識別子)

- 識別子 *number* のプロセスにコマンドまたは **OX_SYNC_BALL** を送信する.
- **Open_XM** において送受信データは **OX_DATA**, **OX_COMMAND**, **OX_SYNC_BALL** の 3 種類に分かれる. 通常, コマンドは何らかの操作に付随して暗黙のうちに送信されるが, これをユーザが個別に送りたい場合に用いられる.
- **OX_SYNC_BALL** は `ox_reset` による計算中断, 復帰の際に送受信されるが, これを個別に送りたい場合に用いる. なお, 通常状態では **OX_SYNC_BALL** は無視される.

```
[3] ox_rpc(0,"fctr",x^100-y^100);
0
[4] ox_push_cmd(0,258);
0
[5] ox_select([0]);
[0]
[6] ox_get(0);
[[1,1],[x^2+y^2,1],[x^4-y*x^3+y^2*x^2-y^3*x+y^4,1],...]
```

参照　7.5.5 節「ox_rpc ox_cmo_rpc ox_execute_string」p.98, 7.5.6 節「ox_reset ox_intr register_handler」p.99

7.5.10 ox_get

ox_get(*number*)
:: プロセス識別子 *number* のプロセスからデータを受信する.

return　受信データ

number　数 (プロセス識別子)

- プロセス識別子 *number* のプロセスからデータを受信する. 既にストリーム上にデータがあることを仮定している.
- ox_push_cmd と組み合わせて用いる.
- ox_pop_cmo, ox_pop_local は, ox_push_cmd と ox_get の組み合わせで実現されている.

```
[11] ox_push_cmo(0,123);
0
[12] ox_push_cmd(0,262); /* 262=OX_popCMO */
0
[13] ox_get(0);
123
```

参照　7.5.8 節「ox_pop_cmo ox_pop_local」p.101, 7.5.9 節「ox_push_cmd ox_sync」p.101

7.5.11 ox_pops

ox_pops(*number*[,*nitem*])
:: プロセス識別子 *number* のプロセスのスタックからデータを取り除く.

return　0

number　数 (プロセス識別子)

nitem　自然数

- プロセス識別子 *number* のプロセスのスタックからデータを取り除く. *nitem* が指定されている場合は *nitem* 個, 指定のない場合は 1 個取り除く.

```
[69] for(I=1;I<=10;I++)ox_push_cmo(0,I);
[70] ox_pops(0,4);
0
[71] ox_pop_cmo(0);
6
```

参照　7.5.8 節「ox_pop_cmo ox_pop_local」p.101

7.5.12 ox_select

ox_select(*nlist*[,*timeout*])
:: 読み出し可能なプロセスの識別子を返す.

return　リスト

nlist　数 (子プロセス識別子) のリスト

timeout　数

- 識別子リスト *nlist* のプロセスのうち既に出力を返しているプロセスの識別子リストを返す.
- 全てのプロセスが RUN 状態のとき, いずれかのプロセスの終了を待つ. 但し, *timeout* が指定されている場合, *timeout* 秒だけ待つ.
- ox_push_cmd() で SM_popCMO あるいは SM_popSerializedLocalObject を送っておき, ox_select() で ready 状態のプロセスを調べて ox_get() することで, ox_pop_local(), ox_pop_cmo()で待ち状態に入るのを防ぐことができる.

```
ox_launch();
0
[220] ox_launch();
1
[221] ox_launch();
2
[222] ox_rpc(2,"fctr",x^500-y^500);
0
[223] ox_rpc(1,"fctr",x^100-y^100);
0
[224] ox_rpc(0,"fctr",x^10-y^10);
0
[225] P=[0,1,2];
[0,1,2]
[226] map(ox_push_cmd,P,258);
[0,0,0]
[227] ox_select(P);
[0]
[228] ox_get(0);
[[1,1],[x^4+y*x^3+y^2*x^2+y^3*x+y^4,1],
[x^4-y*x^3+y^2*x^2-y^3*x+y^4,1],[x-y,1],[x+y,1]]
```

参照　7.5.8 節「ox_pop_cmo ox_pop_local」p.101, 7.5.9 節「ox_push_cmd ox_sync」p.101, 7.5.10 節「ox_get」p.102

7.5.13 ox_flush

ox_flush(*id*)
:: 送信バッファの強制 flush

return　1

id　　子プロセス識別子

- 通常はバッチモードは off であり, データ, コマンド送信ごとに送信バッファは flush される.
- バッチモードは "ctrl" コマンドの "ox_batch" スイッチで on/off できる.
- 細かいデータを多数送る場合に, ctrl("ox_batch",1) でバッチモードを on にすると, バッファがいっぱいになった場合にのみ flush されるため, overhead が小さくなる場合がある. ただしこの場合には, 最後に ox_flush(*id*) を実行して, バッファを強制的に flush する必要がある.
- ox_pop_cmo, ox_pop_local のように, コマンド送信後ただちにデータ待ちに入る関数がハングしないよう, これらの関数の内部では強制 flush が実行されている.

```
[340] ox_launch_nox();
0
[341] cputime(1);
0
7e-05sec + gc : 4.8e-05sec(0.000119sec)
[342] for(I=0;I<10000;I++)ox_push_cmo(0,I);
0.232sec + gc : 0.006821sec(0.6878sec)
[343] ctrl("ox_batch",1);
1
4.5e-05sec(3.302e-05sec)
[344] for(I=0;I<10000;I++)ox_push_cmo(0,I); ox_flush(0);
0.08063sec + gc : 0.06388sec(0.4408sec)
[345] 1
9.6e-05sec(0.01317sec)
```

参照　　7.5.8 節「ox_pop_cmo ox_pop_local」p.101, 6.14.1 節「ctrl」p.82

7.5.14 ox_get_serverinfo

ox_get_serverinfo([*id*])
:: server の Mathcap, 動作中のプロセス識別子の取得

return　　リスト

id　　子プロセス識別子

- 引数 *id* があるとき, プロセス識別子 *id* のプロセスの Mathcap をリストとして返す.
- 引数なしのとき, 現在動作中のプロセス識別子およびその Mathcap からなるペアを, リストとして返す.

```
[343] ox_get_serverinfo(0);
[[199909080,Ox_system=ox_sm1.plain,Version=2.991118,HOSTTYPE=FreeBSD],
[262,263,264,265,266,268,269,272,273,275,276],
[[514],[2130706434,1,2,4,5,17,19,20,22,23,24,25,26,30,31,60,61,27,
33,40,16,34]]]
[344] ox_get_serverinfo();
[[0,[[199909080,Ox_system=ox_sm1.plain,Version=2.991118,
HOSTTYPE=FreeBSD],
[262,263,264,265,266,268,269,272,273,275,276],
[[514],[2130706434,1,2,4,5,17,19,20,22,23,24,25,26,30,31,60,61,27,33,
```

```
 40,16,34]]]],
 [1,[[199901160,ox_asir],
 [276,275,258,262,263,266,267,268,274,269,272,265,264,273,300,270,271],
 [[514,2144202544],
 [1,2,3,4,5,2130706433,2130706434,17,19,20,21,22,24,25,26,31,27,33,60],
 [0,1]]]]]
```

参照　7.2 節「Mathcap」p.92.

7.5.15 ifplot, conplot, plot, polarplot, plotover

ifplot(*func* [,*geometry*] [,*xrange*] [,*yrange*] [,*id*] [,*name*])
:: 2 変数関数の実数上での零点を表示する.

conplot(*func* [,*geometry*] [,*xrange*] [,*yrange*] [,*zrange*] [,*id*] [,*name*])
:: 2 変数関数の実数上での等高線を表示する.

plot(*func* [,*geometry*] [,*xrange*] [,*id*] [,*name*])
:: 1 変数関数のグラフを表示する.

polarplot(*func* [,*geometry*] [,*thetarange*] [,*id*] [,*name*])
:: 極形式で与えられた曲線を表示する.

plotover(*func*,*id*,*number*)
:: すでに存在しているウィンドウへ描画する.

return　整数

func　多項式

geometry xrange yrange zrange
リスト

id number　整数

name　文字列

- ifplot() は, 2 変数関数 *func* の実数上での零点のグラフの表示を行う. conplot() は, 同様の引数に対し, 等高線の表示を行う. plot() は 1 変数関数のグラフの表示を行う. polarplot() は 極形式 $r=f(theta)$ で表された曲線のグラフの表示を行う.
- これらは OpenXM サーバとして実現されている. UNIX 上では 'ox_plot' が, Windows 上では 'engine' がこれらの機能を提供しており, これらは **Asir** の標準ライブラリディレクトリにある. アクティブな 'ox_plot' の id が *id* として指定された場合, そのサーバが用いられる. id の指定がない場合には, 起動されているサーバのうち, 'ox_plot' があればそのサーバが用いられる. 'ox_plot' が起動されていない場合には, ox_launch_nox() が自動的に実行されて, 'ox_plot' が立ち上がり, それが用いられる.
- 引数の内, *func* は必須である. その他の引数はオプションである. オプションの形式およびそのデフォルト値 (カッコ内) は次の通り.

  geometry　ウィンドウのサイズをドット単位で [*x*,*y*] で指定する. ([300,300].)

  xrange yrange
  変数の範囲の指定で, [*v*,*vmin*,*vmax*] で指定する. (いずれの変数も [*v*,-2,2].) この指定がない場合, *func* に含まれる変数の内変数順序の上の変数が 'x', 下の変数が 'y' として扱われる. これを避けるためには *xrange*, *yrange* を指定する. また, *func* が 1 変数の場合, これらの指定は必須となる.

zrange　`conplot()` の場合のみ指定できる. 形式は [*v*,*vmin*,*vmax* [,*step*]] で, *step* が指定された場合には, 等高線の間隔が (*vmax*-*vmin*)/*step* となる. ([*z*,-2,2,16].)

id　遠隔プロセスの番号, すなわち `ox_launch()` が返した番号を指定する. (一番最近に作られ, かつアクティブなプロセスに対応する番号.)

name　ウィンドウの名前. (`Plot`.) 生成されたウィンドウのタイトルは *name*:*n*/*m* となる. これは, プロセス番号 *n* のプロセスの, *m* 番のウィンドウを意味する. この番号は, `plotover()` で用いられる.

- 一つのプロセス上で描画できるウィンドウの数は最大 128 個である.
- `plotover()` は, 指定したウィンドウ上に, 引数である 2 変数多項式の零点を上書きする.
- 描画終了後のウィンドウ上で, マウスの左ボタンを押しながらのドラッグで範囲を指定しボタンを離すと新たなウィンドウが生成され, 指定した範囲が拡大して表示される. ドラッグは左上から右下へと行う. ドラッグを始めた後キャンセルする場合は, マウスポインタを始点の上か左に持っていってボタンを離せばよい. 新しいウインドウの形は, 指定領域と相似で, 最大辺が, 元のウィンドウの最大辺と一致するように定められる. 以下で説明する `precise` が on の場合, 選択した領域が同一 window 上で書き直される.
- ウィンドウ内で右ボタンを押すと, その点の座標がウィンドウの下部に表示される.
- `conplot()` で生成したウィンドウにおいて, ウィンドウの右側のマーカを中ボタンでドラッグすると, 対応する等高線の色が変わり, 右上のウィンドウに対応するレベルが表示される.
- UNIX 版ではいくつかのボタンによりいくつかの設定変更, 操作ができる. UNIX 版では次のボタンがある.

  `quit`　window を破壊する. 計算を中断する場合, `ox_reset()` を用いる.

  `wide` (トグル)
  　現在の表示部分を縦横各 10 倍した領域を表示する. 現在表示されている範囲はこの表示において中央部に長方形で示される. この表示で範囲指定を行うと, その範囲が新しいウィンドウに描画される.

  `precise` (トグル)
  　選択領域を, 整数演算により, より正確に再描画する. これは, *func* が有理数係数の 2 変数多項式の場合にのみ有効である. このモードでは Sturm 列と二分法により, 区間内の零点の個数を正確に求めていくもので, デフォルトの計算法よりも正確な描画が期待できる. ただし, 描画時間は余計にかかる場合が多い. この説明から明らかなように, この機能は有理数係数の多項式の描画に対してのみ有効である. ((x^2+y^2-1)^2 の描画で試してみよ.)

  `formula`　対応する式を表示する.

  `noaxis` (トグル)
  　座標軸を消す.
- '`ox_plot`' が起動されるマシンによっては, スタックを大量に使用するものもあるため, '`.cshrc`' でスタックサイズを大きめ (16MB 程度) に指定しておくのが安全である. スタックサイズは `limit stacksize 16m` などと指定する.
- `X` では、ウインドウの各部分について resource により色付けや、ボタンの形を変えることができる。resource の指定の仕方は以下の通り。(デフォルトを示しておく)

plot*form*shapeStyle は、rectangle, oval, ellipse, roundedRectangle が、指定できる。

```
plot*background:white
plot*form*shapeStyle:rectangle
plot*form*background:white
plot*form*quit*background:white
plot*form*wide*background:white
plot*form*precise*background:white
plot*form*formula*background:white
plot*form*noaxis*background:white
plot*form*xcoord*background:white
plot*form*ycoord*background:white
plot*form*level*background:white
plot*form*xdone*background:white
plot*form*ydone*background:white
```

参照　7.5.1 節「ox_launch ox_launch_nox ox_shutdown」p.94, 7.5.6 節「ox_reset ox_intr register_handler」p.99

7.5.16 open_canvas, clear_canvas, draw_obj, draw_string

open_canvas(*id*[,*geometry*])
:: 描画用ウィンドウ (キャンバス) を生成する.

clear_canvas(*id*,*index*)
:: キャンバスをクリアする.

draw_obj(*id*,*index*,*pointorsegment* [,*color*])
:: キャンバス上に点または線分を描画する.

draw_string(*id*,*index*,[*x*,*y*],*string* [,*color*])
:: キャンバス上に文字列を描画する.

return　0

id index color x y
整数

pointorsegment
リスト

string　文字列

- これらは OpenXM サーバ 'ox_plot' (Windows 上では 'engine') により提供される.
- open_canvas は, 描画用のウィンドウ (キャンバス) を生成する. *geometry* によりウィンドウのサイズを pixel 単位で [x,y] で指定する. default size は [300,300]. キャンバスの識別子として, 整数値を OpenXM サーバのスタックに push する. この識別子は draw_obj の呼び出しに必要であり, ox_pop_cmo により取り出して保持する必要がある.
- clear_canvas は, サーバ id *id*, キャンバス id *index* で指定されるキャンバスをクリアする.
- draw_obj は, サーバ id *id*, キャンバス id *index* で指定されるキャンバスに点または線分を描画する. *pointorsegment* が [x,y] の場合点の座標, [x,y,u,v] の場合 [x,y], [u,v] を

結ぶ線分を表すと見なされる．キャンバスの座標は, 左上隅を原点として横方向に第一座標, 縦方向に第二座標をとる．値は pixel 単位で指定する．*color* の指定がある場合, *color/65536* mod 256, *color/256* mod 256, *color* mod 256 をそれぞれ Red, Green, Blue の値 (最大 255) とみなす.

- `draw_string` は, サーバ id *id*, キャンバス id *index* で指定されるキャンバスに文字列を描画する．位置は [*x*,*y*] により指定する.

```
[182] Id=ox_launch_nox(0,"ox_plot");
0
[183] open_canvas(Id);
0
[184] Ind=ox_pop_cmo(Id);
0
[185] draw_obj(Id,Ind,[100,100]);
0
[186] draw_obj(Id,Ind,[200,200],0xffff);
0
[187] draw_obj(Id,Ind,[10,10,50,50],0xff00ff);
0
[187] draw_string(Id,Ind,[100,50],"hello",0xffff00);
0
[189] clear_canvas(Id,Ind);
0
```

参照　7.5.1 節「`ox_launch ox_launch_nox ox_shutdown`」p.94, 7.5.6 節「`ox_reset ox_intr register_handler`」p.99, 7.5.8 節「`ox_pop_cmo ox_pop_local`」p.101.

8 グレブナ基底の計算

8.1 分散表現多項式

分散表現多項式とは, 多項式の内部形式の一つである. 通常の多項式 (type が 2) は, 再帰表現と呼ばれる形式で表現されている. すなわち, 特定の変数を主変数とする 1 変数多項式で, その他の変数は, その 1 変数多項式の係数に, 主変数を含まない多項式として現れる. この係数が, また, ある変数を主変数とする多項式となっていることから再帰表現と呼ばれる.

$(x+y+z)^2 = 1\cdot x^2 + (2\cdot y + (2\cdot z))\cdot x + ((2\cdot z)\cdot y + (1\cdot z^2))$

これに対し, 多項式を, 変数の冪積と係数の積の和として表現したものを分散表現と呼ぶ.

$(x+y+z)^2 = 1\cdot x^2 + 2\cdot xy + 2\cdot xz + 1\cdot y^2 + 2\cdot yz + 1\cdot z^2$

グレブナ基底計算においては, 単項式に注目して操作を行うため多項式が分散表現されている方がより効率のよい演算が可能になる. このため, 分散表現多項式が, 識別子 9 の型として **Asir** のトップレベルから利用可能となっている. ここで, 後の説明のために, いくつかの言葉を定義しておく.

項 (term) 変数の冪積. すなわち, 係数 1 の単項式のこと. **Asir** においては,

```
<<0,1,2,3,4>>
```

という形で表示され, また, この形で入力可能である. この例は, 5 変数の項を示す. 各変数を a, b, c, d, e とするとこの項は b*c^2*d^3*e^4 を表す.

項順序 (term order)

分散表現多項式における項は, 次の性質を満たす全順序により整列される.

1. 任意の項 t に対し t > 1
2. t, s, u を項とする時, t > s ならば tu > su

この性質を満たす全順序を項順序と呼ぶ. この順序は変数順序 (変数のリスト) と項順序型 (数, リストまたは行列) により指定される.

単項式 (monomial)

項と係数の積.

```
2*<<0,1,2,3,4>>
```

という形で表示され, また, この形で入力可能である.

頭単項式 (head monomial)

頭項 (head term)

頭係数 (head coefficient)

分散表現多項式における各単項式は, 項順序により整列される. この時順序最大の単項式を頭単項式, それに現れる項, 係数をそれぞれ頭項, 頭係数と呼ぶ.

8.2 ファイルの読み込み

グレブナ基底を計算するための基本的な函数は dp_gr_main() および dp_gr_mod_main(), dp_gr_f_main() なる 3 つの組み込み函数であるが, 通常は, パラメタ設定などを行ったのちこれらを呼び出すユーザ函数を用いるのが便利である. これらのユーザ函数は, ファイル 'gr' を load() により読み込むことにより使用可能となる. 'gr' は, **Asir** の標準ライブラリディレクトリに置かれている.

```
[0] load("gr")$
```

8.3 基本的な函数

'gr' では数多くの函数が定義されているが, 直接グレブナ基底を計算するためのトップレベルは次の 3 つである. 以下で, *plist* は多項式のリスト, *vlist* は変数 (不定元) のリスト, *order* は変数順序型, p は 2^27 未満の素数である.

gr(*plist*,*vlist*,*order*)
: Gebauer-Moeller による useless pair elimination criteria, sugar strategy および Traverso による trace-lifting を用いた Buchberger アルゴリズムによる有理数係数グレブナ基底計算函数. 一般にはこの函数を用いる.

hgr(*plist*,*vlist*,*order*)
: 入力多項式を斉次化した後 gr() のグレブナ基底候補生成部により候補生成し, 非斉次化, interreduce したものを gr() のグレブナ基底チェック部でチェックする. 0 次元システム (解の個数が有限個の方程式系) の場合, sugar strategy が係数膨張を引き起こす場合がある. このような場合, strategy を斉次化による strategy に置き換えることにより係数膨張を抑制することができる場合が多い.

gr_mod(*plist*,*vlist*,*order*,*p*)
: Gebauer-Moeller による useless pair elimination criteria, sugar strategy および Buchberger アルゴリズムによる GF(p) 係数グレブナ基底計算函数.

8.4 計算および表示の制御

グレブナ基底の計算において, さまざまなパラメタ設定を行うことにより計算, 表示を制御することができる. これらは, 組み込み函数 dp_gr_flags() により設定参照することができる. 無引数で dp_gr_flags() を実行すると, 現在設定されているパラメタが, 名前と値のリストで返される.

```
[100] dp_gr_flags();
[Demand,0,NoSugar,0,NoCriB,0,NoGC,0,NoMC,0,NoRA,0,NoGCD,0,Top,0,
ShowMag,1,Print,1,Stat,0,Reverse,0,InterReduce,0,Multiple,0]
[101]
```

以下で, 各パラメタの意味を説明する. on の場合とは, パラメタが 0 でない場合をいう. これらのパラメタの初期値は全て 0 (off) である.

NoSugar
: on の場合, sugar strategy の代わりに Buchberger の normal strategy が用いられる.

NoCriB
: on の場合, 不必要対検出規準のうち, 規準 B を適用しない.

NoGC
: on の場合, 結果がグレブナ基底になっているかどうかのチェックを行わない.

NoMC
: on の場合, 結果が入力イデアルと同等のイデアルであるかどうかのチェックを行わない.

NoRA
: on の場合, 結果を reduced グレブナ基底にするための interreduce を行わない.

NoGCD
: on の場合, 有理式係数のグレブナ基底計算において, 生成された多項式の, 係数の content をとらない.

Top
: on の場合, normal form 計算において頭項消去のみを行う.

Reverse
: on の場合, normal form 計算の際の reducer を, 新しく生成されたものを優先して選ぶ.

`Print`　on の場合, グレブナ基底計算の途中におけるさまざまな情報を表示する.

`PrintShort`
on で、Print が off の場合, グレブナ基底計算の途中の情報を短縮形で表示する.

`Stat`　on で `Print` が off ならば, `Print` が on のとき表示されるデータの内, 集計データのみが表示される.

`ShowMag`　on で `Print` が on ならば, 生成が生成される毎に, その多項式の係数のビット長の和を表示し, 最後に, それらの和の最大値を表示する.

`Content`
`Multiple`　0 でない有理数の時, 有理数上の正規形計算において, 係数のビット長の和が `Content` 倍になるごとに係数全体の GCD が計算され, その GCD で割った多項式を簡約する. `Content` が 1 ならば, 簡約するごとに GCD 計算が行われ一般には効率が悪くなるが, `Content` を 2 程度とすると, 巨大な整数が係数に現れる場合, 効率が良くなる場合がある. backward compatibility のため、`Multiple` で整数値を指定できる.

`Demand`
正当なディレクトリ名 (文字列) を値に持つとき, 生成された多項式はメモリ中におかれず, そのディレクトリ中にバイナリデータとして置かれ, その多項式を用いる normal form 計算の際, 自動的にメモリ中にロードされる. 各多項式は, 内部でのインデックスをファイル名に持つファイルに格納される. ここで指定されたディレクトリに書かれたファイルは自動的には消去されないため, ユーザが責任を持って消去する必要がある.

`Print` が 0 でない場合次のようなデータが表示される.

```
[93] gr(cyclic(4),[c0,c1,c2,c3],0)$
mod= 99999989, eval = []
(0)(0)<<0,2,0,0>>(2,3),nb=2,nab=5,rp=2,sugar=2,mag=4
(0)(0)<<0,1,2,0>>(1,2),nb=3,nab=6,rp=2,sugar=3,mag=4
(0)(0)<<0,1,1,2>>(0,1),nb=4,nab=7,rp=3,sugar=4,mag=6
.
(0)(0)<<0,0,3,2>>(5,6),nb=5,nab=8,rp=2,sugar=5,mag=4
(0)(0)<<0,1,0,4>>(4,6),nb=6,nab=9,rp=3,sugar=5,mag=4
(0)(0)<<0,0,2,4>>(6,8),nb=7,nab=10,rp=4,sugar=6,mag=6
....gb done
reduceall
.......
membercheck
(0,0)(0,0)(0,0)(0,0)
gbcheck total 8 pairs
........
UP=(0,0)SP=(0,0)SPM=(0,0)NF=(0,0)NFM=(0.010002,0)ZNFM=(0.010002,0)
PZ=(0,0)NP=(0,0)MP=(0,0)RA=(0,0)MC=(0,0)GC=(0,0)T=40,B=0 M=8 F=6
D=12 ZR=5 NZR=6 Max_mag=6
[94]
```

最初に表示される `mod`, `eval` は, trace-lifting で用いられる法である. `mod` は素数, `eval` は有理式係数の場合に用いられる数のリストである.
計算途中で多項式が生成される毎に次の形のデータが表示される.

```
(TNF)(TCONT)HT(INDEX),nb=NB,nab=NAB,rp=RP,sugar=S,mag=M
```

それらの意味は次の通り.

TNF
: normal form 計算時間 (秒)

TCONT
: content 計算時間 (秒)

HT
: 生成された多項式の頭項

INDEX
: S-多項式を構成する多項式のインデックスのペア

NB
: 現在の, 冗長性を除いた基底の数

NAB
: 現在までに生成された基底の数

RP
: 残りのペアの数

S
: 生成された多項式の sugar の値

M
: 生成された多項式の係数のビット長の和 (ShowMag が on の時に表示される.)

最後に, 集計データが表示される. 意味は次の通り. (時間の表示において, 数字が 2 つあるものは, 計算時間と GC 時間のペアである.)

UP
: ペアのリストの操作にかかった時間

SP
: 有理数上の S-多項式計算時間

SPM
: 有限体上の S-多項式計算時間

NF
: 有理数上の normal form 計算時間

NFM
: 有限体上の normal form 計算時間

ZNFM
: NFM の内, 0 への reduction にかかった時間

PZ
: content 計算時間

NP
: 有理数係数多項式の係数に対する剰余演算の計算時間

MP
: S-多項式を生成するペアの選択にかかった時間

RA
: interreduce 計算時間

MC
: trace-lifting における, 入力多項式のメンバシップ計算時間

GC
: 結果のグレブナ基底候補のグレブナ基底チェック時間

T
: 生成されたペアの数

B, M, F, D
: 各 criterion により除かれたペアの数

ZR
: 0 に reduce されたペアの数

NZR
: 0 でない多項式に reduce されたペアの数

Max_mag
: 生成された多項式の, 係数のビット長の和の最大値

8.5 項順序の設定

項は内部では, 各変数に関する指数を成分とする整数ベクトルとして表現される. 多項式を分散表現多項式に変換する際, 各変数がどの成分に対応するかを指定するのが, 変数リストである. さらに, それら整数ベクトルの全順序を指定するのが項順序の型である. 項順序型は, 数, 数のリストあるいは行列で表現される.

基本的な項順序型として次の 3 つがある.

0 (DegRevLex; **全次数逆辞書式順序**)
: 一般に, この順序によるグレブナ基底計算が最も高速である. ただし, 方程式を解くという目的に用いることは, 一般にはできない. この順序によるグレブナ基底は, 解の個数の計算, イデアルのメンバシップや, 他の変数順序への基底変換のためのソースとして用いられる.

1 (DegLex; **全次数辞書式順序**)
: この順序も, 辞書式順序に比べて高速にグレブナ基底を求めることができるが, DegRevLex と同様直接その結果を用いることは困難である. しかし, 辞書式順序のグレブナ基底を求める際に, 斉次化後にこの順序でグレブナ基底を求めている.

2 (Lex; **辞書式順序**)
: この順序によるグレブナ基底は, 方程式を解く場合に最適の形の基底を与えるが計算時間がかかり過ぎるのが難点である. 特に, 解が有限個の場合, 結果の係数が

極めて長大な多倍長数になる場合が多い. この場合, gr(), hgr() による計算が極めて有効になる場合が多い.

これらを組み合わせてリストで指定することにより, 様々な消去順序が指定できる. これは,

```
[[O1,L1],[O2,L2],...]
```

で指定される. Oi は 0, 1, 2 のいずれかで, Li は変数の個数を表す. この指定は, 変数を先頭から L1, L2 , ... 個ずつの組に分け, それぞれの変数に関し, 順に O1, O2, ... の項順序型で大小が決定するまで比較することを意味する. この型の順序は一般に消去順序と呼ばれる.

さらに, 行列により項順序を指定することができる. 一般に, n 行 m 列の実数行列 M が次の性質を持つとする.

1. 長さ m の整数ベクトル v に対し Mv=0 と v=0 は同値.
2. 非負成分を持つ長さ m の 0 でない整数ベクトル v に対し, Mv の 0 でない最初の成分は非負.

この時, 2 つのベクトル t, s に対し, t>s を, M(t-s) の 0 でない最初の成分が非負, で定義することにより項順序が定義できる.

項順序型は, gr() などの引数として指定される他, 組み込み函数 dp_ord() で指定され, さまざまな函数の実行の際に参照される.

これらの順序の具体的な定義およびグレブナ基底に関する更に詳しい解説は [Becker,Weispfenning] などを参照のこと.

項順序型の設定の他に, 変数の順序自体も計算時間に大きな影響を与える.

```
[90] B=[x^10-t,x^8-z,x^31-x^6-x-y]$
[91] gr(B,[x,y,z,t],2);
[x^2-2*y^7+(-41*t^2-13*t-1)*y^2+(2*t^17-12*t^14+42*t^12+30*t^11-168*t^9
-40*t^8+70*t^7+252*t^6+30*t^5-140*t^4-168*t^3+2*t^2-12*t+16)*z^2*y
+(-12*t^16+72*t^13-28*t^11-180*t^10+112*t^8+240*t^7+28*t^6-127*t^5
-167*t^4-55*t^3+30*t^2+58*t-15)*z^4,
(y+t^2*z^2)*x+y^7+(20*t^2+6*t+1)*y^2+(-t^17+6*t^14-21*t^12-15*t^11
+84*t^9+20*t^8-35*t^7-126*t^6-15*t^5+70*t^4+84*t^3-t^2+5*t-9)*z^2*y
+(6*t^16-36*t^13+14*t^11+90*t^10-56*t^8-120*t^7-14*t^6+64*t^5+84*t^4
+27*t^3-16*t^2-30*t+7)*z^4,
(t^3-1)*x-y^6+(-6*t^13+24*t^10-20*t^8-36*t^7+40*t^5+24*t^4-6*t^3-20*t^2
-6*t-1)*y+(t^17-6*t^14+9*t^12+15*t^11-36*t^9-20*t^8-5*t^7+54*t^6+15*t^5
+10*t^4-36*t^3-11*t^2-5*t+9)*z^2,
-y^8-8*t*y^3+16*z^2*y^2+(-8*t^16+48*t^13-56*t^11-120*t^10+224*t^8+160*t^7
-56*t^6-336*t^5-112*t^4+112*t^3+224*t^2+24*t-56)*z^4*y+(t^24-8*t^21
+20*t^19+28*t^18-120*t^16-56*t^15+14*t^14+300*t^13+70*t^12-56*t^11
-400*t^10-84*t^9+84*t^8+268*t^7+84*t^6-56*t^5-63*t^4-36*t^3+46*t^2
-12*t+1)*z,2*t*y^5+z*y^2+(-2*t^11+8*t^8-20*t^6-12*t^5+40*t^3+8*t^2
-10*t-20)*z^3*y+8*t^14-32*t^11+48*t^8-t^7-32*t^5-6*t^4+9*t^2-t,
-z*y^3+(t^7-2*t^4+3*t^2+t)*y+(-2*t^6+4*t^3+2*t-2)*z^2,
2*t^2*y^3+z^2*y^2+(-2*t^5+4*t^2-6)*z^4*y
+(4*t^8-t^7-8*t^5+2*t^4-4*t^3+5*t^2-t)*z,
z^3*y^2+2*t^3*y+(-t^7+2*t^4+t^2-t)*z^2,
-t*z*y^2-2*z^3*y+t^8-2*t^5-t^3+t^2,
-t^3*y^2-2*t^2*z^2*y+(t^6-2*t^3-t+1)*z^4,z^5-t^4]
[93] gr(B,[t,z,y,x],2);
```

```
[x^10-t,x^8-z,x^31-x^6-x-y]
```

変数順序 [x,y,z,t] におけるグレブナ基底は, 基底の数も多く, それぞれの式も大きい. しかし, 順序 [t,z,y,x] にもとでは, B がすでにグレブナ基底となっている. 大雑把にいえば, 辞書式順序でグレブナ基底を求めることは, 左側の (順序の高い) 変数を, 右側の (順序の低い) 変数で書き表すことであり, この例の場合は, t, z, y が既に x で表されていることからこのような極端な結果となったわけである. 実際に現れる計算においては, このように選ぶべき変数順序が明らかであることは少なく, 試行錯誤が必要な場合もある.

8.6 Weight

前節で紹介した項順序は, 各変数に weight (重み) を設定することでより一般的なものとなる.

```
[0] dp_td(<<1,1,1>>);
3
[1] dp_set_weight([1,2,3])$
[2] dp_td(<<1,1,1>>);
6
```

単項式の全次数を計算する際, デフォルトでは各変数の指数の和を全次数とする. これは各変数の weight を 1 と考えていることに相当する. この例では, 第一, 第二, 第三変数の weight をそれぞれ 1,2,3 と指定している. このため, <<1,1,1>> の全次数 (以下ではこれを単項式の weight と呼ぶ) が 1*1+1*2+1*3=6 となる. weight を設定することで, 同じ項順序型のもとで異なる項順序が定義できる. 例えば, weight をうまく設定することで, 多項式を weighted homogeneous にすることができる場合がある.

各変数に対する weight をまとめたものを weight vector と呼ぶ. すべての成分が正であり, グレブナ基底計算において, 全次数の代わりに用いられるものを特に sugar weight と呼ぶことにする. sugar strategy において, 全次数の代わりに使われるからである. 一方で, 各成分が必ずしも正とは限らない weight vector は, sugar weight として設定することはできないが, 項順序の一般化には有用である. これらは, 行列による項順序の設定にすでに現れている. すなわち, 項順序を定義する行列の各行が, 一つの weight vector と見なされる. また, ブロック順序は, 各ブロックの変数に対応する成分のみ 1 で他は 0 の weight vector による比較を最初に行ってから, 各ブロック毎の tie breaking を行うことに相当する.

weight vector の設定は dp_set_weight() で行うことができるが, 項順序を指定する際の他のパラメタ (項順序型, 変数順序) とまとめて設定できることが望ましい. このため, 次のような形でも項順序が指定できる.

```
[64] B=[x+y+z-6,x*y+y*z+z*x-11,x*y*z-6]$
[65] dp_gr_main(B|v=[x,y,z],sugarweight=[3,2,1],order=0);
[z^3-6*z^2+11*z-6,x+y+z-6,-y^2+(-z+6)*y-z^2+6*z-11]
[66] dp_gr_main(B|v=[y,z,x],order=[[1,1,0],[0,1,0],[0,0,1]]);
[x^3-6*x^2+11*x-6,x+y+z-6,-x^2+(-y+6)*x-y^2+6*y-11]
[67] dp_gr_main(B|v=[y,z,x],order=[[x,1,y,2,z,3]]);
[x+y+z-6,x^3-6*x^2+11*x-6,-x^2+(-y+6)*x-y^2+6*y-11]
```

いずれの例においても, 項順序は option として指定されている. 最初の例では v により変数順序を, sugarweight により sugar weight vector を, orderにより項順序型を指定している. 二つ目の例における order の指定は matrix order と同様である. すなわち, 指定された weight vector を左から順に使って weight の比較を行う. 三つ目の例も同様であるが, ここでは weight vector の要素を変数毎に指定している. 指定がないものは 0 となる. 三つ目の

例では, order による指定では項順序が決定しない. この場合には, tie breaker として全次数逆辞書式順序が自動的に設定される. この指定方法は, dp_gr_main, dp_gr_mod_main などの組み込み関数でのみ可能であり, gr などのユーザ定義関数では未対応である.

8.7 有理式を係数とするグレブナ基底計算

gr() などのトップレベル函数は, いずれも, 入力多項式リストに現れる変数 (不定元) と, 変数リストに現れる変数を比較して, 変数リストにない変数が入力多項式に現れている場合には, 自動的に, その変数を, 係数体の元として扱う.

```
[64] gr([a*x+b*y-c,d*x+e*y-f],[x,y],2);
[(-e*a+d*b)*x-f*b+e*c,(-e*a+d*b)*y+f*a-d*c]
```

この例では, a, b, c, d が係数体の元として扱われる. すなわち, 有理函数体 $\mathbf{F} = \mathbf{Q}$(a,b,c,d) 上の 2 変数多項式環 $\mathbf{F}$[x,y] におけるグレブナ基底を求めることになる. 注意すべきことは, 係数が体として扱われていることである. すなわち, 係数の間に多項式としての共通因子があった場合には, 結果からその因子は除かれているため, 有理数体上の多項式環上の問題として考えた場合の結果とは一般には異なる. また, 主として計算効率上の問題のため, 分散表現多項式の係数として実際に許されるのは多項式までである. すなわち, 分母を持つ有理式は分散表現多項式の係数としては許されない.

8.8 基底変換

辞書式順序のグレブナ基底を求める場合, 直接 gr() などを起動するより, 一旦他の順序 (例えば全次数逆辞書式順序) のグレブナ基底を計算して, それを入力として辞書式順序のグレブナ基底を計算する方が効率がよい場合がある. また, 入力が何らかの順序でのグレブナ基底になっている場合, 基底変換と呼ばれる方法により, Buchberger アルゴリズムによらずに効率良く辞書式順序のグレブナ基底が計算できる場合がある. このような目的のための函数が, ユーザ定義函数として 'gr' にいくつか定義されている. 以下の 2 つの函数は, 変数順序 *vlist1*, 項順序型 *order* で既にグレブナ基底となっている多項式リスト *gbase* を, 変数順序 *vlist2* における辞書式順序のグレブナ基底に変換する函数である.

tolex(*gbase*,*vlist1*,*order*,*vlist2*)
: この函数は, *gbase* が有理数体上のシステムの場合にのみ使用可能である. この函数は, 辞書式順序のグレブナ基底を, 有限体上で計算されたグレブナ基底を雛型として, 未定係数法および Hensel 構成により求めるものである.

tolex_tl(*gbase*,*vlist1*,*order*,*vlist2*,*homo*)
: この函数は, 辞書式順序のグレブナ基底を Buchberger アルゴリズムにより求めるものであるが, 入力がある順序におけるグレブナ基底である場合の trace-lifting におけるグレブナ基底候補の頭項, 頭係数の性質を利用して, 最終的なグレブナ基底チェック, イデアルメンバシップチェックを省略しているため, 単に Buchberger アルゴリズムを繰り返すより効率よく計算できる. 更に, 入力が 0 次元システムの場合, 自動的にもう 1 つの中間的な項順序を経由して辞書式順序のグレブナ基底を計算する. 多くの場合, この方法は, 直接辞書式順序の計算を行うより効率がよい. (もちろん例外あり.) 引数 *homo* が 0 でない時, hgr() と同様に斉次化を経由して計算を行う.

その他, 0 次元システムに対し, 与えられた多項式の最小多項式を求める函数, 0 次元システムの解を, よりコンパクトに表現するための函数などが 'gr' で定義されている. これらについては個々の函数の説明を参照のこと.

8.9 Weyl 代数

これまでは, 通常の可換な多項式環におけるグレブナ基底計算について述べてきたが, グレブナ基底の理論は, ある条件を満たす非可換な環にも拡張できる. このような環の中で, 応用上も重要な, Weyl 代数, すなわち多項式環上の微分作用素環の演算およびグレブナ基底計算が Risa/Asir に実装されている.

体 K 上の n 次元 Weyl 代数 D=K<x1,...,xn,D1,...,Dn> は

xi*xj-xj*xi=0, Di*Dj-Dj*Di=0, Di*xj-xj*Di=0 (i!=j), Di*xi-xi*Di=1

という基本関係を持つ環である. D は多項式環 K[x1,...,xn] を係数とする微分作用素環で, Di は xi による微分を表す. 交換関係により, D の元は, x1^i1*...*xn^in*D1^j1*...*Dn^jn なる単項式の K 線形結合として書き表すことができる. Risa/Asir においては, この単項式を, 可換な多項式と同様に<<i1,...,in,j1,...,jn>> で表す. すなわち, D の元も分散表現多項式として表される. 加減算は, 可換の場合と同様に, +, - により実行できるが, 乗算は, 非可換性を考慮して dp_weyl_mul() という関数により実行する.

```
[0] A=<<1,2,2,1>>;
(1)*<<1,2,2,1>>
[1] B=<<2,1,1,2>>;
(1)*<<2,1,1,2>>
[2] A*B;
(1)*<<3,3,3,3>>
[3] dp_weyl_mul(A,B);
(1)*<<3,3,3,3>>+(1)*<<3,2,3,2>>+(4)*<<2,3,2,3>>+(4)*<<2,2,2,2>>
+(2)*<<1,3,1,3>>+(2)*<<1,2,1,2>>
```

グレブナ基底計算についても, Weyl 代数専用の関数として, 次の関数が用意してある. dp_weyl_gr_main(), dp_weyl_gr_mod_main(), dp_weyl_gr_f_main(), dp_weyl_f4_main(), dp_weyl_f4_mod_main(). また, 応用として, global b 関数の計算が実装されている.

8.10 グレブナ基底に関する函数

8.10.1 gr, hgr, gr_mod, dgr

gr(*plist*,*vlist*,*order*)
hgr(*plist*,*vlist*,*order*)
gr_mod(*plist*,*vlist*,*order*,*p*)
dgr(*plist*,*vlist*,*order*,*procs*)
:: グレブナ基底の計算

return リスト

plist vlist procs
リスト

order 数, リストまたは行列

p 2^27 未満の素数

- 標準ライブラリの 'gr' で定義されている.

- いずれも, 多項式リスト *plist* の, 変数順序 *vlist*, 項順序型 *order* に関するグレブナ基底を求める. gr(), hgr() は 有理数係数, gr_mod() は GF(p) 係数として計算する.
- *vlist* は不定元のリスト. *vlist* に現れない不定元は, 係数体に属すると見なされる.
- gr(), trace-lifting (モジュラ演算を用いた高速化) および sugar strategy による計算, hgr() は trace-lifting および斉次化による 矯正された sugar strategy による計算を行う.
- dgr() は, gr(), hgr() を子プロセスリスト *procs* の 2 つのプロセスにより同時に計算させ, 先に結果を返した方の結果を返す. 結果は同一であるが, どちらの方法が高速か一般には不明のため, 実際の経過時間を短縮するのに有効である.
- dgr() で表示される時間は, この函数が実行されているプロセスでの CPU 時間であり, この函数の場合はほとんど通信のための時間である.
- 多項式リスト *plist* の要素が分散表現多項式の場合は結果も分散表現多項式のリストである. この場合, 引数の分散多項式は与えられた順序に従い dp_sort でソートされてから計算される. 多項式リストの要素が分散表現多項式の場合も変数の数分の不定元のリストを *vlist* 引数として与えないといけない (ダミー).

```
[0] load("gr")$
[64] load("cyclic")$
[74] G=gr(cyclic(5),[c0,c1,c2,c3,c4],2);
[c4^15+122*c4^10-122*c4^5-1,...]
[75] GM=gr_mod(cyclic(5),[c0,c1,c2,c3,c4],2,31991)$
24628*c4^15+29453*c4^10+2538*c4^5+7363
[76] (G[0]*24628-GM[0])%31991;
0
```

参照　8.10.6 節「dp_gr_main dp_gr_mod_main dp_gr_f_main dp_weyl_gr_main dp_weyl_gr_mod_main dp_weyl_gr_f_main」p.122, 8.10.10 節「dp_ord」p.125.

8.10.2 lex_hensel, lex_tl, tolex, tolex_d, tolex_tl

lex_hensel(*plist*,*vlist1*,*order*,*vlist2*,*homo*)
lex_tl(*plist*,*vlist1*,*order*,*vlist2*,*homo*)
:: 基底変換による辞書式順序グレブナ基底の計算

tolex(*plist*,*vlist1*,*order*,*vlist2*)
tolex_d(*plist*,*vlist1*,*order*,*vlist2*,*procs*)
tolex_tl(*plist*,*vlist1*,*order*,*vlist2*,*homo*)
:: グレブナ基底を入力とする, 基底変換による辞書式順序グレブナ基底の計算

return　リスト

plist vlist1 vlist2 procs
リスト

order　数, リストまたは行列

homo　フラグ

- 標準ライブラリの 'gr' で定義されている.
- lex_hensel(), lex_tl() は, 多項式リスト *plist* の, 変数順序 *vlist1*, 項順序型 *order* に関するグレブナ基底を求め, それを, 変数順序 *vlist2* の辞書式順序グレブナ基底に変換する.

- `tolex()`, `tolex_tl()` は, 変数順序 *vlist1*, 項順序型 *order* に関するグレブナ基底である多項式リスト *plist* を変数順序 *vlist2* の辞書式順序グレブナ基底に変換する. `tolex_d()` は, `tolex()` における, 各基底の計算を, 子プロセスリスト *procs* の各プロセスに分散計算させる.
- `lex_hensel()`, `lex_tl()` においては, 辞書式順序グレブナ基底の計算は次のように行われる. (`[Noro,Yokoyama]` 参照.)
  1. *vlist1*, *order* に関するグレブナ基底 *G0* を計算する. (`lex_hensel()` のみ.)
  2. *G0* の各元の *vlist2* に関する辞書式順序における頭係数を割らないような素数 p を選び, GF(p) 上での辞書式順序グレブナ基底 Gp を計算する.
  3. Gp に現れるすべての項の, *G0* に関する正規形 *NF* を計算する.
  4. Gp の各元 f につき, f の係数を未定係数で, f の各項を対応する *NF* の元で置き換え, 各項の係数を 0 と置いた, 未定係数に関する線形方程式系 Lf を作る.
  5. Lf が, 法 p で一意解を持つことを用いて Lf の解を法 p の解から Hensel 構成により求める.
  6. すべての Gp の元につき線形方程式が解けたらその解全体が求める辞書式順序でのグレブナ基底. もしどれかの線形方程式の求解に失敗したら, p をとり直してやり直す.
- `lex_tl()`, `tolex_tl()` においては, 辞書式順序グレブナ基底の計算は次のように行われる.
  1. *vlist1*, *order* に関するグレブナ基底 *G0* を計算する. (`lex_hensel()` のみ.)
  2. *G0* が 0 次元システムでないとき, *G0* を入力として, *G0* の各元の *vlist2* に関する辞書式順序における頭係数を割らないような素数 p を選び, p を用いた trace-lifting により辞書式順序のグレブナ基底候補を求め, もし求まったならチェックなしにそれが求めるグレブナ基底となる. もし失敗したら, p をとり直してやり直す.
  3. *G0* が 0 次元システムのとき, *G0* を入力として, まず, *vlist2* の最後の変数以外を消去する消去順序によりグレブナ基底 *G1* を計算し, それから辞書式順序のグレブナ基底を計算する. その際, 各ステップでは, 入力の各元の, 求める順序における頭係数を割らない素数を用いた trace-lifting でグレブナ基底候補を求め, もし求まったらチェックなしにそれがその順序でのグレブナ基底となる.
- 有理式係数の計算は, `lex_tl()`, `tolex_tl()` のみ受け付ける.
- `homo` が 0 でない場合, 内部で起動される Buchberger アルゴリズムにおいて, 斉次化が行われる.
- `tolex_d()` で表示される時間は, この函数が実行されているプロセスにおいて行われた計算に対応していて, 子プロセスにおける時間は含まれない.

```
[78] K=katsura(5)$
30msec + gc : 20msec
[79] V=[u5,u4,u3,u2,u1,u0]$
0msec
[80] G0=hgr(K,V,2)$
91.558sec + gc : 15.583sec
[81] G1=lex_hensel(K,V,0,V,0)$
49.049sec + gc : 9.961sec
[82] G2=lex_tl(K,V,0,V,1)$
31.186sec + gc : 3.500sec
```

```
[83] gb_comp(G0,G1);
1
10msec
[84] gb_comp(G0,G2);
1
```

参照　8.10.6 節「dp_gr_main dp_gr_mod_main dp_gr_f_main dp_weyl_gr_main dp_weyl_gr_mod_main dp_weyl_gr_f_main」p.122, 8.10.10 節「dp_ord」p.125, 第 7 章「分散計算」p.91

8.10.3 lex_hensel_gsl, tolex_gsl, tolex_gsl_d

lex_hensel_gsl(*plist*,*vlist1*,*order*,*vlist2*,*homo*)
:: GSL 形式のイデアル基底の計算

tolex_gsl(*plist*,*vlist1*,*order*,*vlist2*)
tolex_gsl_d(*plist*,*vlist1*,*order*,*vlist2*,*procs*)
:: グレブナ基底を入力とする, GSL 形式のイデアル基底の計算

return　リスト

plist vlist1 vlist2 procs
リスト

order　数, リストまたは行列

homo　フラグ

- lex_hensel_gsl() は lex_hensel() の, tolex_gsl() は tolex() の変種で, 結果のみが異なる. tolex_gsl_d() は, 基底計算を, procs で指定される子プロセスに分散計算させる.
- 入力が 0 次元システムで, その辞書式順序グレブナ基底が [f0,x1-f1,...,xn-fn] (f0,...,fn は x0 の 1 変数多項式) なる形 (これを SL 形式と呼ぶ) を持つ場合, [[x1,g1,d1],...,[xn,gn,dn],[x0,f0,f0']] なるリスト (これを GSL 形式と呼ぶ) を返す. ここで, gi は, di*f0'*fi-gi が f0 で割り切れるような x0 の 1 変数多項式で, 解は f0(x0)=0 なる x0 に対し, [x1=g1/(d1*f0'),...,xn=gn/(dn*f0')] となる. 辞書式順序グレブナ基底が上のような形でない場合, tolex() による通常のグレブナ基底を返す.
- GSL 形式により表される基底はグレブナ基底ではないが, 一般に係数が SL 形式のグレブナ基底より非常に小さいため計算も速く, 解も求めやすい. tolex_gsl_d() で表示される時間は, この函数が実行されているプロセスにおいて行われた計算に対応していて, 子プロセスにおける時間は含まれない.

```
[103] K=katsura(5)$
[104] V=[u5,u4,u3,u2,u1,u0]$
[105] G0=gr(K,V,0)$
[106] GSL=tolex_gsl(G0,V,0,V)$
[107] GSL[0];
[u1,8635837421130477667200000000*u0^31-...]
[108] GSL[1];
[u2,10352277157007342793600000000*u0^31-...]
[109] GSL[5];
```

```
[u0,1177102187619306412464000000*u0^32-...,
37667270003817805198848000000*u0^31-...]
```

参照　8.10.2 節「lex_hensel lex_tl tolex tolex_d tolex_tl」p.118, 第 7 章「分散計算」p.91

8.10.4 gr_minipoly, minipoly

gr_minipoly(*plist*,*vlist*,*order*,*poly*,*v*,*homo*)
:: 多項式の, イデアルを法とした最小多項式の計算

minipoly(*plist*,*vlist*,*order*,*poly*,*v*)
:: グレブナ基底を入力とする, 多項式の最小多項式の計算

return　多項式

plist vlist　リスト

order　数, リストまたは行列

poly　多項式

v　不定元

homo　フラグ

- gr_minipoly() はグレブナ基底の計算から行い, minipoly() は入力をグレブナ基底とみなす.
- イデアル I が体 K 上の多項式環 K[X] の 0 次元イデアルの時, K[v] の元 f(v) に f(p) mod I を対応させる環準同型の核は 0 でない多項式により生成される. この生成元を p の, 法 I での最小多項式と呼ぶ.
- gr_minipoly(), minipoly() は, 多項式 p の最小多項式を求め, v を変数とする多項式として返す.
- 最小多項式は, グレブナ基底の 1 つの元として計算することもできるが, 最小多項式のみを求めたい場合, minipoly(), gr_minipoly() はグレブナ基底を用いる方法に比べて効率がよい.
- gr_minipoly() に指定する項順序としては, 通常全次数逆辞書式順序を用いる.

```
[117] G=tolex(G0,V,0,V)$
43.818sec + gc : 11.202sec
[118] GSL=tolex_gsl(G0,V,0,V)$
17.123sec + gc : 2.590sec
[119] MP=minipoly(G0,V,0,u0,z)$
4.370sec + gc : 780msec
```

参照　8.10.2 節「lex_hensel lex_tl tolex tolex_d tolex_tl」p.118.

8.10.5 tolexm, minipolym

tolexm(*plist*,*vlist1*,*order*,*vlist2*,*mod*)
:: 法 *mod* での基底変換によるグレブナ基底計算

minipolym(*plist*,*vlist1*,*order*,*poly*,*v*,*mod*)
:: 法 *mod* でのグレブナ基底による多項式の最小多項式の計算

return　tolexm() : リスト, minipolym() : 多項式

plist vlist1 vlist2
　リスト

order　数, リストまたは行列

mod　素数

- 入力 *plist* はいずれも 変数順序 *vlist1*, 項順序型 *order*, 法 *mod* におけるグレブナ基底でなければならない.
- minipolym() は minipoly に対応する計算を法 *mod* で行う.
- tolexm() は FGLM 法による基底変換により *vlist2*, 辞書式順序によるグレブナ基底を計算する.

```
[197] tolexm(G0,V,0,V,31991);
[8271*u0^31+10435*u0^30+816*u0^29+26809*u0^28+...,...]
[198] minipolym(G0,V,0,u0,z,31991);
z^32+11405*z^31+20868*z^30+21602*z^29+...
```

参照　8.10.2 節「lex_hensel lex_tl tolex tolex_d tolex_tl」p.118, 8.10.4 節「gr_minipoly minipoly」p.121.

8.10.6 dp_gr_main, dp_gr_mod_main, dp_gr_f_main, dp_weyl_gr_main, dp_weyl_gr_mod_main, dp_weyl_gr_f_main

dp_gr_main(*plist*,*vlist*,*homo*,*modular*,*order*)
dp_gr_mod_main(*plist*,*vlist*,*homo*,*modular*,*order*)
dp_gr_f_main(*plist*,*vlist*,*homo*,*order*)
dp_weyl_gr_main(*plist*,*vlist*,*homo*,*modular*,*order*)
dp_weyl_gr_mod_main(*plist*,*vlist*,*homo*,*modular*,*order*)
dp_weyl_gr_f_main(*plist*,*vlist*,*homo*,*order*)
　:: グレブナ基底の計算 (組み込み函数)

return　リスト

plist vlist　リスト

order　数, リストまたは行列

homo　フラグ

modular　フラグまたは素数

- これらの函数は, グレブナ基底計算の基本的組み込み函数であり, gr(), hgr(), gr_mod() などはすべてこれらの函数を呼び出して計算を行っている. 関数名に weyl が入っているものは, Weyl 代数上の計算のための関数である.
- dp_gr_f_main(), dp_weyl_f_main() は, 種々の有限体上のグレブナ基底を計算する場合に用いる. 入力は, あらかじめ, simp_ff() などで, 考える有限体上に射影されている必要がある.
- フラグ *homo* が 0 でない時, 入力を斉次化してから Buchberger アルゴリズムを実行する.
- dp_gr_mod_main() に対しては, *modular* は, GF(*modular*) 上での計算を意味する. dp_gr_main() に対しては, *modular* は次のような意味を持つ.

1. *modular* が 1 の時, trace-lifting による計算を行う. 素数は `lprime(0)` から順に成功するまで `lprime()` を呼び出して生成する.
2. *modular* が 2 以上の自然数の時, その値を素数とみなして trace-lifting を行う. その素数で失敗した場合, 0 を返す.
3. *modular* が負の場合, -*modular* に対して上述の規則が適用されるが, trace-lifting の最終段階のグレブナ基底チェックとイデアルメンバシップチェックが省略される.

- `gr(P,V,O)` は `dp_gr_main(P,V,0,1,O)`, `hgr(P,V,O)` は `dp_gr_main(P,V,1,1,O)`, `gr_mod(P,V,O,M)` は `dp_gr_mod_main(P,V,0,M,O)` をそれぞれ実行する.
- *homo*, *modular* の他に, `dp_gr_flags()` で設定されるさまざまなフラグにより計算が制御される.

参照　8.10.10 節「`dp_ord`」p.125, 8.10.9 節「`dp_gr_flags dp_gr_print`」p.124, 8.10.1 節「`gr hgr gr_mod`」p.117, 10.5.1 節「`setmod_ff`」p.152, 8.4 節「計算および表示の制御」p.110.

8.10.7 dp_f4_main, dp_f4_mod_main, dp_weyl_f4_main, dp_weyl_f4_mod_main

`dp_f4_main`(*plist*,*vlist*,*order*)
`dp_f4_mod_main`(*plist*,*vlist*,*order*)
`dp_weyl_f4_main`(*plist*,*vlist*,*order*)
`dp_weyl_f4_mod_main`(*plist*,*vlist*,*order*)
:: F4 アルゴリズムによるグレブナ基底の計算 (組み込み函数)

return　リスト

plist vlist　リスト

order　数, リストまたは行列

- F4 アルゴリズムによりグレブナ基底の計算を行う.
- F4 アルゴリズムは, J.C. Faugere により提唱された新世代グレブナ基底算法であり, 本実装は, 中国剰余定理による線形方程式求解を用いた試験的な実装である.
- 斉次化の引数がないことを除けば, 引数および動作はそれぞれ `dp_gr_main()`, `dp_gr_mod_main()`, `dp_weyl_gr_main()`, `dp_weyl_gr_mod_main()` と同様である.

参照　8.10.10 節「`dp_ord`」p.125, 8.10.9 節「`dp_gr_flags dp_gr_print`」p.124, 8.10.1 節「`gr hgr gr_mod`」p.117, 8.4 節「計算および表示の制御」p.110.

8.10.8 nd_gr, nd_gr_trace, nd_f4, nd_weyl_gr, nd_weyl_gr_trace

`nd_gr`(*plist*,*vlist*,*p*,*order*)
`nd_gr_trace`(*plist*,*vlist*,*homo*,*p*,*order*)
`nd_f4`(*plist*,*vlist*,*modular*,*order*)
`nd_weyl_gr`(*plist*,*vlist*,*p*,*order*)
`nd_weyl_gr_trace`(*plist*,*vlist*,*homo*,*p*,*order*)
:: グレブナ基底の計算 (組み込み函数)

return　リスト

plist vlist　リスト

order　数, リストまたは行列

homo　フラグ

modular　フラグまたは素数

- これらの函数は, グレブナ基底計算組み込み関数の新実装である.
- nd_gr は, p が 0 のとき有理数体上の Buchberger アルゴリズムを実行する. p が 2 以上の自然数のとき, GF(p) 上の Buchberger アルゴリズムを実行する.
- nd_gr_trace は有理数体上で trace アルゴリズムを実行する. p が 0 または 1 のとき, 自動的に選ばれた素数を用いて, 成功するまで trace アルゴリズムを実行する. p が 2 以上のとき, trace は GF(p) 上で計算される. trace アルゴリズムが失敗した場合 0 が返される. p が負の場合, グレブナ基底チェックは行わない. この場合, p が -1 ならば自動的に選ばれた素数が, それ以外は指定された素数を用いてグレブナ基底候補の計算が行われる.
- nd_f4 は, 有限体上の F4 アルゴリズムを実行する.
- nd_weyl_gr, nd_weyl_gr_trace は Weyl 代数用である.
- いずれの関数も, 有理関数体上の計算は未対応である.
- 一般に dp_gr_main, dp_gr_mod_main より高速であるが, 特に有限体上の場合顕著である.

```
[38] load("cyclic")$
[49] C=cyclic(7)$
[50] V=vars(C)$
[51] cputime(1)$
[52] dp_gr_mod_main(C,V,0,31991,0)$
26.06sec + gc : 0.313sec(26.4sec)
[53] nd_gr(C,V,31991,0)$
ndv_alloc=1477188
5.737sec + gc : 0.1837sec(5.921sec)
[54] dp_f4_mod_main(C,V,31991,0)$
3.51sec + gc : 0.7109sec(4.221sec)
[55] nd_f4(C,V,31991,0)$
1.906sec + gc : 0.126sec(2.032sec)
```

参照　8.10.10 節「dp_ord」p.125, 8.10.9 節「dp_gr_flags dp_gr_print」p.124, 8.4 節「計算および表示の制御」p.110.

8.10.9 dp_gr_flags, dp_gr_print

dp_gr_flags([*list*])
dp_gr_print([*i*])
:: 計算および表示用パラメタの設定, 参照

return　設定値

list　リスト

i　整数

- dp_gr_main(), dp_gr_mod_main(), dp_gr_f_main() 実行時におけるさまざまなパラメタを設定, 参照する.
- 引数がない場合, 現在の設定が返される.
- 引数は, ["Print",1,"NoSugar",1,...] なる形のリストで, 左から順に設定される. パラメタ名は文字列で与える必要がある.
- dp_gr_print() は, 特にパラメタ Print, PrintShort の値を直接設定, 参照できる. 設定される値は次の通りである。

  i=0　　Print=0, PrintShort=0

  i=1　　Print=1, PrintShort=0

  i=2　　Print=0, PrintShort=1

  これは, dp_gr_main() などをサブルーチンとして用いるユーザ函数において, そのサブルーチンが中間情報の表示を行う際に, 迅速にフラグを見ることができるように用意されている.

参照　　8.4 節「計算および表示の制御」p.110

8.10.10 dp_ord

dp_ord([*order*])
:: 変数順序型の設定, 参照

return　　変数順序型 (数, リストまたは行列)

order　　数, リストまたは行列

- 引数がある時, 変数順序型を *order* に設定する. 引数がない時, 現在設定されている変数順序型を返す.
- 分散表現多項式に関する函数, 演算は引数として変数順序型をとるものととらないものがあり, とらないものに関しては, その時点で設定されている値を用いて計算が行われる.
- gr() など, 引数として変数順序型をとるものは, 内部で dp_ord() を呼び出し, 変数順序型を設定する. この設定は, 計算終了後も生き残る.
- 分散表現多項式の四則演算も, 設定されている値を用いて計算される. 従って, その多項式が生成された時点における変数順序型が, 四則演算時に正しく設定されていなければならない. また, 演算対象となる多項式は, 同一の変数順序型に基づいて生成されたものでなければならない.
- トップレベル函数以外の函数を直接呼び出す場合には, この函数により変数順序型を正しく設定しなければならない.

```
[19] dp_ord(0)$
[20] <<1,2,3>>+<<3,1,1>>;
(1)*<<1,2,3>>+(1)*<<3,1,1>>
[21] dp_ord(2)$
[22] <<1,2,3>>+<<3,1,1>>;
(1)*<<3,1,1>>+(1)*<<1,2,3>>
```

参照　　8.5 節「項順序の設定」p.113

8.10.11 dp_ptod

dp_ptod(*poly*,*vlist*)
:: 多項式を分散表現多項式に変換する.

return　分散表現多項式

poly　多項式

vlist　リスト

- 変数順序 *vlist* および現在の変数順序型に従って分散表現多項式に変換する.
- *vlist* に含まれない不定元は, 係数体に属するとして変換される.

```
[50] dp_ord(0);
1
[51] dp_ptod((x+y+z)^2,[x,y,z]);
(1)*<<2,0,0>>+(2)*<<1,1,0>>+(1)*<<0,2,0>>+(2)*<<1,0,1>>+(2)*<<0,1,1>>
+(1)*<<0,0,2>>
[52] dp_ptod((x+y+z)^2,[x,y]);
(1)*<<2,0>>+(2)*<<1,1>>+(1)*<<0,2>>+(2*z)*<<1,0>>+(2*z)*<<0,1>>
+(z^2)*<<0,0>>
```

参照　8.10.12 節「dp_dtop」p.126, 8.10.10 節「dp_ord」p.125.

8.10.12 dp_dtop

dp_dtop(*dpoly*,*vlist*)
:: 分散表現多項式を多項式に変換する.

return　多項式

dpoly　分散表現多項式

vlist　リスト

- 分散表現多項式を, 与えられた不定元リストを用いて多項式に変換する.
- 不定元リストは, 長さ分散表現多項式の変数の個数と一致していれば何でもよい.

```
[53] T=dp_ptod((x+y+z)^2,[x,y]);
(1)*<<2,0>>+(2)*<<1,1>>+(1)*<<0,2>>+(2*z)*<<1,0>>+(2*z)*<<0,1>>
+(z^2)*<<0,0>>
[54] P=dp_dtop(T,[a,b]);
z^2+(2*a+2*b)*z+a^2+2*b*a+b^2
```

8.10.13 dp_mod, dp_rat

dp_mod(*p*,*mod*,*subst*)
:: 有理数係数分散表現多項式の有限体係数への変換

dp_rat(*p*)
:: 有限体係数分散表現多項式の有理数係数への変換

return　分散表現多項式

p　分散表現多項式

mod　　素数

subst　　リスト

- dp_nf_mod(), dp_true_nf_mod() は, 入力として有限体係数の分散表現多項式を必要とする. このような場合, dp_mod() により有理数係数分散表現多項式を変換して用いることができる. また, 得られた結果は, 有限体係数多項式とは演算できるが, 有理数係数多項式とは演算できないため, dp_rat() により変換する必要がある.
- 有限体係数の演算においては, あらかじめ setmod() により有限体の元の個数を指定しておく必要がある.
- *subst* は, 係数が有理式の場合, その有理式の変数にあらかじめ数を代入した後有限体係数に変換するという操作を行う際の, 代入値を指定するもので, [[*var*,*value*],...] の形のリストである.

参照　　8.10.16 節「dp_nf dp_nf_mod dp_true_nf dp_true_nf_mod」p.128, 6.3.11 節「subst psubst」p.48, 6.1.16 節「setmod」p.40.

8.10.14 dp_homo, dp_dehomo

dp_homo(*dpoly*)
:: 分散表現多項式の斉次化

dp_dehomo(*dpoly*)
:: 斉次分散表現多項式の非斉次化

return　　分散表現多項式

dpoly　　分散表現多項式

- dp_homo() は, *dpoly* の 各項 t について, 指数ベクトルの長さを 1 伸ばし, 最後の成分の値を d-deg(t) (d は *dpoly* の全次数) とした分散表現多項式を返す.
- dp_dehomo() は, *dpoly* の各項について, 指数ベクトルの最後の成分を取り除いた分散多項式を返す.
- いずれも, 生成された多項式を用いた演算を行う場合, それらに適合する項順序を正しく設定する必要がある.
- hgr() などにおいて, 内部的に用いられている.

```
[202] X=<<1,2,3>>+3*<<1,2,1>>;
(1)*<<1,2,3>>+(3)*<<1,2,1>>
[203] dp_homo(X);
(1)*<<1,2,3,0>>+(3)*<<1,2,1,2>>
[204] dp_dehomo(@);
(1)*<<1,2,3>>+(3)*<<1,2,1>>
```

参照　　8.10.1 節「gr hgr gr_mod」p.117.

8.10.15 dp_ptozp, dp_prim

dp_ptozp(*dpoly*)
:: 定数倍して係数を整数係数かつ係数の整数 GCD を 1 にする.

dp_prim(*dpoly*)
:: 有理式倍して係数を整数係数多項式係数かつ係数の多項式 GCD を 1 にする.

return　分散表現多項式

dpoly　分散表現多項式

- dp_ptozp() は, ptozp() に相当する操作を分散表現多項式に対して行う. 係数が多項式を含む場合, 係数に含まれる多項式共通因子は取り除かない.
- dp_prim() は, 係数が多項式を含む場合, 係数に含まれる多項式共通因子を取り除く.

```
[208] X=dp_ptod(3*(x-y)*(y-z)*(z-x),[x]);
(-3*y+3*z)*<<2>>+(3*y^2-3*z^2)*<<1>>+(-3*z*y^2+3*z^2*y)*<<0>>
[209] dp_ptozp(X);
(-y+z)*<<2>>+(y^2-z^2)*<<1>>+(-z*y^2+z^2*y)*<<0>>
[210] dp_prim(X);
(1)*<<2>>+(-y-z)*<<1>>+(z*y)*<<0>>
```

参照　6.3.18 節「ptozp」p.52.

8.10.16 dp_nf, dp_nf_mod, dp_true_nf, dp_true_nf_mod

dp_nf(*indexlist*,*dpoly*,*dpolyarray*,*fullreduce*)

dp_nf_mod(*indexlist*,*dpoly*,*dpolyarray*,*fullreduce*,*mod*)
:: 分散表現多項式の正規形を求める. (結果は定数倍されている可能性あり)

dp_true_nf(*indexlist*,*dpoly*,*dpolyarray*,*fullreduce*)

dp_true_nf_mod(*indexlist*,*dpoly*,*dpolyarray*,*fullreduce*,*mod*)
:: 分散表現多項式の正規形を求める. (真の結果を [分子, 分母] の形で返す)

return　dp_nf() : 分散表現多項式, dp_true_nf() : リスト

indexlist　リスト

dpoly　分散表現多項式

dpolyarray
配列

fullreduce　フラグ

mod　素数

- 分散表現多項式 *dpoly* の正規形を求める.
- dp_nf_mod(), dp_true_nf_mod() の入力は, dp_mod() などにより, 有限体上の分散表現多項式になっていなければならない.
- 結果に有理数, 有理式が含まれるのを避けるため, dp_nf() は真の値の定数倍の値を返す. 有理式係数の場合の dp_nf_mod() も同様であるが, 係数体が有限体の場合 dp_nf_mod() は真の値を返す.
- dp_true_nf(), dp_true_nf_mod() は, [*nm*,*dn*] なる形のリストを返す. ただし, *nm* は係数に分数, 有理式を含まない分散表現多項式, *dn* は数または多項式で nm/dn が真の値となる.
- *dpolyarray* は分散表現多項式を要素とするベクトル, *indexlist* は正規化計算に用いる *dpolyarray* の要素のインデックスのリスト.
- *fullreduce* が 0 でないとき全ての項に対して簡約を行う. *fullreduce* が 0 のとき頭項のみに対して簡約を行う.

- *indexlist* で指定された多項式は, 前の方のものが優先的に使われる.
- 一般には *indexlist* の与え方により函数の値は異なる可能性があるが, グレブナ基底に対しては一意的に定まる.
- 分散表現でない固定された多項式集合による正規形を多数求める必要がある場合に便利である. 単一の演算に関しては, `p_nf`, `p_true_nf` を用いるとよい.

```
[0] load("gr")$
[64] load("katsura")$
[69] K=katsura(4)$
[70] dp_ord(2)$
[71] V=[u0,u1,u2,u3,u4]$
[72] DP1=newvect(length(K),map(dp_ptod,K,V))$
[73] G=gr(K,V,2)$
[74] DP2=newvect(length(G),map(dp_ptod,G,V))$
[75] T=dp_ptod((u0-u1+u2-u3+u4)^2,V)$
[76] dp_dtop(dp_nf([0,1,2,3,4],T,DP1,1),V);
u4^2+(6*u3+2*u2+6*u1-2)*u4+9*u3^2+(6*u2+18*u1-6)*u3+u2^2
+(6*u1-2)*u2+9*u1^2-6*u1+1
[77] dp_dtop(dp_nf([4,3,2,1,0],T,DP1,1),V);
-5*u4^2+(-4*u3-4*u2-4*u1)*u4-u3^2-3*u3-u2^2+(2*u1-1)*u2-2*u1^2-3*u1+1
[78] dp_dtop(dp_nf([0,1,2,3,4],T,DP2,1),V);
-113808797684516577808861229727307852034709700102047145563335304922104
5677593000571650556006208715092840087615021707982031143947756058758348
8*u4^15+...
[79] dp_dtop(dp_nf([4,3,2,1,0],T,DP2,1),V);
-113808797684516577808861229727307852034709700102047145563335304922104
5677593000571650556006208715092840087615021707982031143947756058758348
8*u4^15+...
[80] @78==@79;
1
```

参照　8.10.12 節「`dp_dtop`」p.126, 8.10.10 節「`dp_ord`」p.125, 8.10.13 節「`dp_mod dp_rat`」p.126, 8.10.27 節「`p_nf p_nf_mod p_true_nf p_true_nf_mod`」p.134.

8.10.17 dp_hm, dp_ht, dp_hc, dp_rest

`dp_hm(`*dpoly*`)`
:: 頭単項式を取り出す.

`dp_ht(`*dpoly*`)`
:: 頭項を取り出す.

`dp_hc(`*dpoly*`)`
:: 頭係数を取り出す.

`dp_rest(`*dpoly*`)`
:: 頭単項式を取り除いた残りを返す.

return　`dp_hm()`, `dp_ht()`, `dp_rest()` : 分散表現多項式, `dp_hc()` : 数または多項式

dpoly　分散表現多項式

- これらは, 分散表現多項式の各部分を取り出すための函数である.

- 分散表現多項式 p に対し次が成り立つ.

  p = dp_hm(p) + dp_rest(p)

  dp_hm(p) = dp_hc(p) dp_ht(p)

```
 [87] dp_ord(0)$
 [88] X=ptozp((a46^2+7/10*a46+7/48)*u3^4-50/27*a46^2-35/27*a46-49/216)$
 [89] T=dp_ptod(X,[u3,u4,a46])$
 [90] dp_hm(T);
 (2160)*<<4,0,2>>
 [91] dp_ht(T);
 (1)*<<4,0,2>>
 [92] dp_hc(T);
 2160
 [93] dp_rest(T);
 (1512)*<<4,0,1>>+(315)*<<4,0,0>>+(-4000)*<<0,0,2>>+(-2800)*<<0,0,1>>
 +(-490)*<<0,0,0>>
```

8.10.18 dp_td, dp_sugar

dp_td(*dpoly*)
:: 頭項の全次数を返す.

dp_sugar(*dpoly*)
:: 多項式の sugar を返す.

return　自然数

dpoly　分散表現多項式

onoff　フラグ

- dp_td() は, 頭項の全次数, すなわち各変数の指数の和を返す.
- 分散表現多項式が生成されると, sugar と呼ばれるある整数が付与される. この値は 仮想的に斉次化して計算した場合に結果が持つ全次数の値となる.
- sugar は, グレブナ基底計算における正規化対の選択のストラテジを決定するための重要な指針となる.

```
 [74] dp_ord(0)$
 [75] X=<<1,2>>+<<0,1>>$
 [76] Y=<<1,2>>+<<1,0>>$
 [77] Z=X-Y;
 (-1)*<<1,0>>+(1)*<<0,1>>
 [78] dp_sugar(T);
 3
```

8.10.19 dp_lcm

dp_lcm(*dpoly1*,*dpoly2*)
:: 最小公倍項を返す.

return　分散表現多項式

dpoly1 dpoly2
　　分散表現多項式

- それぞれの引数の頭項の最小公倍項を返す. 係数は 1 である.

```
[100] dp_lcm(<<1,2,3,4,5>>,<<5,4,3,2,1>>);
(1)*<<5,4,3,4,5>>
```

参照　　8.10.27 節「p_nf p_nf_mod p_true_nf p_true_nf_mod」p.134.

8.10.20 dp_redble

dp_redble(*dpoly1*,*dpoly2*)
　　:: 頭項どうしが整除可能かどうか調べる.

return　　整数

dpoly1 dpoly2
　　分散表現多項式

- *dpoly1* の頭項が *dpoly2* の頭項で割り切れれば 1, 割り切れなければ 0 を返す.
- 多項式の簡約を行う際, どの項を簡約できるかを探すのに用いる.

```
[148] C;
(1)*<<1,1,1,0,0>>+(1)*<<0,1,1,1,0>>+(1)*<<1,1,0,0,1>>+(1)*<<1,0,0,1,1>>
[149] T;
(3)*<<2,1,0,0,0>>+(3)*<<1,2,0,0,0>>+(1)*<<0,3,0,0,0>>+(6)*<<1,1,1,0,0>>
[150] for ( ; T; T = dp_rest(T)) print(dp_redble(T,C));
0
0
0
1
```

参照　　8.10.25 節「dp_red dp_red_mod」p.133.

8.10.21 dp_subd

dp_subd(*dpoly1*,*dpoly2*)
　　:: 頭項の商単項式を返す.

return　　分散表現多項式

dpoly1 dpoly2
　　分散表現多項式

- dp_ht(*dpoly1*)/dp_ht(*dpoly2*) を求める. 結果の係数は 1 である.
- 割り切れることがあらかじめわかっている必要がある.

```
[162] dp_subd(<<1,2,3,4,5>>,<<1,1,2,3,4>>);
(1)*<<0,1,1,1,1>>
```

参照　　8.10.25 節「dp_red dp_red_mod」p.133.

8.10.22 dp_vtoe, dp_etov

dp_vtoe(*vect*)
:: 指数ベクトルを項に変換

dp_etov(*dpoly*)
:: 頭項を指数ベクトルに変換

return　dp_vtoe：分散表現多項式, dp_etov：ベクトル

vect　ベクトル

dpoly　分散表現多項式

- dp_vtoe() は, ベクトル *vect* を指数ベクトルとする項を生成する.
- dp_etov() は, 分散表現多項式 dpoly の頭項の指数ベクトルをベクトルに変換する.

```
[211] X=<<1,2,3>>;
(1)*<<1,2,3>>
[212] V=dp_etov(X);
[ 1 2 3 ]
[213] V[2]++$
[214] Y=dp_vtoe(V);
(1)*<<1,2,4>>
```

8.10.23 dp_mbase

dp_mbase(*dplist*)
:: monomial 基底の計算

return　分散表現多項式のリスト

dplist　分散表現多項式のリスト

- ある順序でグレブナ基底となっている多項式集合の, その順序に関する分散表現である *dplist* について, *dplist* が K[X] 中で生成するイデアル I が 0 次元の時, K 上有限次元線形空間である K[X]/I の monomial による基底を求める.
- 得られた基底の個数が, K[X]/I の K-線形空間としての次元に等しい.

```
[215] K=katsura(5)$
[216] V=[u5,u4,u3,u2,u1,u0]$
[217] G0=gr(K,V,0)$
[218] H=map(dp_ptod,G0,V)$
[219] map(dp_ptod,dp_mbase(H),V)$
[u0^5,u4*u0^3,u3*u0^3,u2*u0^3,u1*u0^3,u0^4,u3^2*u0,u2*u3*u0,u1*u3*u0,
u1*u2*u0,u1^2*u0,u4*u0^2,u3*u0^2,u2*u0^2,u1*u0^2,u0^3,u3^2,u2*u3,u1*u3,
u1*u2,u1^2,u4*u0,u3*u0,u2*u0,u1*u0,u0^2,u4,u3,u2,u1,u0,1]
```

参照　8.10.1 節「gr hgr gr_mod」p.117.

8.10.24 dp_mag

dp_mag(*p*)
:: 係数のビット長の和を返す

return　数

p　分散表現多項式

- 分散表現多項式の係数に現れる有理数につき, その分母分子 (整数の場合は分子) のビット長の総和を返す.
- 対象となる多項式の大きさの目安として有効である. 特に, 0 次元システムにおいては係数膨張が問題となり, 途中生成される多項式が係数膨張を起こしているかどうかの判定に役立つ.
- `dp_gr_flags()` で, `ShowMag`, `Print` を on にすることにより途中生成される多項式にたいする `dp_mag()` の値を見ることができる.

```
[221] X=dp_ptod((x+2*y)^10,[x,y])$
[222] dp_mag(X);
115
```

参照　8.10.9 節「dp_gr_flags dp_gr_print」p.124.

8.10.25 dp_red, dp_red_mod

`dp_red(`*dpoly1*`,`*dpoly2*`,`*dpoly3*`)`

`dp_red_mod(`*dpoly1*`,`*dpoly2*`,`*dpoly3*`,`*mod*`)`
:: 一回の簡約操作

return　リスト

dpoly1 dpoly2 dpoly3
分散表現多項式

vlist　リスト

mod　素数

- *dpoly1* + *dpoly2* なる分散表現多項式を *dpoly3* で 1 回簡約する.
- `dp_red_mod()` の入力は, 全て有限体係数に変換されている必要がある.
- 簡約される項は *dpoly2* の頭項である. 従って, *dpoly2* の頭項が *dpoly3* の頭項で割り切れることがあらかじめわかっていなければならない.
- 引数が整数係数の時, 簡約は, 分数が現れないよう, 整数 a, *b*, 項 *t* により a(*dpoly1* + *dpoly2*)-*bt dpoly3* として計算される.
- 結果は, [a *dpoly1*,a *dpoly2* - *bt dpoly3*] なるリストである.

```
[157] D=(3)*<<2,1,0,0,0>>+(3)*<<1,2,0,0,0>>+(1)*<<0,3,0,0,0>>;
(3)*<<2,1,0,0,0>>+(3)*<<1,2,0,0,0>>+(1)*<<0,3,0,0,0>>
[158] R=(6)*<<1,1,1,0,0>>;
(6)*<<1,1,1,0,0>>
[159] C=12*<<1,1,1,0,0>>+(1)*<<0,1,1,1,0>>+(1)*<<1,1,0,0,1>>;
(12)*<<1,1,1,0,0>>+(1)*<<0,1,1,1,0>>+(1)*<<1,1,0,0,1>>
[160] dp_red(D,R,C);
[(6)*<<2,1,0,0,0>>+(6)*<<1,2,0,0,0>>+(2)*<<0,3,0,0,0>>,
(-1)*<<0,1,1,1,0>>+(-1)*<<1,1,0,0,1>>]
```

参照　8.10.13 節「dp_mod dp_rat」p.126.

8.10.26 dp_sp, dp_sp_mod

dp_sp(*dpoly1*,*dpoly2*)

dp_sp_mod(*dpoly1*,*dpoly2*,*mod*)
:: S-多項式の計算

return　分散表現多項式

dpoly1 dpoly2
分散表現多項式

mod　素数

- *dpoly1*, *dpoly2* の S-多項式を計算する.
- dp_sp_mod() の入力は, 全て有限体係数に変換されている必要がある.
- 結果に有理数, 有理式が入るのを避けるため, 結果が定数倍, あるいは多項式倍されている可能性がある.

```
[227] X=dp_ptod(x^2*y+x*y,[x,y]);
(1)*<<2,1>>+(1)*<<1,1>>
[228] Y=dp_ptod(x*y^2+x*y,[x,y]);
(1)*<<1,2>>+(1)*<<1,1>>
[229] dp_sp(X,Y);
(-1)*<<2,1>>+(1)*<<1,2>>
```

参照　8.10.13 節「dp_mod dp_rat」p.126.

8.10.27 p_nf, p_nf_mod, p_true_nf, p_true_nf_mod

p_nf(*poly*,*plist*,*vlist*,*order*)
p_nf_mod(*poly*,*plist*,*vlist*,*order*,*mod*)
:: 表現多項式の正規形を求める. (結果は定数倍されている可能性あり)

p_true_nf(*poly*,*plist*,*vlist*,*order*)
p_true_nf_mod(*poly*,*plist*,*vlist*,*order*,*mod*)
:: 表現多項式の正規形を求める. (真の結果を [分子, 分母] の形で返す)

return　p_nf : 多項式, p_true_nf : リスト

poly　多項式

plist vlist　リスト

order　数, リストまたは行列

mod　素数

- 'gr' で定義されている.
- 多項式の, 多項式リストによる正規形を求める.
- dp_nf(), dp_true_nf(), dp_nf_mod(), dp_true_nf_mod に対するインタフェースである.
- *poly* および *plist* は, 変数順序 *vlist* および変数順序型 *otype* に従って分散表現多項式に変換され, dp_nf(), dp_true_nf(), dp_nf_mod(), dp_true_nf_mod() に渡される.

- dp_nf(), dp_true_nf(), dp_nf_mod(), dp_true_nf_mod() は *fullreduce* が 1 で呼び出される.
- 結果は多項式に変換されて出力される.
- p_true_nf(), p_true_nf_mod() の出力に関しては, dp_true_nf(), dp_true_nf_mod() の項を参照.

```
[79] K = katsura(5)$
[80] V = [u5,u4,u3,u2,u1,u0]$
[81] G = hgr(K,V,2)$
[82] p_nf(K[1],G,V,2);
0
[83] L = p_true_nf(K[1]+1,G,V,2);
[-1503...,-1503...]
[84] L[0]/L[1];
1
```

参照　8.10.11 節「dp_ptod」p.126, 8.10.12 節「dp_dtop」p.126, 8.10.10 節「dp_ord」p.125, 8.10.16 節「dp_nf dp_nf_mod dp_true_nf dp_true_nf_mod」p.128.

8.10.28 p_terms

p_terms(*poly*,*vlist*,*order*)
:: 多項式にあらわれる単項をリストにする.

return　リスト

poly　多項式

vlist　リスト

order　数, リストまたは行列

- 'gr' で定義されている.
- 多項式を単項に展開した時に現れる項をリストにして返す. *vlist* および *order* により定まる項順序により, 順序の高いものがリストの先頭に来るようにソートされる.
- グレブナ基底はしばしば係数が巨大になるため, 実際にどの項が現れているのかを見るためなどに用いる.

```
[233] G=gr(katsura(5),[u5,u4,u3,u2,u1,u0],2)$
[234] p_terms(G[0],[u5,u4,u3,u2,u1,u0],2);
[u5,u0^31,u0^30,u0^29,u0^28,u0^27,u0^26,u0^25,u0^24,u0^23,u0^22,
u0^21,u0^20,u0^19,u0^18,u0^17,u0^16,u0^15,u0^14,u0^13,u0^12,u0^11,
u0^10,u0^9,u0^8,u0^7,u0^6,u0^5,u0^4,u0^3,u0^2,u0,1]
```

8.10.29 gb_comp

gb_comp(*plist1*, *plist2*)
:: 多項式リストが, 符号を除いて集合として等しいかどうか調べる.

return 0 または *1*

plist1 plist2

- *plist1*, *plist2* について, 符号を除いて集合として等しいかどうか調べる.

- 異なる方法で求めたグレブナ基底は, 基底の順序, 符号が異なる場合があり, それらが等しいかどうかを調べるために用いる.

```
[243] C=cyclic(6)$
[244] V=[c0,c1,c2,c3,c4,c5]$
[245] G0=gr(C,V,0)$
[246] G=tolex(G0,V,0,V)$
[247] GG=lex_tl(C,V,0,V,0)$
[248] gb_comp(G,GG);
1
```

8.10.30 katsura, hkatsura, cyclic, hcyclic

katsura(*n*)

hkatsura(*n*)

cyclic(*n*)

hcyclic(*n*)

:: 多項式リストの生成

return　リスト

n　整数

- katsura() は 'katsura', cyclic() は 'cyclic' で定義されている.
- グレブナ基底計算でしばしばテスト, ベンチマークに用いられる katsura, cyclic およびその斉次化を生成する.
- cyclic は Arnborg, Lazard, Davenport などの名で呼ばれることもある.

```
[74] load("katsura")$
[79] load("cyclic")$
[89] katsura(5);
[u0+2*u4+2*u3+2*u2+2*u1+2*u5-1,2*u4*u0-u4+2*u1*u3+u2^2+2*u5*u1,
2*u3*u0+2*u1*u4-u3+(2*u1+2*u5)*u2,2*u2*u0+2*u2*u4+(2*u1+2*u5)*u3
-u2+u1^2,2*u1*u0+(2*u3+2*u5)*u4+2*u2*u3+2*u1*u2-u1,
u0^2-u0+2*u4^2+2*u3^2+2*u2^2+2*u1^2+2*u5^2]
[90] hkatsura(5);
[-t+u0+2*u4+2*u3+2*u2+2*u1+2*u5,
-u4*t+2*u4*u0+2*u1*u3+u2^2+2*u5*u1,-u3*t+2*u3*u0+2*u1*u4+(2*u1+2*u5)*u2,
-u2*t+2*u2*u0+2*u2*u4+(2*u1+2*u5)*u3+u1^2,
-u1*t+2*u1*u0+(2*u3+2*u5)*u4+2*u2*u3+2*u1*u2,
-u0*t+u0^2+2*u4^2+2*u3^2+2*u2^2+2*u1^2+2*u5^2]
[91] cyclic(6);
[c5*c4*c3*c2*c1*c0-1,
((((c4+c5)*c3+c5*c4)*c2+c5*c4*c3)*c1+c5*c4*c3*c2)*c0+c5*c4*c3*c2*c1,
(((c3+c5)*c2+c5*c4)*c1+c5*c4*c3)*c0+c4*c3*c2*c1+c5*c4*c3*c2,
((c2+c5)*c1+c5*c4)*c0+c3*c2*c1+c4*c3*c2+c5*c4*c3,
(c1+c5)*c0+c2*c1+c3*c2+c4*c3+c5*c4,c0+c1+c2+c3+c4+c5]
[92] hcyclic(6);
[-c^6+c5*c4*c3*c2*c1*c0,
((((c4+c5)*c3+c5*c4)*c2+c5*c4*c3)*c1+c5*c4*c3*c2)*c0+c5*c4*c3*c2*c1,
(((c3+c5)*c2+c5*c4)*c1+c5*c4*c3)*c0+c4*c3*c2*c1+c5*c4*c3*c2,
```

```
((c2+c5)*c1+c5*c4)*c0+c3*c2*c1+c4*c3*c2+c5*c4*c3,
(c1+c5)*c0+c2*c1+c3*c2+c4*c3+c5*c4,c0+c1+c2+c3+c4+c5]
```

参照　8.10.12 節「dp_dtop」p.126.

8.10.31 primadec, primedec

primadec(*plist*,*vlist*)

primedec(*plist*,*vlist*)

:: イデアルの分解

return

plist　多項式リスト

vlist　変数リスト

- primadec(), primedec は 'primdec' で定義されている.
- primadec(), primedec() はそれぞれ有理数体上でのイデアルの準素分解, 根基の素イデアル分解を行う.
- 引数は多項式リストおよび変数リストである. 多項式は有理数係数のみが許される.
- primadec は [準素成分, 付属素イデアル] のリストを返す.
- primadec は 素因子のリストを返す.
- 結果において, 多項式リストとして表示されている各イデアルは全てグレブナ基底である. 対応する項順序は, それぞれ変数 PRIMAORD, PRIMEORD に格納されている.
- primadec は [Shimoyama,Yokoyama] の準素分解アルゴリズムを実装している.
- もし素因子のみを求めたいなら, primedec を使う方がよい. これは, 入力イデアルが根基イデアルでない場合に, primadec の計算に余分なコストが必要となる場合があるからである.

```
[84] load("primdec")$
[102] primedec([p*q*x-q^2*y^2+q^2*y,-p^2*x^2+p^2*x+p*q*y,
(q^3*y^4-2*q^3*y^3+q^3*y^2)*x-q^3*y^4+q^3*y^3,
-q^3*y^4+2*q^3*y^3+(-q^3+p*q^2)*y^2],[p,q,x,y]);
[[y,x],[y,p],[x,q],[q,p],[x-1,q],[y-1,p],[(y-1)*x-y,q*y^2-2*q*y-p+q]]
[103] primadec([x,z*y,w*y^2,w^2*y-z^3,y^3],[x,y,z,w]);
[[[x,z*y,y^2,w^2*y-z^3],[z,y,x]],[[w,x,z*y,z^3,y^3],[w,z,y,x]]]
```

参照　6.3.15 節「fctr sqfr」p.50, 8.5 節「項順序の設定」p.113.

8.10.32 primedec_mod

primedec_mod(*plist*,*vlist*,*ord*,*mod*,*strategy*)

:: イデアルの分解

return

plist　多項式リスト

vlist　変数リスト

ord　数, リストまたは行列

mod　正整数

strategy　整数

- `primedec_mod()` は ‘`primdec_mod`’ で定義されている．`[Yokoyama]` の素イデアル分解アルゴリズムを実装している．
- `primedec_mod()` は有限体上でのイデアルの根基の素イデアル分解を行い, 素イデアルのリストを返す．
- `primedec_mod()` は, GF(*mod*) 上での分解を与える．結果の各成分の生成元は, 整数係数多項式である．
- 結果において, 多項式リストとして表示されている各イデアルは全て [*vlist*,*ord*] で指定される項順序に関するグレブナ基底である．
- *strategy* が 0 でないとき, incremental に component の共通部分を計算することによる early termination を行う．一般に, イデアルの次元が高い場合に有効だが, 0 次元の場合など, 次元が小さい場合には overhead が大きい場合がある．
- 計算途中で内部情報を見たい場合には、前もって `dp_gr_print(2)` を実行しておけばよい．

```
[0] load("primdec_mod")$
[246] PP444=[x^8+x^2+t,y^8+y^2+t,z^8+z^2+t]$
[247] primedec_mod(PP444,[x,y,z,t],0,2,1);
[[y+z,x+z,z^8+z^2+t],[x+y,y^2+y+z^2+z+1,z^8+z^2+t],
[y+z+1,x+z+1,z^8+z^2+t],[x+z,y^2+y+z^2+z+1,z^8+z^2+t],
[y+z,x^2+x+z^2+z+1,z^8+z^2+t],[y+z+1,x^2+x+z^2+z+1,z^8+z^2+t],
[x+z+1,y^2+y+z^2+z+1,z^8+z^2+t],[y+z+1,x+z,z^8+z^2+t],
[x+y+1,y^2+y+z^2+z+1,z^8+z^2+t],[y+z,x+z+1,z^8+z^2+t]]
[248]
```

参照　6.3.17 節「`modfctr`」p.51, 8.10.6 節「`dp_gr_main dp_gr_mod_main dp_gr_f_main dp_weyl_gr_main dp_weyl_gr_mod_main dp_weyl_gr_f_main`」p.122, 8.5 節「項順序の設定」p.113. 8.10.9 節「`dp_gr_flags dp_gr_print`」p.124.

8.10.33 bfunction, bfct, generic_bfct, ann, ann0

`bfunction`(*f*)
`bfct`(*f*)
`generic_bfct`(*plist*,*vlist*,*dvlist*,*weight*)
:: *b* 関数の計算

`ann`(*f*)
`ann0`(*f*)　:: 多項式のベキの annihilator の計算

return　多項式またはリスト

f　多項式

plist　多項式リスト

vlist dvlist
変数リスト

- ‘`bfct`’ で定義されている．
- `bfunction`(*f*), `bfct`(*f*) は多項式 *f* の global *b* 関数 `b(s)` を計算する．`b(s)` は, Weyl 代数 `D` 上の一変数多項式環 `D[s]` の元 `P(x,s)` が存在して, `P(x,s)f^(s+1)=b(s)f^s` を満たすような多項式 `b(s)` の中で, 次数が最も低いものである．

- generic_bfct(*f*,*vlist*,*dvlist*,*weight*) は, *plist* で生成される D の左イデアル I の, ウェイト *weight* に関する global b 関数を計算する. *vlist* は x-変数, *vlist* は対応する D-変数を順に並べる.
- bfunction と bfct では用いているアルゴリズムが異なる. どちらが高速かは入力による.
- ann(*f*) は, *f*^s の annihilator ideal の生成系を返す. ann(*f*) は, [*a*,*list*] なるリストを返す. ここで, *a* は f の b 関数の最小整数根, *list* は ann(*f*) の結果の s$ に, *a* を代入したものである.
- 詳細については, [Saito,Sturmfels,Takayama] を見よ.

```
[0] load("bfct")$
[216] bfunction(x^3+y^3+z^3+x^2*y^2*z^2+x*y*z);
-9*s^5-63*s^4-173*s^3-233*s^2-154*s-40
[217] fctr(@);
[[-1,1],[s+2,1],[3*s+4,1],[3*s+5,1],[s+1,2]]
[218] F = [4*x^3*dt+y*z*dt+dx,x*z*dt+4*y^3*dt+dy,
x*y*dt+5*z^4*dt+dz,-x^4-z*y*x-y^4-z^5+t]$
[219] generic_bfct(F,[t,z,y,x],[dt,dz,dy,dx],[1,0,0,0]);
20000*s^10-70000*s^9+101750*s^8-79375*s^7+35768*s^6-9277*s^5
+1278*s^4-72*s^3
[220] P=x^3-y^2$
[221] ann(P);
[2*dy*x+3*dx*y^2,-3*dx*x-2*dy*y+6*s]
[222] ann0(P);
[-1,[2*dy*x+3*dx*y^2,-3*dx*x-2*dy*y-6]]
```

参照　8.9 節「Weyl 代数」p.117.

9 代数的数に関する演算

9.1 代数的数の表現

Asir においては, 代数体という対象は定義されない. 独立した対象として定義されるのは, 代数的数である. 代数体は, 有理数体に, 代数的数を有限個順次添加した体として仮想的に定義される. 新たな代数的数は, 有理数およびこれまで定義された代数的数の多項式を係数とする 1 変数多項式を定義多項式として定義される. 以下, ある定義多項式の根として定義された代数的数を, root と呼ぶことにする.

```
[0] A0=newalg(x^2+1);
(#0)
[1] A1=newalg(x^3+A0*x+A0);
(#1)
[2]  [type(A0),ntype(A0)];
[1,2]
```

この例では, A0 は x^2+1=0 の根, A1 は, その A0 を係数に含む x^3+A0*x+A0=0 の根として定義されている.

newalg() の引数すなわち定義多項式には次のような制限がある.

1. 定義多項式は 1 変数多項式でなければならない.
2. newalg() の引数である定義多項式は, 代数的数を含む式の簡単化のために用いられる. この簡単化は, 組み込み函数 srem() に相当する内部ルーチンを用いて行われる. このため, 定義多項式の主係数は, 有理数になっている必要がある.
3. 定義多項式の係数は すでに定義されている root の有理数係数多項式でなければならない.
4. 定義多項式は, その係数に含まれる全ての root を有理数に添加した体上で既約でなければならない.

newalg() が行う引数チェックは, 1 および 2 のみである. 特に, 引数の定義多項式の既約性は全くチェックされない. これは既約性のチェックが多大な計算量を必要とするためで, この点に関しては, ユーザの責任に任されている.

一旦 newalg() によって定義された代数的数は, 数としての識別子を持ち, また, 数の中では代数的数としての識別子を持つ. (type(), vtype() 参照.) さらに, 有理数と, root の有理式も同様に代数的数となる.

```
[87] N=(A0^2+A1)/(A1^2-A0-1);
((#1+#0^2)/(#1^2-#0-1))
[88] [type(N),ntype(N)];
[1,2]
```

例からわかるように, rootは #n と表示される. しかし, ユーザはこの形では入力できない. root は変数に格納して用いるか, あるいは alg(n) により取り出す. また, 効率は落ちるが, 全く同じ引数 (変数は異なっていてもよい) により newalg() を呼べば, 新しい代数的数は定義されずに既に定義されたものが得られる.

```
[90] alg(0);
(#0)
[91] newalg(t^2+1);
(#0)
```

root の定義多項式は, defpoly() により取り出せる.

```
[96] defpoly(A0);
t#0^2+1
[97] defpoly(A1);
t#1^3+t#0*t#1+t#0
```

ここで現れた, t#0, t#1 はそれぞれ #0, #1 に対応する不定元である. これらもユーザが入力することはできない. var() で取り出すか, あるいは algv(n) により取り出す.

```
[98] var(@);
t#1
[99] algv(0);
t#0
[100]
```

9.2 代数的数の演算

前節で, 代数的数の表現, 定義について述べた. ここでは, 代数的数を用いた演算について述べる. 代数的数に関しては, 組み込み函数として提供されている機能はごく少数で, 大部分はユーザ定義函数により実現されている. ファイルは, 'sp' で, 'gr' と同様 **Asir** の標準ライブラリディレクトリにおかれている.

```
[0] load("gr")$
[1] load("sp")$
```

あるいは, 常に用いるならば, '$HOME/.asirrc' に書いておくのもよい.

root は その他の数と同様, 四則演算が可能となる. しかし, 定義多項式による簡単化は自動的には行われないので, ユーザの判断で適宜行わなければならない. 特に, 分母が 0 になる場合に致命的なエラーとなるため, 実際に分母を持つ代数的数を生成する場合には細心の注意が必要となる.

代数的数の, 定義多項式による簡単化は, simpalg() で行う.

```
[49] T=A0^2+1;
(#0^2+1)
[50] simpalg(T);
0
```

simpalg() は有理式の形をした代数的数を, 多項式の形に簡単化する.

```
[39] A0=newalg(x^2+1);
(#0)
[40] T=(A0^2+A0+1)/(A0+3);
((#0^2+#0+1)/(#0+3))
[41] simpalg(T);
(3/10*#0+1/10)
[42] T=1/(A0^2+1);
((1)/(#0^2+1))
[43] simpalg(T);
div : division by 0
stopped in invalgp at line 258 in file "/usr/local/lib/asir/sp"
258                     return 1/A;
(debug)
```

この例では, 分母が 0 の代数的数を簡単化しようとして 0 による除算が生じたため, ユーザ定義函数である simpalg() の中でデバッガが呼ばれたことを示す. simpalg() は, 代数的数を係数とする多項式の各係数を簡単化できる.

```
[43] simpalg(1/A0*x+1/(A0+1));
(-#0)*x+(-1/2*#0+1/2)
```

代数的数を係数とする多項式の基本演算は, 適宜 simpalg() を呼ぶことを除けば通常の場合と同様であるが, 因数分解などで頻繁に用いられるノルムの計算などにおいては, root を不定元に置き換える必要が出てくる. この場合, algptorat() を用いる.

```
[83] A0=newalg(x^2+1);
(#0)
[84] A1=newalg(x^3+A0*x+A0);
(#1)
[85] T=(2*A0+A1*A0+A1^2)*x+(1+A1)/(2+A0);
(#1^2+#0*#1+2*#0)*x+((#1+1)/(#0+2))
[86] S=algptorat(T);
(((t#0+2)*t#1^2+(t#0^2+2*t#0)*t#1+2*t#0^2+4*t#0)*x+t#1+1)/(t#0+2)
[87] algptorat(coef(T,1));
t#1^2+t#0*t#1+2*t#0
```

このように, algptorat() は, 多項式, 数に含まれる root を, 対応する不定元, すなわち #n に対する t#n に置き換える. 既に述べたように, この不定元はユーザが入力することはできない. これは, ユーザの入力した不定元と, root に対応する不定元が一致しないようにするためである.

逆に, root に対応する不定元を, 対応する root に置き換えるためには rattoalgp() を用いる.

```
[88] rattoalgp(S,[alg(0)]);
(((#0+2)/(#0+2))*t#1^2+((#0^2+2*#0)/(#0+2))*t#1
+((2*#0^2+4*#0)/(#0+2)))*x+((1)/(#0+2))*t#1+((1)/(#0+2))
[89] rattoalgp(S,[alg(0),alg(1)]);
(((#0^3+6*#0^2+12*#0+8)*#1^2+(#0^4+6*#0^3+12*#0^2+8*#0)*#1
+2*#0^4+12*#0^3+24*#0^2+16*#0)/(#0^3+6*#0^2+12*#0+8))*x
+(((#0+2)*#1+#0+2)/(#0^2+4*#0+4))
[90] rattoalgp(S,[alg(1),alg(0)]);
(((#0+2)*#1^2+(#0^2+2*#0)*#1+2*#0^2+4*#0)/(#0+2))*x
+((#1+1)/(#0+2))
[91] simpalg(@89);
(#1^2+#0*#1+2*#0)*x+((-1/5*#0+2/5)*#1-1/5*#0+2/5)
[92] simpalg(@90);
(#1^2+#0*#1+2*#0)*x+((-1/5*#0+2/5)*#1-1/5*#0+2/5)
```

rattoalgp() は, 置換の対象となる root のリストを第 2 引数にとり, 左から順に, 対応する不定元を置き換えて行く. この例は, 置換する順序を換えると簡単化を行わないことにより結果が一見異なるが, 簡単化により実は一致することを示している. algptorat(), rattoalgp() は, ユーザが独自の簡単化を行いたい場合などにも用いることができる.

9.3 代数体上での 1 変数多項式の演算

'sp' では, 1 変数多項式に限り, GCD, 因数分解およびそれらの応用として最小分解体を求める函数を提供している.

9.3.1 GCD

代数体上での GCD は cr_gcda() により計算される. この函数はモジュラ演算および中国剰余定理により代数体上の GCD を計算するもので, 逐次拡大に対しても有効である.

```
[63] A=newalg(t^9-15*t^6-87*t^3-125);
(#0)
[64] B=newalg(75*s^2+(10*A^7-175*A^4-470*A)*s+3*A^8-45*A^5-261*A^2);
(#1)
[65] P1=75*x^2+(150*B+10*A^7-175*A^4-395*A)*x
+(75*B^2+(10*A^7-175*A^4-395*A)*B+13*A^8-220*A^5-581*A^2)$
[66] P2=x^2+A*x+A^2$
[67] cr_gcda(P1,P2);
27*x+((#0^6-19*#0^3-65)*#1-#0^7+19*#0^4+38*#0)
```

9.3.2 無平方分解, 因数分解

無平方分解は, 多項式とその微分との GCD の計算から始まるもっとも一般的なアルゴリズムを採用している. 函数は asq() である.

```
[116] A=newalg(x^2+x+1);
(#4)
[117] T=simpalg((x+A+1)*(x^2-2*A-3)^2*(x^3-x-A)^2);
x^11+(#4+1)*x^10+(-4*#4-8)*x^9+(-10*#4-4)*x^8+(16*#4+20)*x^7
+(24*#4-6)*x^6+(-29*#4-31)*x^5+(-15*#4+28)*x^4+(38*#4+29)*x^3
+(#4-23)*x^2+(-21*#4-7)*x+(3*#4+8)
[118] asq(T);
[[x^5+(-2*#4-4)*x^3+(-#4)*x^2+(2*#4+3)*x+(#4-2),2],[x+(#4+1),1]]
```

結果は通常と同様に, [**因子**, **重複度**] のリストとなるが, 全ての因子の積は, もとの多項式と定数倍の差はあり得る. これは, 因子を整数係数にして見やすくするためで, 因数分解でも同様である.

代数体上での因数分解は, Trager によるノルム法を改良したもので, 特にある多項式に対し, その根を添加した体上でその多項式自身を因数分解する場合に特に有効である.

```
[119] af(T,[A]);
[[x^3-x+(-#4),2],[x^2+(-2*#4-3),2],[x+(#4+1),1]]
```

引数は 2 つで, 第 2 引数は, root のリストである. 因数分解は有理数体に, それらの root を添加した体上で行われる. root の順序には制限がある. すなわち, 後で定義されたものほど前の方にこなければならない. 並べ換えは, 自動的には行われない. ユーザの責任となる.

ノルムを用いた因数分解においては, ノルムの計算と整数係数 1 変数多項式の因数分解の効率が, 全体の効率を左右する. このうち, 特に高次の多項式の場合に後者において組合せ爆発により計算不能になる場合がしばしば生ずる.

```
[120] B=newalg(x^2-2*A-3);
(#5)
[121] af(T,[B,A]);
[[x+(#5),2],[x^3-x+(-#4),2],[x+(-#5),2],[x+(#4+1),1]]
```

9.3.3 最小分解体

やや特殊な演算ではあるが, 前節の因数分解を反復適用することにより, 多項式の最小分解体を求めることができる. 函数は sp() である.

```
[103] sp(x^5-2);
[[x+(-#1),2*x+(#0^3*#1^3+#0^4*#1^2+2*#1+2*#0),2*x+(-#0^4*#1^2),
2*x+(-#0^3*#1^3),x+(-#0)],
[[(#1),t#1^4+t#0*t#1^3+t#0^2*t#1^2+t#0^3*t#1+t#0^4],[(#0),t#0^5-2]]]
```

sp() は 1 引数で, 結果は [1 次因子のリスト, [[root, algptorat(定義多項式)] のリスト] なるリストである. 第 2 要素の [root,algptorat(定義多項式)] のリストは, 右から順に, 最小分解体が得られるまで添加していった root を示す. その定義多項式は, その直前までの root を添加した体上で既約であることが保証されている.

結果の第 1 要素である 1 次因子のリストは, 第 2 要素の root を全て添加した体上での, sp() の引数の多項式の全ての因子を表す. その体は最小分解体となっているので, 因子は全て 1 次となるわけである. af() と同様, 全ての因子の積は, もとの多項式と定数倍の差はあり得る.

9.4 代数的数に関する函数のまとめ

9.4.1 newalg

newalg(*defpoly*)
:: root を生成する.

return　代数的数 (root)

defpoly　多項式

- *defpoly* を定義多項式とする代数的数 (root) を生成する.
- *defpoly* に対する制限に関しては, 9.1 節「代数的数の表現」p.140 を参照.

```
[0] A0=newalg(x^2-2);
(#0)
```

参照　9.4.2 節「defpoly」p.144

9.4.2 defpoly

defpoly(*alg*)
:: root の定義多項式を返す.

return　多項式

alg　代数的数 (root)

- root *alg* の定義多項式を返す.
- root を #*n* とすれば, 定義多項式の主変数は t#*n* となる.

```
[1] defpoly(A0);
t#0^2-2
```

参照　9.4.1 節「newalg」p.144, 9.4.3 節「alg」p.145, 9.4.4 節「algv」p.145

9.4.3 alg

alg(*i*)　:: インデックスに対応する root を返す.

return　代数的数 (root)

i　整数

- root #*i* を返す.
- #*i* はユーザが直接入力できないため, alg(*i*) という形で入力する.

```
[2] x+#0;
syntax error
0
[3] alg(0);
(#0)
```

参照　9.4.1 節「newalg」p.144, 9.4.4 節「algv」p.145

9.4.4 algv

algv(*i*)　:: alg(*i*) に対応する不定元を返す.

return　多項式

i　整数

- 多項式 t#*i* を返す.
- t#*i* はユーザが直接入力できないため, algv(*i*) という形で入力する.

```
[4] var(defpoly(A0));
t#0
[5] t#0;
syntax error
0
[6] algv(0);
t#0
```

参照　9.4.1 節「newalg」p.144, 9.4.2 節「defpoly」p.144, 9.4.3 節「alg」p.145

9.4.5 simpalg

simpalg(*rat*)
:: 有理式に含まれる代数的数を簡単化する.

return　有理式

rat　有理式

- 'sp' で定義されている.
- 数, 多項式, 有理式に含まれる代数的数を, 含まれる root の定義多項式により簡単化する.
- 数の場合, 分母があれば有理化され, 結果は root の多項式となる.
- 多項式の場合, 各係数が簡単化される.
- 有理式の場合, 分母分子が多項式として簡単化される.

```
[7] simpalg((1+A0)/(1-A0));
simpalg undefined
return to toplevel
[7] load("sp")$
[46] simpalg((1+A0)/(1-A0));
(-2*#0-3)
[47] simpalg((2-A0)/(2+A0)*x^2-1/(3+A0));
(-2*#0+3)*x^2+(1/7*#0-3/7)
[48] simpalg((x+1/(A0-1))/(x-1/(A0+1)));
(x+(#0+1))/(x+(-#0+1))
```

9.4.6 algptorat

algptorat(*poly*)
:: 多項式に含まれる root を, 対応する不定元に置き換える.

return　多項式

poly　多項式

- 'sp' で定義されている.
- 多項式に含まれる root #n を全て t#n に置き換える.

```
[49] algptorat((-2*alg(0)+3)*x^2+(1/7*alg(0)-3/7));
(-2*t#0+3)*x^2+1/7*t#0-3/7
```

参照　9.4.2 節「defpoly」p.144, 9.4.4 節「algv」p.145

9.4.7 rattoalgp

rattoalgp(*poly*,*alglist*)
:: 多項式に含まれる root に対応する不定元を root に置き換える.

return　多項式

poly　多項式

alglist　リスト

- 'sp' で定義されている.
- 第 2 引数は root のリストである. rattoalgp() は, この root に対応する不定元を, それぞれ root に置き換える.

```
[51] rattoalgp((-2*algv(0)+3)*x^2+(1/7*algv(0)-3/7),[alg(0)]);
(-2*#0+3)*x^2+(1/7*#0-3/7)
```

参照　9.4.3 節「alg」p.145, 9.4.4 節「algv」p.145

9.4.8 cr_gcda

cr_gcda(*poly1*,*poly2*)
:: 代数体上の 1 変数多項式の GCD

return　多項式

poly1 poly2
　　多項式

- ‘sp’ で定義されている.
- 2 つの 1 変数多項式の GCD を求める.

```
[76] X=x^6+3*x^5+6*x^4+x^3-3*x^2+12*x+16$
[77] Y=x^6+6*x^5+24*x^4+8*x^3-48*x^2+384*x+1024$
[78] A=newalg(X);
(#0)
[79] cr_gcda(X,subst(Y,x,x+A));
x+(-#0)
```

参照　8.10.1 節「gr hgr gr_mod」p.117, 9.4.10 節「asq af af_noalg」p.147

9.4.9 sp_norm

sp_norm(*alg*,*var*,*poly*,*alglist*)
　　:: 代数体上でのノルムの計算

return　多項式

var　*poly* の主変数

poly　1 変数多項式

alg　root

alglist　root のリスト

- ‘sp’ で定義されている.
- *poly* の, *alg* に関するノルムをとる. すなわち, $\mathbf{K} = \mathbf{Q}(alglist \setminus \{alg\})$ とするとき, *poly* に現れる *alg* を, *alg* の $\mathbf{K}$ 上の共役に置き換えたもの全ての積を返す.
- 結果は $\mathbf{K}$ 上の多項式となる.
- 実際には入力により場合わけが行われ, 終結式の直接計算や中国剰余定理により計算されるが, 最適な選択が行われているとは限らない. 大域変数 USE_RES を 1 に設定することにより, 常に終結式により計算させることができる.

```
[0] load("sp")$
[39] A0=newalg(x^2+1)$
[40] A1=newalg(x^2+A0)$
[41] sp_norm(A1,x,x^3+A0*x+A1,[A1,A0]);
x^6+(2*#0)*x^4+(#0^2)*x^2+(#0)
[42] sp_norm(A0,x,@@,[A0]);
x^12+2*x^8+5*x^4+1
```

参照　6.3.14 節「res」p.49, 9.4.10 節「asq af af_noalg」p.147

9.4.10 asq, af, af_noalg

asq(*poly*)　:: 代数体上の 1 変数多項式の無平方分解

af(*poly*,*alglist*)
af_noalg(*poly*,*defpolylist*)
　　:: 代数体上の 1 変数多項式の因数分解

return　リスト

poly　多項式

alglist　root のリスト

defpolylist　root を表す不定元と定義多項式のペアのリスト

- いずれも 'sp' で定義されている.
- root を含まない場合は整数上の函数が呼び出され高速であるが, root を含む場合には, cr_gcda() が起動されるためしばしば時間がかかる.
- af() は, 基礎体の指定, すなわち第 2 引数の, root のリストの指定が必要である.
- alglist で指定される root は, 後で定義されたものほど前の方に来なければならない.
- af(F,AL) において, AL は代数的数のリストであり, 有理数体の代数拡大を表す. AL=[An,...,A1] と書くとき, 各 Ak は, それより右にある代数的数を係数とした, モニックな定義多項式で定義されていなければならない.

```
[1] A1 = newalg(x^2+1);
[2] A2 = newalg(x^2+A1);
[3] A3 = newalg(x^2+A2*x+A1);
[4] af(x^2+A2*x+A1,[A2,A1]);
[[x^2+(#1)*x+(#0),1]]
```

  af_noalg では, *poly* に含まれる代数的数 *ai* を不定元 *vi* で置き換える. defpolylist は, [[vn,dn(vn,...,v1)],...,[v1,d(v1)]] なるリストである. ここで *di*(vi,...,v1) は *ai* の定義多項式において代数的数を全て *vj* に置き換えたものである.

```
[1] af_noalg(x^2+a2*x+a1,[[a2,a2^2+a1],[a1,a1^2+1]]);
[[x^2+a2*x+a1,1]]
```

- 結果は, 通常の無平方分解, 因数分解と同様 [**因子**, **重複度**] のリストである. af_noalg の場合, **因子** に現れる代数的数は, *defpolylist* に従って不定元に置き換えられる.
- 重複度を込めた因子の全ての積は, *poly* と定数倍の違いがあり得る.

```
[98] A = newalg(t^2-2);
(#0)
[99] asq(-x^4+6*x^3+(2*alg(0)-9)*x^2+(-6*alg(0))*x-2);
[[-x^2+3*x+(#0),2]]
[100] af(-x^2+3*x+alg(0),[alg(0)]);
[[x+(#0-1),1],[-x+(#0+2),1]]
[101] af_noalg(-x^2+3*x+a,[[a,x^2-2]]);
[[x+a-1,1],[-x+a+2,1]]
```

参照　9.4.8 節「cr_gcda」p.146, 6.3.15 節「fctr sqfr」p.50

9.4.11 sp, sp_noalg

sp(*poly*)
sp_noalg(*poly*)
:: 最小分解体を求める.

return　リスト

poly　多項式

- ‘sp’ で定義されている.
- 有理数係数の 1 変数多項式 *poly* の最小分解体, およびその体上での *poly* の 1 次因子への分解を求める.
- 結果は, *poly* の因子のリストと, 最小分解体の, 逐次拡大による表現からなるリストである. sp_noalg では, 全ての代数的数が, 対応する不定元 (即ち #i に対する t#i) に置き換えられる. これにより, sp_noalg の出力は, 整数係数多変数多項式のリストとなる.
- 最小分解体は, [root,algptorat(defpoly(root))] のリストとして表現されている. すなわち, 求める最小分解体は, 有理数体に, この root を全て添加した体として得られる. 添加は, 右の方の root から順に行われる.
- sp() は, 内部でノルムの計算のために sp_norm() をしばしば起動する. ノルムの計算は, 状況に応じてさまざまな方法で行われるが, そこで用いられる方法が最善とは限らず, 単純な終結式の計算の方が高速である場合もある. 大域変数 USE_RES を 1 に設定することにより, 常に終結式により計算させることができる.

```
[101] L=sp(x^9-54);
[[x+(-#2),-54*x+(#1^6*#2^4),54*x+(#1^6*#2^4+54*#2),
54*x+(-#1^8*#2^2),-54*x+(#1^5*#2^5),54*x+(#1^5*#2^5+#1^8*#2^2),
-54*x+(-#1^7*#2^3-54*#1),54*x+(-#1^7*#2^3),x+(-#1)],
[[(#2),t#2^6+t#1^3*t#2^3+t#1^6],[(#1),t#1^9-54]]]
[102] for(I=0,M=1;I<9;I++)M*=L[0][I];
[111] M=simpalg(M);
-1338925209984*x^9+72301961339136
[112] ptozp(M);
-x^9+54
```

参照　9.4.10 節「asq af af_noalg」p.147, 9.4.2 節「defpoly」p.144, 9.4.6 節「algptorat」p.146, 9.4.9 節「sp_norm」p.147.

10 有限体に関する演算

10.1 有限体の表現および演算

Asir においては, 有限体は, 正標数素体 GF(p), 標数 2 の有限体 GF(2^n), GF(p) の n 次拡大 GF(p^n) が定義できる. これらは全て, `setmod_ff()` により定義される.

```
[0] P=pari(nextprime,2^50);
1125899906842679
[1] setmod_ff(P);
1125899906842679
[2] field_type_ff();
1
[3] load("fff");
1
[4] F=defpoly_mod2(50);
x^50+x^4+x^3+x^2+1
[5] setmod_ff(F);
x^50+x^4+x^3+x^2+1
[6] field_type_ff();
2
[7] setmod_ff(x^3+x+1,1125899906842679);
[1*x^3+1*x+1,1125899906842679]
[8] field_type_ff();
3
[9] setmod_ff(3,5);
[3,x^5+2*x+1,x]
[10] field_type_ff();
4
```

`setmod_ff()` は, さまざまなタイプの有限体を基礎体としてセットする. 引数が正整数 p の場合 GF(p), n 次多項式 f(x) の場合, f(x) mod 2 を定義多項式とする GF(2^n) をそれぞれ基礎体としてセットする. また, 有限素体の有限次拡大も定義できる. 詳しくは 3.2 節「数の型」p.12 を参照. `setmod_ff()` においては引数の既約チェックは行わず, 呼び出し側が責任を持つ.

基礎体とは, あくまで有限体の元として宣言あるいは定義されたオブジェクトが, セットされた基礎体の演算に従うという意味である. 即ち, 有理数どうしの演算の結果は有理数となる. 但し, 四則演算において一方のオペランドが有限体の元の場合には, 他の元も自動的に同じ有限体の元と見なされ, 演算結果も同様になる.

0 でない有限体の元は, 数オブジェクトであり, 識別子の値は 1 である. さらに, 0 でない有限体の元の数識別子は, GF(p) の場合 6, GF(2^n) の場合 7 となる.

有限体の元の入力方法は, 有限体の種類により様々である. GF(p) の場合, `simp_ff()` による.

```
[0] P=pari(nextprime,2^50);
1125899906842679
[1] setmod_ff(P);
1125899906842679
[2] A=simp_ff(2^100);
```

```
3025
[3] ntype(@@);
6
```

また, GF(2^n) の場合いくつかの方法がある.

```
[0] setmod_ff(x^50+x^4+x^3+x^2+1);
x^50+x^4+x^3+x^2+1
[1] A=@;
(@)
[2] ptogf2n(x^50+1);
(@^50+1)
[3] simp_ff(@@);
(@^4+@^3+@^2)
[4] ntogf2n(2^10-1);
(@^9+@^8+@^7+@^6+@^5+@^4+@^3+@^2+@+1)
```

有限体の元は数であり, 体演算が可能である. `@` は GF(2^n) の, GF(2) 上の生成元である. 詳しくは 3.2 節「数の型」p.12 を参照.

10.2 有限体上での 1 変数多項式の演算

'`fff`' では, 有限体上の 1 変数多項式に対し, 無平方分解, DDF, 因数分解, 多項式の既約判定などの関数が定義されている.

いずれも, 結果は [**因子**, **重複度**] のリストとなるが, 因子は monic となり, 入力多項式の主係数は捨てられる.

無平方分解は, 多項式とその微分との GCD の計算から始まるもっとも一般的なアルゴリズムを採用している.

有限体上での因数分解は, DDF の後, 次数別因子の分解の際に, Berlekamp アルゴリズムで零空間を求め, 基底ベクトルの最小多項式を求め, その根を Cantor-Zassenhaus アルゴリズムにより求める, という方法を実装している.

10.3 小標数有限体上での多項式の演算

小標数有限体係数の多項式に限り, 多変数多項式の因数分解が組み込み関数として実装されている. 関数は `sffctr()` である. また, `modfctr()` も, 有限素体上で多変数多項式の因数分解を行うが, 実際には, 内部で十分大きな拡大体を設定し, `sffctr()` を呼び出して, 最終的に素体上の因子を構成する, という方法で計算している.

10.4 有限体上の楕円曲線に関する演算

有限体上の楕円曲線に関するいくつかの基本的な演算が, 組み込み関数として提供されている.

楕円曲線の指定は, 長さ 2 のベクトル [a b] で行う. a, b は有限体の元で, `setmod_ff` で定義されている有限体が素体の場合, y^2=x^3+ax+b, 標数 2 の体の場合 y^2+xy=x^3+ax^2+b を表す.

楕円曲線上の点は, 無限遠点も込めて加法群をなす. この演算に関して, 加算 (`ecm_add_ff()`), 減算 (`ecm_sub_ff()`) および逆元計算のための関数 (`ecm_chsgn_ff()`) が提供されている. 注意すべきは, 演算の対象となる点の表現が,

- 無限遠点は 0.
- それ以外の点は, 長さ 3 のベクトル [*x y z*]. ただし, *z* は 0 でない.

という点である. [*x y z*] は斉次座標による表現であり, アフィン座標では [*x/z y/z*] なる点を表す. よって, アフィン座標 [*x y*] で表現された点を演算対象とするには, [*x y 1*] なるベクトルを生成する必要がある. 演算結果も斉次座標で得られるが, *z* 座標が 1 とは限らないため, アフィン座標を求めるためには *x*, *y* 座標を *z* 座標で割る必要がある.

10.5 有限体に関する函数のまとめ

10.5.1 setmod_ff

`setmod_ff([`*prime*`|`*poly*`])`
`setmod_ff(`*prime*`,`*n*`])`
:: 有限体の設定, 設定されている有限体の法, 定義多項式の表示

return　数または多項式

prime　素数

poly　GF(2) 上既約な 1 変数多項式

n　拡大次数

- 引数が正整数 *prime* の時, GF(*prime*) を基礎体として設定する.
- 引数が多項式 *poly* の時, GF(2^deg(*poly* mod 2)) = GF(2)[t]/(*poly*(t) mod 2) を基礎体として設定する.
- 引数が *p* と *n* の時, GF(*p*^*n*) を基礎体として設定する. *p*^*n* は *2^29* 未満でなければならない. また, *p* が *2^14* 以上のとき, *n* は 1 でなければならない.
- 無引数の時, 設定されている基礎体が GF(*prime*) の場合 *prime*, GF(2^*n*) の場合定義多項式を返す. 基礎体が GF(p^*n*) (*p*^*n* が *2^14* 未満) の場合, [*p*,*defpoly*,*prim_elem*] を返す. ここで, *defpoly* は, *n* 次拡大の定義多項式, *prim_elem* は, GF(*p*^n) 乗法群の生成元を意味する.
- GF(2^*n*) の定義多項式は, GF(2) 上 n 次既約ならなんでも良いが, 効率に影響するため, `defpoly_mod2()` で生成するのがよい.

```
[174] defpoly_mod2(100);
x^100+x^15+1
[175] setmod_ff(@@);
x^100+x^15+1
[176] setmod_ff();
x^100+x^15+1
[177] setmod_ff(2,5);
[2,x^5+x^2+1,x]
```

参照　10.5.14 節「defpoly_mod2」p.158

10.5.2 field_type_ff

`field_type_ff()`
:: 設定されている基礎体の種類

return　整数

- 設定されている基礎体の種類を返す.
- 設定なしなら 0, GF(p) なら 1, GF(2^n) なら 2 を返す.

```
[0] field_type_ff();
0
[1] setmod_ff(3);
3
[2] field_type_ff();
1
[3] setmod_ff(x^2+x+1);
x^2+x+1
[4] field_type_ff();
2
```

参照　10.5.1 節「setmod_ff」p.152

10.5.3 field_order_ff

`field_order_ff()`
:: 設定されている基礎体の位数

return　整数

- 設定されている基礎体の位数 (元の個数) を返す.
- 設定されている体が GF(q) ならば q を返す.

```
[0] field_order_ff();
field_order_ff : current_ff is not set
return to toplevel
[0] setmod_ff(3);
3
[1] field_order_ff();
3
[2] setmod_ff(x^2+x+1);
x^2+x+1
[3] field_order_ff();
4
```

参照　10.5.1 節「setmod_ff」p.152

10.5.4 characteristic_ff

`characteristic_ff()`
:: 設定されている体の標数

return　整数

- 設定されている体の標数を返す.
- GF(p) の場合 p, GF(2^n) の場合 2 を返す.

```
[0] characteristic_ff();
characteristic_ff : current_ff is not set
return to toplevel
[0] setmod_ff(3);
3
[1] characteristic_ff();
3
[2] setmod_ff(x^2+x+1);
x^2+x+1
[3] characteristic_ff();
2
```

参照　10.5.1 節「setmod_ff」p.152

10.5.5 extdeg_ff

extdeg_ff()
:: 設定されている基礎体の, 素体に対する拡大次数

return　整数

- 設定されている基礎体の, 素体に対する拡大次数を返す.
- GF(p) の場合 1, GF(2^n) の場合 n を返す.

```
[0] extdeg_ff();
extdeg_ff : current_ff is not set
return to toplevel
[0] setmod_ff(3);
3
[1] extdeg_ff();
1
[2] setmod_ff(x^2+x+1);
x^2+x+1
[3] extdeg_ff();
2
```

参照　10.5.1 節「setmod_ff」p.152

10.5.6 simp_ff

simp_ff(*obj*)
:: 数, あるいは多項式の係数を有限体の元に変換

return　数または多項式

obj　数または多項式

- 数, あるいは多項式の係数を有限体の元に変換する.
- 整数, あるいは整数係数多項式を, 有限体, あるいは有限体係数に変換するために用いる.
- 有限体の元に対し, 法あるいは定義多項式による reduction を行う場合にも用いる.

- 小標数有限体の元に変換する場合, 一旦素体上に射影してから, 拡大体の元に変換される. 拡大体の元に直接変換するには ptosfp() を用いる.

```
[0] simp_ff((x+1)^10);
x^10+10*x^9+45*x^8+120*x^7+210*x^6+252*x^5+210*x^4+120*x^3+45*x^2+10*x+1
[1] setmod_ff(3);
3
[2] simp_ff((x+1)^10);
1*x^10+1*x^9+1*x+1
[3] ntype(coef(@@,10));
6
[4] setmod_ff(2,3);
[2,x^3+x+1,x]
[5] simp_ff(1);
@_0
[6] simp_ff(2);
0
[7] ptosfp(2);
@_1
```

参照　10.5.1 節「setmod_ff」p.152, 10.5.8 節「lmptop」p.155, 10.5.10 節「gf2nton」p.156, 10.5.13 節「ptosfp sfptop」p.158

10.5.7 random_ff

random_ff()
:: 有限体の元の乱数生成

return　有限体の元

- 有限体の元を乱数生成する.
- random(), lrandom() と同じ 32bit 乱数発生器を使用している.

```
[0] random_ff();
random_ff : current_ff is not set
return to toplevel
[0] setmod_ff(pari(nextprime,2^40));
1099511627791
[1] random_ff();
561856154357
[2] random_ff();
45141628299
```

参照　10.5.1 節「setmod_ff」p.152, 6.1.8 節「random」p.36, 6.1.9 節「lrandom」p.36

10.5.8 lmptop

lmptop(*obj*)
:: GF(p) 係数多項式の係数を整数に変換

return　整数係数多項式

obj　GF(p) 係数多項式

- GF(p) 係数多項式の係数を整数に変換する.
- GF(p) の元は, 0 以上 p 未満の整数で表現されている. 多項式の各係数は, その値を整数オブジェクト (数識別子 0) としたものに変換される.

```
[0] setmod_ff(pari(nextprime,2^40));
1099511627791
[1] F=simp_ff((x-1)^10);
1*x^10+1099511627781*x^9+45*x^8+1099511627671*x^7+210*x^6
+1099511627539*x^5+210*x^4+1099511627671*x^3+45*x^2+1099511627781*x+1
[2] setmod_ff(547);
547
[3] F=simp_ff((x-1)^10);
1*x^10+537*x^9+45*x^8+427*x^7+210*x^6+295*x^5+210*x^4+427*x^3
+45*x^2+537*x+1
[4] lmptop(F);
x^10+537*x^9+45*x^8+427*x^7+210*x^6+295*x^5+210*x^4+427*x^3
+45*x^2+537*x+1
[5] lmptop(coef(F,1));
537
[6] ntype(@@);
0
```

参照　10.5.6 節「simp_ff」p.154

10.5.9 ntogf2n

ntogf2n(m)
:: 自然数を GF(2^n) の元に変換

return　GF(2^n) の元

m　非負整数

- 自然数 m の 2 進表現 m=*m0*+*m1**2+...+*mk**2^k に対し, GF(2^n)=GF(2)[t]/(g(t)) の元 *m0*+*m1**t+...+*mk**t^k mod g(t) を返す.
- 定義多項式による剰余は自動的には計算されないため, simp_ff() を適用する必要がある.

```
[1] setmod_ff(x^30+x+1);
x^30+x+1
[2] N=ntogf2n(2^100);
(@^100)
[3] simp_ff(N);
(@^13+@^12+@^11+@^10)
```

参照　10.5.10 節「gf2nton」p.156

10.5.10 gf2nton

gf2nton(m)
:: GF(2^n) の元を自然数に変換

return　非負整数

m　　GF(2^*n*) の元

- gf2nton の逆変換である.

```
[1] setmod_ff(x^30+x+1);
x^30+x+1
[2] N=gf2nton(2^100);
(@^100)
[3] simp_ff(N);
(@^13+@^12+@^11+@^10)
[4] gf2nton(N);
1267650600228229401496703205376
[5] gf2nton(simp_ff(N));
15360
```

参照　　10.5.10 節「gf2nton」p.156

10.5.11 ptogf2n

ptogf2n(*poly*)
:: 一変数多項式を GF(2^*n*) の元に変換

return　　GF(2^*n*) の元

poly　　一変数多項式

- *poly* の表す GF(2^*n*) の元を生成する. 係数は, 2 で割った余りに変換される. *poly* の変数に @ を代入した結果と等しい.

```
[1] setmod_ff(x^30+x+1);
x^30+x+1
[2] ptogf2n(x^100);
(@^100)
```

参照　　10.5.12 節「gf2ntop」p.157

10.5.12 gf2ntop

gf2ntop(*m*[,*v*])
:: GF(2^*n*) の元を多項式に変換

return　　一変数多項式

m　　GF(2^*n*) の元

v　　不定元

- *m* を表す多項式を, 整数係数の多項式オブジェクトとして返す.
- *v* の指定がない場合, 直前の ptogf2n() 呼び出しにおける引数の変数 (デフォルトは x), 指定がある場合には指定された不定元を変数とする多項式を返す.

```
[1] setmod_ff(x^30+x+1);
x^30+x+1
[2] N=simp_ff(gf2ntop(2^100));
(@^13+@^12+@^11+@^10)
[5] gf2ntop(N);
```

```
[207] gf2ntop(N);
x^13+x^12+x^11+x^10
[208] gf2ntop(N,t);
t^13+t^12+t^11+t^10
```

参照　10.5.11 節「ptogf2n」p.157

10.5.13 ptosfp, sfptop

ptosfp(p)
sfptop(p)
:: 小標数有限体への変換, 逆変換

return　多項式

p　多項式

- ptosfp() は, 多項式の係数を, 現在設定されている小標数有限体 GF(p^n) の元に直接変換する. 係数が既に有限体の元の場合は変化しない. 正整数の場合, まず位数で剰余を計算したあと, 標数 p により p 進展開し, p を x に置き換えた多項式を, 原始元表現に変換する. 例えば, GF(3^5) は GF(3)[x]/(x^5+2*x+1) として表現され, その各元は原始元 x に関するべき指数 k により @_k として表示される. このとき, 例えば *23 = 2*3^2+3+2* は, 2*x^2+x+2 と表現され, これは結局 x^17 と法 x^5+2*x+1 で等しいので, @_*17* と変換される.
- sfptop() は ptosfp() の逆変換である.

```
[196] setmod_ff(3,5);
[3,x^5+2*x+1,x]
[197] A = ptosfp(23);
@_17
[198] 9*2+3+2;
23
[199] x^17-(2*x^2+x+2);
x^17-2*x^2-x-2
[200] sremm(@,x^5+2*x+1,3);
0
[201] sfptop(A);
23
```

参照　10.5.1 節「setmod_ff」p.152, 10.5.6 節「simp_ff」p.154

10.5.14 defpoly_mod2

defpoly_mod2(d)
:: GF(2) 上既約な一変数多項式の生成

return　多項式

d　正整数

- 'fff' で定義されている.
- 与えられた次数 d に対し, GF(2) 上 d 次の既約多項式を返す.

- もし 既約 3 項式が存在すれば, 第 2 項の次数がもっとも小さい 3 項式, もし 既約 3 項式が存在しなければ, 既約 5 項式の中で, 第 2 項の次数がもっとも小さく, その中で第 3 項の次数がもっとも小さく, その中で第 4 項の次数がもっとも小さいものを返す.

参照　10.5.1 節「setmod_ff」p.152

10.5.15 sffctr

sffctr(*poly*)
:: 多項式の小標数有限体上での既約分解

return　リスト

poly　有限体上の 多項式

- 多項式を, 現在設定されている小標数有限体上で既約分解する.
- 結果は, [[*f1*,*m1*],[*f2*,*m2*],...] なるリストである. ここで, *fi* は monic な既約因子, *mi* はその重複度である.

```
[0] setmod_ff(2,10);
[2,x^10+x^3+1,x]
[1] sffctr((z*y^3+z*y)*x^3+(y^5+y^3+z*y^2+z)*x^2+z^11*y*x+z^10*y^3+z^11);
[[@_0,1],[@_0*z*y*x+@_0*y^3+@_0*z,1],[(@_0*y+@_0)*x+@_0*z^5,2]]
```

参照　10.5.1 節「setmod_ff」p.152, 6.3.17 節「modfctr」p.51

10.5.16 fctr_ff

fctr_ff(*poly*)
:: 1 変数多項式の有限体上での既約分解

return　リスト

poly　有限体上の 1 変数多項式

- 'fff' で定義されている.
- 一変数多項式を, 現在設定されている有限体上で既約分解する.
- 結果は, [[*f1*,*m1*],[*f2*,*m2*],...] なるリストである. ここで, *fi* は monic な既約因子, *mi* はその重複度である.
- *poly* の主係数は捨てられる.

```
[178] setmod_ff(2^64-95);
18446744073709551521
[179]  fctr_ff(x^5+x+1);
[[1*x+14123390394564558010,1],[1*x+6782485570826905238,1],
[1*x+15987612182027639793,1],[1*x^2+1*x+1,1]]
```

参照　10.5.1 節「setmod_ff」p.152

10.5.17 irredcheck_ff

irredcheck_ff(*poly*)
:: 1 変数多項式の有限体上での既約判定

return　0|1

poly　有限体上の 1 変数多項式

- 'fff' で定義されている.
- 有限体上の 1 変数多項式の既約判定を行い, 既約の場合 1, それ以外は 0 を返す.

```
[178] setmod_ff(2^64-95);
18446744073709551521
[179] ] F=x^10+random_ff();
x^10+14687973587364016969
[180] irredcheck_ff(F);
1
```

参照　10.5.1 節「setmod_ff」p.152

10.5.18 randpoly_ff

randpoly_ff(*d*,*v*)
:: 有限体上の 乱数係数 1 変数多項式の生成

return　多項式

d　正整数

v　不定元

- 'fff' で定義されている.
- *d* 次未満, 変数が *v*, 係数が現在設定されている有限体に属する 1 変数多項式を生成する. 係数は random_ff() により生成される.

```
[178] setmod_ff(2^64-95);
18446744073709551521
[179] ] F=x^10+random_ff();
[180] randpoly_ff(3,x);
17135261454578964298*x^2+476682669965361 5429*x+18317369440429479651
[181] randpoly_ff(3,x);
7565988813172050604*x^2+743007576727966 5339*x+4699662986224873544
[182] randpoly_ff(3,x);
10247781277095450395*x^2+10243690944992524936*x+4063829049268845492
```

参照　10.5.1 節「setmod_ff」p.152, 10.5.7 節「random_ff」p.155

10.5.19 ecm_add_ff, ecm_sub_ff, ecm_chsgn_ff

ecm_add_ff(*p1*,*p2*,*ec*)
ecm_sub_ff(*p1*,*p2*,*ec*)
ecm_chsgn_ff(*p1*)
:: 楕円曲線上の点の加算, 減算, 逆元

return　ベクトルまたは 0

p1 p2　長さ 3 のベクトルまたは 0

ec　長さ 2 のベクトル

- 現在設定されている有限体上で, *ec* で定義される楕円曲線上の点 *p1*, *p2* の和 *p1*+*p2*, 差 *p1*-*p2*, 逆元 -*p1* を返す.
- *ec* は, 設定されている有限体が奇標数素体の場合, y^2=x^3+ec[0]x+ec[1], 標数 2 の場合 y^2+xy=x^3+ec[0]x^2+ec[1] を表す.
- 引数, 結果ともに, 無限遠点は 0 で表される.
- *p1*, *p2* が長さ 3 のベクトルの場合, 斉次座標による曲線上の点を表す. この場合, 第 3 座標は 0 であってはいけない.
- 結果が長さ 3 のベクトルの場合, 第 3 座標は 0 でないが, 1 とは限らない. アフィン座標による結果を得るためには, 第 1 座標, 第 2 座標を第 3 座標で割る必要がある.
- *p1*, *p2* が楕円曲線上の点かどうかのチェックはしない.

```
[0] setmod_ff(1125899906842679)$
[1] EC=newvect(2,[ptolmp(1),ptolmp(1)])$
[2] Pt1=newvect(3,[1,-412127497938252,1])$
[3] Pt2=newvect(3,[6,-252647084363045,1])$
[4] Pt3=ecm_add_ff(Pt1,Pt2,EC);
[ 560137044461222 184453736165476 125 ]
[5] F=y^2-(x^3+EC[0]*x+EC[1])$
[6] subst(F,x,Pt3[0]/Pt3[2],y,Pt3[1]/Pt3[2]);
0
[7] ecm_add_ff(Pt3,ecm_chsgn_ff(Pt3),EC);
0
[8] D=ecm_sub_ff(Pt3,Pt2,EC);
[ 886545905133065 119584559149586 886545905133065 ]
[9] D[0]/D[2]==Pt1[0]/Pt1[2];
1
[10] D[1]/D[2]==Pt1[1]/Pt1[2];
1
```

参照　10.5.1 節「setmod_ff」p.152

付録 A 付録

A.1 文法の詳細

<式>:
　　‘(’<式>‘)’
　　<式> <二項演算子> <式>
　　‘+’ <式>
　　‘-’ <式>
　　<左辺値>
　　<左辺値> <代入演算子> <式>
　　<左辺値> ‘++’
　　<左辺値> ‘--’
　　‘++’ <左辺値>
　　‘--’ <左辺値>
　　‘!’ <式>
　　<式> ‘?’ <式> ‘:’ <式>
　　<函数> ‘(’ <式並び> ‘)’
　　<函数> ‘(’ <式並び> ‘|’ <オプション並び> ‘)’
　　<文字列>
　　<指数ベクトル>
　　<アトム>
　　<リスト>

(4.2.10 節「さまざまな式」p.22 を参照.)

<左辺値>:
　　<変数> [‘[’<式>‘]’]*

<二項演算子>:
　　‘+’ ‘-’ ‘*’ ‘/’ ‘%’ ‘^’(冪)
　　‘==’ ‘!=’ ‘<’ ‘>’ ‘<=’ ‘>=’ ‘&&’ ‘||’

<代入演算子>:
　　‘=’ ‘+=’ ‘-=’ ‘*=’ ‘/=’ ‘%=’ ‘^=’

<式並び>:
　　<空>
　　<式> [‘,’ <式>]*

<オプション>:
　　alphabet で始まる文字列 ‘=’ <式>

<オプション並び>:
　　<オプション>
　　<オプション> [‘,’ <オプション>]*

<リスト>:
　　‘[’ <式並び> ‘]’

<変数>:
　　大文字で始まる文字列 (X,Y,Japan など)

(4.2.2 節「変数および不定元」p.18 を参照.)

<函数>:
　　小文字で始まる文字列 (fctr,gcd など)

<アトム>:
　<不定元>
　<数>

<不定元>:
　小文字で始まる文字列 (a,bCD,c1_2 など)

(4.2.2 節「変数および不定元」p.18 を参照.)

<数>:
　<有理数>
　<浮動小数>
　<代数的数>
　<複素数>

(3.2 節「数の型」p.12 を参照.)

<有理数>:
　0, 1, -2, 3/4

<浮動小数>:
　0.0, 1.2e10

<代数的数>:
　newalg(x^2+1), alg(0)^2+1

(第 9 章「代数的数に関する演算」p.140 を参照.)

<複素数>:
　1+@i, 2.3*@i

<文字列>:
　‘"’ で囲まれた文字列

<指数ベクトル>:
　‘<<’ <式並び> ‘>>’

(第 8 章「グレブナ基底の計算」p.109 を参照.)

<文>:
　<式> <終端>
　<複文>
　‘break’ <終端>
　‘continue’ <終端>
　‘return’ <終端>
　‘return’ <式> <終端>
　‘if’ ‘(’ <式並び> ‘)’ <文>
　‘if’ ‘(’ <式並び> ‘)’ <文> ‘else’ <文>
　‘for’ ‘(’ <式並び> ‘;’ <式並び> ‘;’ <式並び> ‘)’ <文>
　‘do’ <文> ‘while’ ‘(’ <式並び> ‘)’ <終端>
　‘while’ ‘(’ <式並び> ‘)’ <文>
　‘def’ <函数> ‘(’ <式並び> ‘)’ ‘{’ <変数宣言> <文並び> ‘}’
　‘end(quit)’ <終端>

(4.2.5 節「文」p.20 を参照.)

<終端>:
　‘;’ ‘$’

<変数宣言>:
　[‘extern’ <変数> [‘,’ <変数>]* <終端>]*

```
<複文>:
    ‘{’ <文並び> ‘}’
<文並び>:
    [<文>]*
```

A.2 添付のユーザ定義函数ファイル

標準ライブラリディレクトリ (デフォルトでは ‘/usr/local/lib/asir’) にはいくつかのユーザ定義函数ファイルがおかれている. これらのうちの主なものについて説明する.

‘fff’　大標数素体および標数 2 の有限体上の一変数多項式因数分解 (第 10 章「有限体に関する演算」p.150 を参照.)

‘gr’　グレブナ基底計算パッケージ. (第 8 章「グレブナ基底の計算」p.109 を参照.)

‘sp’　代数的数の演算および因数分解, 最小分解体. (第 9 章「代数的数に関する演算」p.140 を参照.)

‘alpi’
‘bgk’
‘cyclic’
‘katsura’
‘kimura’　グレブナ基底計算において, ベンチマークその他で用いられる例. (8.10.30 節「katsura hkatsura cyclic hcyclic」p.136 を参照.)

‘defs.h’　いくつかのマクロ定義. (4.2.11 節「プリプロセッサ」p.23 を参照.)

‘fctrtest’
整数上の多項式の因数分解のテスト. REDUCE の ‘factor.tst’ および重複度の大きいいくつかの例を含む. これは, load() すると直ちに計算が始まる. 入手した **Asir** が正しく動作しているかのテストにも使うことができる.

‘fctrdata’
‘fctrtest’ で使われている例を含む, 因数分解テスト用の例. Alg[] に収められている例は, af() (9.4.10 節「asq af af_noalg」p.147) 用の例である.

```
[45] load("sp")$
[84] load("fctrdata")$
[175] cputime(1)$
0msec
[176] Alg[5];
x^9-15*x^6-87*x^3-125
0msec
[177] af(Alg[5],[newalg(Alg[5])]);
[[1,1],[75*x^2+(10*#0^7-175*#0^4-470*#0)*x
+(3*#0^8-45*#0^5-261*#0^2),1],
[75*x^2+(-10*#0^7+175*#0^4+395*#0)*x
+(3*#0^8-45*#0^5-261*#0^2),1],
[25*x^2+(25*#0)*x+(#0^8-15*#0^5-87*#0^2),1],
[x^2+(#0)*x+(#0^2),1],[x+(-#0),1]]
3.600sec + gc : 1.040sec
```

‘ifplot’　描画 (7.5.15 節「ifplot conplot plot polarplot plotover」p.105) のための例. IS[] には有名な曲線の例, 変数 H, D, C, S にはトランプのハート, ダイヤ, クラブ, スペード (らしき) 曲線の例が入っている.

‘num’　数に関する簡単な演算函数の例.

‘mat’　行列に関する簡単な演算函数の例.

‘ratint’　有理函数の不定積分. ‘sp’, ‘gr’ が必要. ratint() という函数が定義されているが, その返す結果はやや複雑である. 例で説明する.

```
[0] load("gr")$
[45] load("sp")$
[84] load("ratint")$
[102] ratint(x^6/(x^5+x+1),x);
[1/2*x^2,
[[(#2)*log(-140*x+(-2737*#2^2+552*#2-131)),
161*t#2^3-23*t#2^2+15*t#2-1],
[(#1)*log(-5*x+(-21*#1-4)),21*t#1^2+3*t#1+1]]]
```

この例では, x^6/(x^5+x+1) の不定積分の計算を行っている. 結果は 2 つの要素からなるリストで, 第 1 要素は不定積分の有理部分, 第 2 要素は対数部分を表す. 対数部分は更にリストとなっていて, 各要素は, [root*log(poly),defpoly] という形をしている. これは, 不定積分においては, defpoly の全ての根 root に対して root*log(poly) を作りそれらを足し合わせるという意味である. ここで poly は root を含んでいて, root を入れ替える場合には poly に対しても同じ操作を行うものとする. この操作を, 結果の第 2 要素の各成分に対して行って, 全てを足し合わせたものが対数部分となる.

‘primdec’　有理数体上の多項式イデアルの準素イデアル分解とその根基の素イデアル分解 (8.10.31 節「primadec primedec」p.137 参照).

‘primdec_mod’
有限体上の多項式イデアルの根基の素イデアル分解 (8.10.32 節「primedec_mod」p.137 参照).

‘bfct’　b 関数の計算. (8.10.33 節「bfunction bfct generic_bfct ann ann0」p.138 参照).

A.3 入力インタフェース

DOS 版, Windows 版では入力インタフェースとしてコマンドライン編集およびヒストリ置き換えが組み込まれている. UNIX 版ではこのような機能は組み込まれていないが, 以下で述べるような入力インタフェースが用意されている. これらは **Asir** バイナリとともに ftp 可能である. ftp server に関しては 1.3 節「入手方法」p.2 を参照.

Windows 版 ‘asirgui.exe’ は, 通常の Windows における慣習とは異なる形のコピーペースト機能を提供している. Window 上に表示されている文字列に対しマウス左ボタンを押しながらドラッグすると文字列が選択される. ボタンを離すと反転表示が元に戻るが, その文字列はコピーバッファに取り込まれている. マウス右ボタンを押すと, コピーバッファ内の文字列が現在のカーソル位置に挿入される. 既に表示された部分は readonly であり, その部分を改変できないことに注意して欲しい.

A.3.1 fep

fep とは, SRA の歌代氏により開発されたコマンドライン編集, ヒストリ置き換え用の入力フロントエンドである. このプログラムの元で 'asir' を起動することにより vi あるいは emacs 風のコマンドライン編集および csh 風のヒストリ置き換えが可能になる.

```
% fep asir
...
[0] fctr(x^5-1);
[[1,1],[x-1,1],[x^4+x^3+x^2+x+1,1]]
[1] !!                                  /* !!+Return                          */
fctr(x^5-1);                            /* 直前の入力が現れて編集できる */
...                                     /* 編集+Return                        */
fctr(x^5+1);
[[1,1],[x+1,1],[x^4-x^3+x^2-x+1,1]]
```

fep はフリーソフトでソースが入手可能であるが, オリジナルのものは make できる機種 (OS) が限られている. いくつかの機種上で動作するように我々が改造したものが, ftp で入手可能である.

A.3.2 asir.el

'asir.el' は, **Asir** の GNU Emacs インタフェースである (著者は宮嶋光治氏 (YVE25250@pcvan.or.jp). 'asir.el' においては, 通常の emacs で可能な編集機能の他に, ファイル名, コマンド名の completion が実現されている.

'asir.el' は PC-VAN で既に公開されているが, 今回の改訂に伴う変更を行ったものが, やはり ftp で入手可能である.

セットアップ, 使用方法は, 'asir.el' の先頭に記述されている.

A.4 ライブラリインタフェース

Asir の提供する機能を他のプログラムから使用する方法として, **OpenXM** による他に, ライブラリの直接リンクによる方法が可能である. ライブラリは, GC ライブラリである 'libasir-gc.a' とともに **OpenXM** distribution (http://www.math.kobe-u.ac.jp/OpenXM) に含まれる. 現状では **OpenXM** インタフェースのみが公開されているため, 以下では **OpenXM** がインストールされていると仮定する. **OpenXM** root ディレクトリを$OpenXM_HOMEと書く. ライブラリファイルは全て '$OpenXM_HOME/lib' におかれている. ライブラリには以下の 3 種類がある.

- 'libasir.a'
  PARI, **X11** 関連の機能を含まない. リンクには 'libasir-gc.a' のみが必要.
- 'libasir_pari.a'
  X11 関連の機能を含まない. リンクには 'libasir-gc.a', 'libpari.a' が必要.
- 'libasir_pari_X.a'
  全ての機能を含む. リンクには 'libasir-gc.a', 'libpari.a' および **X11** 関連のライブラリ指定が必要.

提供されている関数は以下の通りである.

- `int asir_ox_init(int` *byteorder*`)`
  ライブラリの初期化. *byteorder* はメモリ上へのバイナリ CMO データへの展開方法を指定する. *byteorder* が 0 のときマシン固有の byteorder を用いる. 1 のとき network byteorder を用いる. 初期化に成功した場合 0, 失敗の時 -1 を返す.
- `void asir_ox_push_cmo(void *`*cmo*`)`
  メモリ上に置かれた CMO データを **Asir** の内部形式に変換してスタックに push する. It converts CMO data pointed by *cmo* into an **Asir** object and it pushes the object onto the stack.
- `int asir_ox_peek_cmo_size()`
  スタックの最上位にある **Asir** データを CMO に変換したときのサイズを返す. 変換不能な場合には -1 を返す.
- `int asir_ox_pop_cmo(void *`*cmo*`, int` *limit*`)`
  スタックの最上位にある **Asir** データを pop し, CMO に変換して *cmo* で指される配列に書き, CMO のサイズを返す. このとき, CMO のサイズが *limit* より大きい場合には -1 を返す. *cmo* は長さが少なくとも *limit* バイトの配列を指す必要がある. 変換された CMO を収容できる配列の長さを知るために, `asir_ox_peek_cmo_size` を用いる.
- `void asir_ox_push_cmd(int` *cmd*`)`
  スタックマシンコマンド *cmd* を実行する.
- `void asir_ox_execute_string(char *`*str*`)`
  Asir が実行可能な文字列 *str* を実行し, その結果をスタックに push する.

include すべき header file は '`$OpenXM_HOME/include/asir/ox.h`' である. この header file には, **OpenXM** に関する全ての tag, command の定義が含まれている. 次の例 ('`$OpenXM_HOME/doc/oxlib/test3.c`') は上記関数の使用法を示す.

```
#include <asir/ox.h>
#include <signal.h>

main(int argc, char **argv)
{
  char buf[BUFSIZ+1];
  int c;
  unsigned char sendbuf[BUFSIZ+10];
  unsigned char *result;
  unsigned char h[3];
  int len,i,j;
  static int result_len = 0;
  char *kwd,*bdy;
  unsigned int cmd;

  signal(SIGINT,SIG_IGN);
  asir_ox_init(1); /* 1: network byte order; 0: native byte order */
  result_len = BUFSIZ;
  result = (void *)malloc(BUFSIZ);
  while ( 1 ) {
    printf("Input>"); fflush(stdout);
    fgets(buf,BUFSIZ,stdin);
    for ( i = 0; buf[i] && isspace(buf[i]); i++ );
```

```
if ( !buf[i] )
  continue;
kwd = buf+i;
for ( ; buf[i] && !isspace(buf[i]); i++ );
buf[i] = 0;
bdy = buf+i+1;
if ( !strcmp(kwd,"asir") ) {
  sprintf(sendbuf,"%s;",bdy);
  asir_ox_execute_string(sendbuf);
} else if ( !strcmp(kwd,"push") ) {
  h[0] = 0;
  h[2] = 0;
  j = 0;
  while ( 1 ) {
    for ( ; (c= *bdy) && isspace(c); bdy++ );
    if ( !c )
      break;
    else if ( h[0] ) {
      h[1] = c;
      sendbuf[j++] = strtoul(h,0,16);
      h[0] = 0;
    } else
      h[0] = c;
    bdy++;
  }
  if ( h[0] )
    fprintf(stderr,"Number of characters is odd.\n");
  else {
    sendbuf[j] = 0;
    asir_ox_push_cmo(sendbuf);
  }
} else if ( !strcmp(kwd,"cmd") ) {
  cmd = atoi(bdy);
  asir_ox_push_cmd(cmd);
} else if ( !strcmp(kwd,"pop") ) {
  len = asir_ox_peek_cmo_size();
  if ( !len )
    continue;
  if ( len > result_len ) {
    result = (char *)realloc(result,len);
    result_len = len;
  }
  asir_ox_pop_cmo(result,len);
  printf("Output>"); fflush(stdout);
  printf("\n");
  for ( i = 0; i < len; ) {
    printf("%02x ",result[i]);
    i++;
    if ( !(i%16) )
```

```
            printf("\n");
          }
          printf("\n");
        }
      }
    }
```

このプログラムは, *keyword body* なる 1 行を入力として受け取り *keyword* に応じて次のような動作を行う.

- `asir` *body*
  body を **Asir** 言語で書かれた式とみなし, 実行結果をスタックに push する. `asir_ox_execute_string()` が用いられる.
- `push` *body*
  body を 16 進数で表示された CMO データとみなし, **Asir** オブジェクトに変換してスタックに push する. `asir_ox_push_cmo()` が用いられる.
- `pop`
  スタック最上位のオブジェクトを CMO に変換し, 16 進数で表示する. `asir_ox_peek_cmo_size()` および `asir_ox_pop_cmo()` が用いられる.
- `cmd` *body*
  body を SM コマンドとみなし, 実行する. `asir_ox_push_cmd()` が用いられる.

A.5 変更点

A.5.1 Version 990831

4 年ぶりの大改訂. 整数の 32bit 化他, 中身はずいぶん変わっているものの, 見掛けはそれほど変わっているようには見えない. むしろ, Windows 版などは, plot が使えないため, 退化している.

旧版のユーザがもっとも注意すべき点は, 旧版で作った bsave file を読み込む場合は `bload27` を使う必要がある, という点である.

A.5.2 Version 950831

A.5.2.1 デバッガ

- 任意の時点にデバッグモードに入れる.
- `finish` コマンドの追加.
- `up`, `down`, `frame` コマンドによる, 任意のスタックフレームの参照.
- `trace` コマンドの追加.

A.5.2.2 組み込み函数

- `sdiv()` などにおける, 主変数の指定のサポート.
- `sdivm()` など, 有限体上での多項式除算の追加.
- `det()`, `res()` などにおける, 有限体上での計算のサポート

- vtol() (ベクトルからリストへの変換) の追加.
- map() の追加.

A.5.2.3 グレブナ基底

- グレブナ基底計算機能の組み込み函数化.
- grm(), hgrm() が gr(), hgr() に変更.
- gr(), hgr() において, 項順序の指定が必要になった.
- 項順序の指定方法が拡張された.
- 有限体上のグレブナ基底計算のサポート.
- 基底変換による辞書式順序グレブナ基底計算のサポート.
- いくつかの新しい組み込み函数の提供.

A.5.2.4 その他

- 分散計算用ツール, 函数の追加.
- 代数体上の GCD 計算におけるモジュラ計算の応用.
- イデアルの準素分解のサポート.
- Windows への移植.

A.5.3 Version 940420

最初の公開版.

A.6 文献

[Batut et al.]
Batut, C., Bernardi, D., Cohen, H., Olivier, M., "User's Guide to PARI-GP", 1993.

[Becker,Weispfenning]
Becker, T., Weispfenning, V., "Groebner Bases", Graduate Texts in Math. 141, Springer-Verlag, 1993.

[Boehm,Weiser]
Boehm, H., Weiser, M., "Garbage Collection in an Uncooperative Environment", Software Practice & Experience, September 1988, 807-820.

[Gebauer,Moeller]
Gebauer, R., Moeller, H. M., "An installation of Buchberger's algorithm", J. of Symbolic Computation 6, 275-286.

[Giovini et al.]
Giovini, A., Mora, T., Niesi, G., Robbiano, L., Traverso, C., ""One sugar cube, please" OR Selection strategies in the Buchberger algorithm", Proc. ISSAC'91, 49-54.

[Noro,Takeshima]
Noro, M., Takeshima, T., "Risa/Asir – A Computer Algebra System", Proc. ISSAC'92, 387-396.

[Noro,Yokoyama]
Noro, M., Yokoyama, K., "A Modular Method to Compute the Rational Univariate Representation of Zero-Dimensional Ideals", J. Symb. Comp. 28/1 (1999), 243-263.

[Saito,Sturmfels,Takayama]
Saito, M., Sturmfels, B., Takayama, N., "Groebner deformations of hypergeometric differential equations", Algorithms and Computation in Mathematics 6, Springer-Verlag (2000).

[Shimoyama,Yokoyama]
Shimoyama, T., Yokoyama, K., "Localization and primary decomposition of polynomial ideals", J. Symb. Comp. 22 (1996), 247-277.

[Shoup] Shoup, V., "A new polynomial factorization algorithm and its implementation", J. Symb. Comp. 20 (1995), 364-397.

[Traverso]
Traverso, C., "Groebner trace algorithms", Proc. ISSAC '88(LNCS 358), 125-138.

[Weber] Weber, K., "The accelerated Integer GCD Algorithm", ACM TOMS, 21, 1(1995), 111-122.

[Yokoyama]
Yokoyama, K., "Prime decomposition of polynomial ideals over finite fields", Proc. ICMS, (2002), 217-227.

索引

%

% 47

A

access 77
af 147
af_noalg 147
alg 145
algptorat 146
algv 145
ann 138
ann0 138
append 58
arfreg 65
args 70
asciitostr 72
asq 147

B

bfct 138
bfunction 138
bload 76
bload27 76
bsave 76

C

call 70
car 58
cdr 58
ceil 81
characteristic_ff 153
clear_canvas 107
close_file 78
coef 44
conj 37
conplot 105
cons 58
cputime 85
cr_gcda 146
ctrl 82
currenttime 86
cyclic 136

D

dabs 81
dacos 80
dasin 80
datan 80
dceil 81
dcos 80
debug 83
defpoly 144
defpoly_mod2 158
deg 44
delete_history 89
det 63
deval 38
dexp 81
dfloor 81
dgr 117
diff 48
dlog 81
dn 37
dp_dehomo 127
dp_dtop 126
dp_etov 132
dp_f4_main 123
dp_f4_mod_main 123
dp_gr_f_main 122
dp_gr_flags 124
dp_gr_main 122
dp_gr_mod_main 122
dp_gr_print 124
dp_hc 129
dp_hm 129
dp_homo 127
dp_ht 129
dp_lcm 130
dp_mag 132
dp_mbase 132
dp_mod 126
dp_nf 128
dp_nf_mod 128
dp_ord 125
dp_prim 127
dp_ptod 126
dp_ptozp 127
dp_rat 126
dp_red 133
dp_red_mod 133
dp_redble 131
dp_rest 129
dp_sp 134
dp_sp_mod 134
dp_subd 131

dp_sugar 130
dp_td 130
dp_true_nf 128
dp_true_nf_mod 128
dp_vtoe 132
dp_weyl_f4_main 123
dp_weyl_f4_mod_main 123
dp_weyl_gr_f_main 122
dp_weyl_gr_main 122
dp_weyl_gr_mod_main 122
draw_obj 107
draw_string 107
drint 81
dsin 80
dsqrt 81
dtan 80

E

ecm_add_ff 160
ecm_chsgn_ff 160
ecm_sub_ff 160
ediff 49
end 74
error 83
eval 38
eval_str 72
extdeg_ff 154

F

fac 33
fctr 50
fctr_ff 159
field_order_ff 153
field_type_ff 153
flist 89
floor 81
funargs 70
functor 70

G

gb_comp 135
gcd 53
generate_port 97
generic_bfct 138
get_byte 78
get_line 78
get_rootdir 90
getenv 90
getopt 90
gf2nton 156
gf2ntop 157
gr 117
gr_minipoly 121
gr_mod 117

H

hcyclic 136
heap 87
help 84
hgr 117
hkatsura 136

I

iand 41
idiv 33
ifplot 105
igcd 34
igcdcntl 34
ilcm 34
int32ton 41
inv 35
invmat 63
ior 41
irem 33
irredcheck_ff 159
ishift 42
isqrt 35
ixor 41

K

katsura 136
kmul 55
ksquare 55
ktmul 55

L

length 58
lex_hensel 118
lex_hensel_gsl 120
lex_tl 118
lmptop 155
load 74
lprime 35
lrandom 36
ltov 61

M

map ... 88
mindeg ... 44
minipoly ... 121
minipolym ... 121
modfctr ... 51
module_definedp ... 79
module_list ... 79
mt_load ... 36
mt_save ... 36

N

nd_f4 ... 123
nd_gr ... 123
nd_gr_trace ... 123
nd_weyl_gr ... 123
nd_weyl_gr_trace ... 123
newalg ... 144
newbytearray ... 61
newmat ... 62
newstruct ... 65
newvect ... 60
nm ... 37
nmono ... 44
ntogf2n ... 156
ntoint32 ... 41
ntype ... 69

O

open_canvas ... 107
open_file ... 78
ord ... 45
output ... 75
ox_cmo_rpc ... 98
ox_execute_string ... 98
ox_flush ... 103
ox_get ... 102
ox_get_serverinfo ... 104
ox_launch ... 94
ox_launch_generic ... 96
ox_launch_nox ... 94
ox_pop_cmo ... 101
ox_pop_local ... 101
ox_pops ... 102
ox_push_cmd ... 101
ox_push_cmo ... 100
ox_push_local ... 100
ox_reset ... 99
ox_rpc ... 98
ox_select ... 103
ox_shutdown ... 94
ox_sync ... 101

P

p_nf ... 134
p_nf_mod ... 134
p_terms ... 135
p_true_nf ... 134
p_true_nf_mod ... 134
pari ... 39
plot ... 105
plotover ... 105
polarplot ... 105
prim ... 52
primadec ... 137
prime ... 35
primedec ... 137
primedec_mod ... 137
print ... 77
psubst ... 48
ptogf2n ... 157
ptosfp ... 158
ptozp ... 52
purge_stdin ... 78
put_byte ... 78

Q

qsort ... 64
quit ... 74

R

random ... 36
random_ff ... 155
randpoly_ff ... 160
rattoalgp ... 146
red ... 54
register_handler ... 99
register_server ... 97
remove_file ... 77
remove_module ... 80
res ... 49
reverse ... 58
rint ... 81
rtostr ... 71

S

sdiv ... 45
sdivm ... 45
set_upfft ... 56
set_upkara ... 56
set_uptkara ... 56
setmod ... 40
setmod_ff ... 152
setprec ... 40
sffctr ... 159
sfptop ... 158
shell ... 88
simp_ff ... 154
simpalg ... 145
size ... 63
sleep ... 87
sp ... 148
sp_norm ... 147
sqfr ... 50
sqr ... 45
sqrm ... 45
srem ... 45
sremm ... 45
str_chr ... 73
str_len ... 73
strtoascii ... 72
strtov ... 72
struct_type ... 67
sub_str ... 73
subst ... 48

T

tdiv ... 47
time ... 85
timer ... 86
tolex ... 118
tolex_d ... 118
tolex_gsl ... 120
tolex_gsl_d ... 120
tolex_tl ... 118

P

tolexm ... 121
try_accept ... 97
try_bind_listen ... 97
try_connect ... 97
tstart ... 85
tstop ... 85
type ... 68

U

uc ... 43
udecomp ... 56
udiv ... 58
ufctrhint ... 50
ugcd ... 58
uinv_as_power_series ... 57
umul ... 54
umul_ff ... 54
urem ... 58
urembymul ... 58
urembymul_precomp ... 58
ureverse ... 56
ureverse_inv_as_power_series ... 57
usquare ... 54
usquare_ff ... 54
utmul ... 54
utmul_ff ... 54
utrunc ... 56

V

var ... 42
vars ... 43
version ... 88
vtol ... 61
vtype ... 69

W

which ... 74

PARI ... 38, 39, 40, 82

簡略目次

1 Introduction 1
2 Risa/Asir 3
3 型 10
4 ユーザ言語 Asir 16
5 デバッガ 28
6 組み込み函数 32
7 分散計算 90
8 グレブナ基底の計算 108
9 代数的数に関する演算 139
10 有限体に関する演算 149
付録 A 付録 161
索引 171

目次

1 Introduction 1

1.1 マニュアルの構成 1
1.2 Notation 1
1.3 入手方法 2

2 Risa/Asir 3

2.1 Risa および Asir 3
2.2 Asir の特徴 3
2.3 Installation 3
2.3.1 UNIX binary version 4
2.3.2 UNIX source code version 4
2.3.3 Windows version 5
2.4 コマンドラインオプション 5
2.5 環境変数 6
2.6 起動から終了まで 6
2.7 割り込み 7
2.8 エラー処理 8
2.9 計算結果, 特殊な数 8

3 型 10

3.1 **Asir** で使用可能な型 10
3.2 数の型 12
3.3 不定元の型 14

4 ユーザ言語 Asir 16

4.1 文法 (C 言語との違い) 16
4.2 ユーザ定義函数の書き方 17
4.2.1 ユーザ定義函数 17
4.2.2 変数および不定元 18
4.2.3 引数 19
4.2.4 コメント 19
4.2.5 文 20
4.2.6 `return` 文 20
4.2.7 `if` 文 21
4.2.8 ループ, `break`, `return`, `continue` 21
4.2.9 構造体定義 22
4.2.10 さまざまな式 22
4.2.11 プリプロセッサ 23
4.2.12 オプション指定 24
4.2.13 モジュール 25

5 デバッガ 28
5.1 デバッガとは 28
5.2 コマンドの解説 28
5.3 デバッガの使用例 30
5.4 デバッガの初期化ファイルの例 31

6 組み込み函数 32
6.1 数の演算 32
6.1.1 idiv, irem 32
6.1.2 fac 32
6.1.3 igcd,igcdcntl 33
6.1.4 ilcm 33
6.1.5 isqrt 34
6.1.6 inv 34
6.1.7 prime, lprime 34
6.1.8 random 35
6.1.9 lrandom 35
6.1.10 mt_save, mt_load 35
6.1.11 nm, dn 36
6.1.12 conj, real, imag 36
6.1.13 eval, deval 37
6.1.14 pari 38
6.1.15 setprec 39
6.1.16 setmod 39
6.1.17 ntoint32, int32ton 40
6.2 bit 演算 40
6.2.1 iand, ior, ixor 40
6.2.2 ishift 41
6.3 多項式, 有理式の演算 41
6.3.1 var 41
6.3.2 vars 42
6.3.3 uc 42
6.3.4 coef 43
6.3.5 deg, mindeg 43
6.3.6 nmono 43
6.3.7 ord 44
6.3.8 sdiv, sdivm, srem, sremm, sqr, sqrm 44
6.3.9 tdiv 46
6.3.10 % 46
6.3.11 subst, psubst 47
6.3.12 diff 47
6.3.13 ediff 48
6.3.14 res 48
6.3.15 fctr, sqfr 49
6.3.16 ufctrhint 49
6.3.17 modfctr 50
6.3.18 ptozp 51
6.3.19 prim, cont 51

6.3.20 gcd, gcdz 52
6.3.21 red 53
6.4 一変数多項式の演算 53
6.4.1 umul, umul_ff, usquare, usquare_ff, utmul, utmul_ff 53
6.4.2 kmul, ksquare, ktmul 54
6.4.3 set_upkara, set_uptkara, set_upfft 55
6.4.4 utrunc, udecomp, ureverse 55
6.4.5 uinv_as_power_series, ureverse_inv_as_power_series 56
6.4.6 udiv, urem, urembymul, urembymul_precomp, ugcd 57
6.5 リストの演算 57
6.5.1 car, cdr, cons, append, reverse, length 57
6.6 配列 59
6.6.1 newvect 59
6.6.2 ltov 60
6.6.3 vtol 60
6.6.4 newbytearray 60
6.6.5 newmat 61
6.6.6 size 62
6.6.7 det,invmat 62
6.6.8 qsort 63
6.7 構造体 64
6.7.1 newstruct 64
6.7.2 arfreg 64
6.7.3 struct_type 66
6.8 型を求める函数 67
6.8.1 type 67
6.8.2 ntype 68
6.8.3 vtype 68
6.9 函数に対する操作 69
6.9.1 call 69
6.9.2 functor, args, funargs 69
6.10 文字列に関する演算 70
6.10.1 rtostr 70
6.10.2 strtov 71
6.10.3 eval_str 71
6.10.4 strtoascii, asciitostr 71
6.10.5 str_len, str_chr, sub_str 72
6.11 入出力 72
6.11.1 end, quit 73
6.11.2 load 73
6.11.3 which 73
6.11.4 output 74
6.11.5 bsave, bload 75
6.11.6 bload27 75
6.11.7 print 76
6.11.8 access 76

6.11.9 remove_file ... 76
6.11.10 open_file, close_file, get_line, get_byte, put_byte, purge_stdin ... 77
6.12 モジュールに対する操作 ... 78
6.12.1 module_list ... 78
6.12.2 module_definedp ... 78
6.12.3 remove_module ... 79
6.13 数値函数 ... 79
6.13.1 dacos, dasin, datan, dcos, dsin, dtan ... 79
6.13.2 dabs, dexp, dlog, dsqrt ... 80
6.13.3 ceil, floor, rint, dceil, dfloor, drint ... 80
6.14 その他 ... 81
6.14.1 ctrl ... 81
6.14.2 debug ... 82
6.14.3 error ... 82
6.14.4 help ... 83
6.14.5 time ... 83
6.14.6 cputime, tstart, tstop ... 84
6.14.7 timer ... 85
6.14.8 currenttime ... 85
6.14.9 sleep ... 85
6.14.10 heap ... 86
6.14.11 version ... 86
6.14.12 shell ... 87
6.14.13 map ... 87
6.14.14 flist ... 88
6.14.15 delete_history ... 88
6.14.16 get_rootdir ... 88
6.14.17 getopt ... 89
6.14.18 getenv ... 89

7 分散計算 ... 90

7.1 OpenXM ... 90
7.2 Mathcap ... 91
7.3 スタックマシンコマンド ... 91
7.4 デバッグ ... 92
7.4.1 エラーオブジェクト ... 92
7.4.2 リセット ... 93
7.4.3 デバッグ用ポップアップウィンドウ ... 93
7.5 分散計算に関する関数 ... 93
7.5.1 ox_launch, ox_launch_nox, ox_shutdown ... 93
7.5.2 ox_launch_generic ... 95
7.5.3 generate_port, try_bind_listen, try_connect, try_accept, register_server ... 96
7.5.4 'ox_asir' ... 97
7.5.5 ox_rpc, ox_cmo_rpc, ox_execute_string ... 97
7.5.6 ox_reset,ox_intr,register_handler ... 98
7.5.7 ox_push_cmo, ox_push_local ... 99

7.5.8 ox_pop_cmo, ox_pop_local 100
7.5.9 ox_push_cmd, ox_sync 100
7.5.10 ox_get 101
7.5.11 ox_pops 101
7.5.12 ox_select 102
7.5.13 ox_flush 102
7.5.14 ox_get_serverinfo 103
7.5.15 ifplot, conplot, plot, polarplot, plotover 104
7.5.16 open_canvas, clear_canvas, draw_obj, draw_string 106

8 グレブナ基底の計算 108

8.1 分散表現多項式 108
8.2 ファイルの読み込み 108
8.3 基本的な函数 109
8.4 計算および表示の制御 109
8.5 項順序の設定 112
8.6 Weight 114
8.7 有理式を係数とするグレブナ基底計算 115
8.8 基底変換 115
8.9 Weyl 代数 116
8.10 グレブナ基底に関する函数 116
8.10.1 gr, hgr, gr_mod, dgr 116
8.10.2 lex_hensel, lex_tl, tolex, tolex_d, tolex_tl ... 117
8.10.3 lex_hensel_gsl, tolex_gsl, tolex_gsl_d 119
8.10.4 gr_minipoly, minipoly 120
8.10.5 tolexm, minipolym 120
8.10.6 dp_gr_main, dp_gr_mod_main, dp_gr_f_main, dp_weyl_gr_main, dp_weyl_gr_mod_main, dp_weyl_gr_f_main 121
8.10.7 dp_f4_main, dp_f4_mod_main, dp_weyl_f4_main, dp_weyl_f4_mod_main 122
8.10.8 nd_gr, nd_gr_trace, nd_f4, nd_weyl_gr, nd_weyl_gr_trace 122
8.10.9 dp_gr_flags, dp_gr_print 123
8.10.10 dp_ord 124
8.10.11 dp_ptod 125
8.10.12 dp_dtop 125
8.10.13 dp_mod, dp_rat 125
8.10.14 dp_homo, dp_dehomo 126
8.10.15 dp_ptozp, dp_prim 126
8.10.16 dp_nf, dp_nf_mod, dp_true_nf, dp_true_nf_mod 127
8.10.17 dp_hm, dp_ht, dp_hc, dp_rest 128
8.10.18 dp_td, dp_sugar 129
8.10.19 dp_lcm 129
8.10.20 dp_redble 130
8.10.21 dp_subd 130

8.10.22 dp_vtoe, dp_etov 131
8.10.23 dp_mbase 131
8.10.24 dp_mag 131
8.10.25 dp_red, dp_red_mod 132
8.10.26 dp_sp, dp_sp_mod 133
8.10.27 p_nf, p_nf_mod, p_true_nf, p_true_nf_mod 133
8.10.28 p_terms 134
8.10.29 gb_comp 134
8.10.30 katsura, hkatsura, cyclic, hcyclic 135
8.10.31 primadec, primedec 136
8.10.32 primedec_mod 136
8.10.33 bfunction, bfct, generic_bfct, ann, ann0 137

9 代数的数に関する演算 139
9.1 代数的数の表現 139
9.2 代数的数の演算 140
9.3 代数体上での 1 変数多項式の演算 141
9.3.1 GCD 142
9.3.2 無平方分解, 因数分解 142
9.3.3 最小分解体 142
9.4 代数的数に関する函数のまとめ 143
9.4.1 newalg 143
9.4.2 defpoly 143
9.4.3 alg 144
9.4.4 algv 144
9.4.5 simpalg 144
9.4.6 algptorat 145
9.4.7 rattoalgp 145
9.4.8 cr_gcda 145
9.4.9 sp_norm 146
9.4.10 asq, af, af_noalg 146
9.4.11 sp, sp_noalg 147

10 有限体に関する演算 149
10.1 有限体の表現および演算 149
10.2 有限体上での 1 変数多項式の演算 150
10.3 小標数有限体上での多項式の演算 150
10.4 有限体上の楕円曲線に関する演算 150
10.5 有限体に関する函数のまとめ 151
10.5.1 setmod_ff 151
10.5.2 field_type_ff 152
10.5.3 field_order_ff 152
10.5.4 characteristic_ff 152
10.5.5 extdeg_ff 153
10.5.6 simp_ff 153
10.5.7 random_ff 154
10.5.8 lmptop 154
10.5.9 ntogf2n 155

10.5.10 gf2nton ... 155
10.5.11 ptogf2n ... 156
10.5.12 gf2ntop ... 156
10.5.13 ptosfp, sfptop ... 157
10.5.14 defpoly_mod2 ... 157
10.5.15 sffctr ... 158
10.5.16 fctr_ff ... 158
10.5.17 irredcheck_ff ... 158
10.5.18 randpoly_ff ... 159
10.5.19 ecm_add_ff, ecm_sub_ff, ecm_chsgn_ff ... 159

付録 A 付録 ... 161

A.1 文法の詳細 ... 161
A.2 添付のユーザ定義函数ファイル ... 163
A.3 入力インタフェース ... 164
A.3.1 fep ... 165
A.3.2 asir.el ... 165
A.4 ライブラリインタフェース ... 165
A.5 変更点 ... 168
A.5.1 Version 990831 ... 168
A.5.2 Version 950831 ... 168
A.5.2.1 デバッガ ... 168
A.5.2.2 組み込み函数 ... 168
A.5.2.3 グレブナ基底 ... 169
A.5.2.4 その他 ... 169
A.5.3 Version 940420 ... 169
A.6 文献 ... 169

索引 ... 171