# Risa/Asir ドリル, 2003

高山信毅, 野呂正行

2003 年 (平成 15 年), 9 月 22 日 版: コメントは
takayama@math.kobe-u.ac.jp または noro@math.kobe-u.ac.jp まで

はじめに

Risa/Asir ドリルは, 著者 (T) が徳島大学総合科学部および神戸大学理学部で数学系の学生におこなってきた計算機プログラミングの入門講義および演習をたたきだいにして書いた本である. この講義ではさまざまな言語 C, Pascal, 8086 の機械語, Ubasic, sm1, Mathematica 等を用いて実習してきたが, 利用するプログラム言語は異なっても基本的内容は同じであった. ちなみに本書の原稿のおおもとは, 1991 年に著者 (T) が学生に配布した, Mathematica の入門テキストである.

2000 年の秋に著者 (N) が富士通研究所より神戸大学へ転職してきたのを機会に, 著者 (T) はこの年の計算機プログラミングの入門講義および演習を富士通研究所で著者 (N) が開発にたづさわってきた数式処理システム Risa/Asir を用いておこなうことにした. Risa/Asir は研究用システムとしてすぐれた点を多くもつシステムであったが, 教育用途に利用するにはいろいろと不満な点もあった. 著者 (N) は, 著者 (T) およびいろいろと珍奇なことをやってくれる学生にのせられて Risa/Asir を教育用途にも使えるよういろいろと改造した. 実習室用に CDROM をいれるだけで Windows 2000 で起動できる Asir, 初心者にやさしい入力エラーの取扱い, ファイル IO, 2 次元簡易グラフィックス, さらには教育用途のために, メモリを直接読み書きする peek, poke まで付けた (ちなみにこれはセキュリティホールになるので普通は利用できない). この Risa/Asir をつかった実習は著者 (T) がおこなった他のいろいろな言語による実習のなかでも成功の部類にはいるものであり, 本として出版してみようという気になったのである.

さて前にもいったようにこの講義および演習は数学科の学生向けであった. 講義の目的は以下のとおりである (目標はそもそもなかなか達成できないものであるが...).

1. 高校数学 A, B, C の計算機に関する部分を教えられるような最低限の知識と技術を身につける.
2. 数学科の学生は卒業後, 計算機ソフトウエア関連の職業につくことが多いが, その基礎となるような計算機科学の全体的な基礎知識を得る.
3. 計算数学が現代の科学技術のなかでどのように利用されているかおぼろげに理解してもらう. また計算数学が数学のなかの一つの研究分野であることを理解してもらい, とくに計算代数への入門を目指す.
4. 数学を活用する仕事 (含む数学者) についた場合に, 数式処理システム等を自由に使えるようにする.

この本では講義の内容に加えてさらに数学的アルゴリズムに関する特論的な内容や, Asir のライブラリプログラムを書くための方法, Asir に C のプログラムを組み込むための方法, 計算数学のシステムをたがいに接続する実験プロジェクトである, OpenXM を利用した分散計算法など, オープンソースソフトとしてリスタートした, Risa/Asir のための情報も加筆した. この本が Risa/Asir の利用者, 開発参加者にも役立つことを願っている.

なお, この本は数学科の学生向けの講義をもとに書かれたが, 一部分を除き大学理系の微分積分学, 線形代数学程度の知識があれば十分理解可能である. 実際本書は工学系の学生や, 高校生向けに利用したこともある. この本がさまざまな人にとり有益であることを願っている.

2002 年 (平成 14 年)10 月, 著者

# 目 次

# 第1章　Asir の操作

この章では計算機の操作法を簡単に説明し, Risa/Asir の起動法, 電卓風の使い方を説明する. 計算機の操作法は要点しか書いていないので, 初心者の人は適当な本を参照されたい. 講義では, プロジェクタで操作方法を説明したあと, 実習を 2, 3 時間おこない練習した.

## 1.1　login, logout, キー操作 (unix 編)

UNIX コンピュータの利用は login (利用者の認証）で開始であり, logout で利用を終了する.
キーボード, マウスの操作:

1. キーはちょんとおせばよい. おしつづけるとその文字がいっぱい入力される.
2. ALT キーや SHIFT キーやCTRL キーは他のキーと一緒に押すことで始めて機能するキーである. これらだけを単独に押してもなにもおきない. 以後 SHIFT キーをおしながら他のキーを押す操作を SHIFT+きー と書くことにする. alt キー, ctrl キーについても同様である.
3. SHIFT+a とすると大文字の A を入力できる.
4. BS とか DEL と書いてあるキー押すと一文字前を消去できる.
5. SPACE キーは空白を入力するキーである. 空白も一つの文字である.(アスキーコード 20H )
6. ’ (シングルクオート) と ‘ (バッククオート) は別の文字である. 注意すること. また, プログラムリストを読む時は 0 (ゼロ）と o （おー）の違いにも注意.
7. マウスの操作には次の三種類がある.
   (a) クリック: 選択するとき, 文字を入力する位置（キャレットの位置）の移動に用いる. マウスのボタンをちょんとおす.
   (b) ドラッグ: 移動, サイズの変更, 範囲の指定, コピーのときなどに用いる. マウスのボタンを押しながら動かす.
   (c) ダブルクリック：プログラムの実行, open(ファイルを開く) をするために用いる. マウスのボタンを２回つずけてちょんちょんとおす. ダブルクリックをしたアイコンは白くなったり形状がかわることがおおい. ダブルクリックしたらしばらく待つ. 計算機が混んでいるときは起動に時間がかかることもあり. むやみにダブルクリックを繰り返すとその回数だけ起動されてなお遅くなる.

**問題 1.1**　login して logout してみる– 文字の入力とクリック.

**問題 1.2**　Web ブラウザ netscape を起動し大きさを変えてみる– ドラッグ.

**問題 1.3**　Netscape を利用して Risa/Asir のマニュアル（使用説明書）を読んでみよう. インターネットへの接続が可能なら, `http://www.openxm.org` と入力し，`More documents` を選択し，次のページで `Asir manual in Japanese and html` を選択することでマニュアルを読むことが可能である.

**問題 1.4**　パスワードの役割はなにか? どのようなパスワードをつかえばよいのか?

## 1.2　logon, logoff, キー操作 (Windows 編)

キー操作等に関しては unix 同様である. Windows の操作は基本的にマウスでグラフィカルに行うので, 言葉ではなかなか説明しにくい. Windows の操作に経験がなければ, 教師のプロジェクタによるデモをみてから自分でいろいろ試すとか, 友達に教えてもらうとかの方法が有効であろう. “できる Windows” などの本が本屋さんにたくさんならんでいる. これらの本は, カラーの図入りで Windows の使いかたを説明してるので, 分りやすい.

とりあえず,

1. ダブルクリックでプログラムを起動する方法,
2. ダブルクリックでフォルダを開く方法,
3. マウスでプルダウンメニューから実行したいメニューを選ぶ方法,
4. マウスでファイルをオープンする方法,
5. 基本的なキーボードの操作,
6. Windows の終了方法,
7. CDROM の挿入, 取り出し方法,

等を納得していれば, Asir の実習には十分である. プログラムの作成をとおしてこれらにさらに習熟していくのが良いであろう.

## 1.3　Risa/Asir の起動法と電卓的な使い方

Unix の場合 Risa/Asir を起動するには `kterm` や `xterm` などのシェルウインドーまたは Emacs のシェルウインドーで, `asir` と入力する.

*bash-2.03$*　`asir`　RETURN

ここで *bash-2.03$* は, シェルのプロンプト (入力促進記号) で利用しているシステムやシェルによりいろいろ変わる. たとえば　`%`　(C-シェルのプロンプト) もよく見かけるであろう. なお, 入力行の編集機能を利用したい場合は `fep asir` と入力して Risa/Asir を起動する.

Windows の場合, Risa/Asir のインストールされているフォルダ (ディレクトリ) をファイルエクスプローラで開き `asirgui` (Graphical User Inteface 版 Asir) のアイコン

をダブルクリックする. なお, asirgui は Windows 風のユーザインタフェースを持っているように見えるが実は全く違う. Unix の xterm 上で fep 付きで Asir を起動した時と似た動作をするように作成してあるので, Windows 風インタフェースだと思いこんで利用してはいけない. 以下 asirgui も単に asir とよぶ. asirgui の利用法については, 1.4 節をみよ.

Risa/Asir は次のように起動メッセージを出力し,

*[ 数字 ]*

なる asir のプロンプト (入力催促文字列) を出力する.

```
bash-2.03$      asir    RETURN

This is Risa/Asir, Version 20010917 (Kobe Distribution).
Copyright (C) 1994-2000, all rights reserved, FUJITSU LABORATORIES LIMITED.
Copyright 2000,2001, Risa/Asir committers, http://www.openxm.org/.
GC 5.3, copyright 1999, H-J. Boehm, A. J. Demers, Xerox, SGI, HP.
PARI 2.0.17(beta), copyright (C) 1989-1999,
   C. Batut, K. Belabas, D. Bernardi, H. Cohen and M. Olivier.

[0]
```

終了は Risa/Asir のプロンプトに対して　quit;　を入力する.

この節では, まず Asir を対話型電卓として利用する方法を説明する.

Asir は数の処理のみならず, 多項式の計算もできる. 電卓的に使うための要点を説明し例をあげよう.

**例題 1.1** +,-,*,/,^ はそれぞれ足し算, 引き算, かけ算, 割算, 巾乗. たとえば,

```
2*(3+5^4);
```

と入力すると $2(3+5^4)$ の値を計算して戻す.

```
[0]      2*(3+5^4);    RETURN
1256
```

**例題 1.2** sin(x), cos(x) はおなじみの三角関数. @pi は円周率を表す定数.

$\sin(x)$ や $\cos(x)$ の近似値を求めるには

```
deval(sin(3.14));
```

と入力する. deval は 64 bit の浮動小数点数により計算する. bit や精度については 11 章でくわしく説明する.

**例題 1.3**

$$\left\{\left(2+\frac{2}{3}\right)4+\frac{1}{3}\right\}+5$$

を Asir で計算してみよう. ; (セミコロン) までを入力してから RETURN を押すと実行する. セミコロン RETURN の入力がないと実行がはじまらない. セミコロン の代わりに $ (ドル記号) を用いると, 計算した値を印刷しない. 数学ではかっこことして, [,],{,} などがつかえるが Risa/Asir では (,) のみ. [,] や {,} は別の意味である.

```
[0]     ((2+2/3)*4+1/3)+5;    RETURN
16
[1]     ((2+2/3)*4+1/3)+5$    RETURN
[2]
```

**例題 1.4** Asir は多項式も計算できる.

1. 小文字ではじまる記号は多項式の変数である. たとえば x2 と書くと, $x2$ という名前の多項式の変数となる. $x$ かける 2 は x*2 と書く.
2. *poly* ; と入力すると *poly* を展開する.
3. fctr( *poly* ) は *poly* を有理数係数の多項式環で因数分解する.

```
[0]     fctr(x^10-1);    RETURN
[[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]     (2*x-1)*(2*x+1);    RETURN
4*x^2-1
[2]     (x+y)^3;    RETURN
y^3+3*x*y^2+3*x^2*y+x^3
```

fctr の出力は $x^{10}-1$ が

$$(x-1)^1,(x+1)^1,(x^4+x^3+x^2+x+1)^1,(x^4-x^3+x^2-x+1)^1$$

の積に因数分解されることを意味している.

**例題 1.5** plot(f); は $x$ の関数 $f$ のグラフを描く x の範囲を指定したいときはたとえば plot(f,[x,0,10]) と入力すると, x は 0 から 10 まで変化する.

```
0
[1]     plot(sin(x));    RETURN
  0
[2]     plot(sin(2*x)+0.5*sin(3*x),[x,-10,10]);    RETURN
```

**例題 1.6** 実行中の計算を中断したい時は CTRL+C (C は cancel の C) を入力する. すると

`interrupt ?(q/t/c/d/u/w/?)`

と表示されるので u RETURN を入力する. 次に

`Abort this computation? (y or n)`

と表示されるので y RETURN を入力する. (q/t/c/d/u/w/) の各文字の説明は ? を入力すれば読むことができる.

```
[0]     fctr(x^1000-y^1000);    RETURN
CTRL+C
        interrupt ?(q/t/c/d/u/w/?)     u   RETURN
        Abort this computation? (y or n)    y   RETURN
return to toplevel
[1]
```

変数に値をとっておくこともできる. 変数名は大文字で始まる. 関数名等は小文字で始まる. 英字の大文字, 子文字を区別しているので注意. Risa/Asir では多項式計算ができるが小文字で始まる文字列は多項式の変数名としても利用される.

**例題 1.7**

```
[0]     Kazu = 5;    RETURN
5
[1]     Hehehe = x+ 8;    RETURN
x+8
[1]     Kazu+Hehehe;    RETURN
x+13
```

くりかえしになるが, セミコロン; を入力しないと, RETURN キーを押しても実行ははじまらない. 計算機用語では ; RETURN で “入力の評価が始まる” という.

**例題 1.8** Risa/Asir を用いて次の値を求めてみよう

$$2+\sqrt{3}, \quad \frac{1}{2-\sqrt{3}}.$$

```
[0]     A=2+deval(3^(1/2));       RETURN
3.73205
[1]     B=1/(2-deval(3^(1/2)));       RETURN
3.73205
[2]     A-B;      RETURN
1.33227e-15
```

1.33227e-15 は $1.33227 \times 10^{-15}$ を意味する．$2+\sqrt{3} = \frac{1}{2-\sqrt{3}}$ のはずなのに値がちがっていることに注目しよう．これは計算機では小数が, 浮動小数点数という近似的な数で表現されるからである．詳しくは 11 章を見よ．

この例では, 値が異なるのにどちらも同じ値 (3.73205) と表示されている．これは, 小数第 5 位まで表示するよう初期設定されているからである．これを変更するには

ctrl("real_digit", 桁数)

を使えばよい．

```
[0]     ctrl("real_digit",16);          RETURN
16
[1]     A=2+deval(3^(1/2));             RETURN
3.732050807568877
[2]     B=1/(2-deval(3^(1/2)));         RETURN
3.732050807568876
```

より, 確かに小数第 15 位で値が異なっていることがわかる．ただし, 16 桁より多く指定しても意味がない．これについても詳しくは 11 章を見よ．ちなみに, 155302.42+157.05; と入力すると 155459 が戻るがこれも小数点以下が表示されていないだけで, やはり real_digit を調整すれば表示される．なお, デフォールトでは表示形式が小数点形式と指数表示で自動的に切り替わるが, ctrl("double_output",1) とすると指数表示への自動切替えがおきない．

さて Asir にはプログラム言語がくみこまれており, この言語の機能をもちいると複雑なことを実行できる．まず一番の基礎であるくりかえしおよび印刷の機能をためしてみよう．

1. for (K=初期値; K<=終りの値; K++) {ループの中で実行するコマンド}; はあることを何度も繰り返したい時に用いる．for ループと呼ばれる．“K<=N” は, “K ≤ N か” という意味である．似た表現に, “K>=N” があるが, これは “K ≥ N か” という意味である．= のない “K<N” は, “K < N か” という意味である．

2. ++K や K++ は K を 1 増やせという意味である． K = K+1 と書いてもよい．同じく, --K や K-- は K を 1 減らせという意味である．

3. ++K や K++ は K を 1 増やせという意味である． K = K+1 と書いてもよい．同じく, --K や K-- は K を 1 減らせという意味である．

4. print( *expr* ) は *expr* を出力する．

5. load("ファイル名"); コマンドで, ファイルを Risa/Asir にロード (読み込み), 実行できる．ファイルの最後の行には end$ を書いておくこと．asirgui では, ファイルメニューの “開く” でファイルをロードしてもよい．

**例題 1.9** [02] for による繰り返しを用いて $\sqrt{x}$ の数表をつくろう．

**入力例 1.1**

```
[0]   for (I=0; I<2; I = I+0.2) {   RETURN
          print(I,0); print(" : ",0);   RETURN
          print(deval(I^(1/2)));   RETURN
      }   RETURN
```

出力結果

```
0 : 0
0.2 : 0.447214
0.4 : 0.632456
0.6 : 0.774597
0.8 : 0.894427
1 : 1
1.2 : 1.09545
1.4 : 1.18322
1.6 : 1.26491
1.8 : 1.34164
2 : 1.41421
```

`print(A)` は変数 `A` の値を画面に表示する. `print(`文字列`)` は文字列を画面に表示する. `print(A,0)` は変数 `A` の値を画面に表示するが, 表示したあとの改行をしない. 空白も文字である. したがって, `A=10; print(A,0); print(A+1);` を実行すると, 1011 と表示されてしまう. `A=10; print(A,0); print(" ",0);print(A+1);` を実行すると, 10 11 と表示される.

なおこの例題では, 入力を間違えたら最初からやりなおすしかない. これは当然面倒くさい. したがって, 次の節で説明するように通常はこの入力をファイルに書いておいてロードして実行する.

ところで, この例では条件が `I` $< 2$ なのに `I` $= 2$ の場合が表示されている. 実際に Asir 上で実行してみるとこうなるが, 理由を知るには、浮動小数の計算機上での表現についての知識が必要である (11 章を参照). とりあえず,

> 整数や分数の計算は Asir 上で正確に実行されるが, 小数についてはそうでない.

と覚えておこう.

**例題 1.10** [02] `Unix` 版では `help("fctr");` と入力すると関数 `fctr` の使い方の説明が出る. このように `help(`関数名`);` で オンラインマニュアルを読むことが可能である. `flist()` と入力すると現在定義されている関数の一覧を見ることが可能である. Windows 版では, ヘルプメニューで, 各関数のオンラインマニュアルを読むことができる.

**入力例 1.2**

```
[0]   help("fctr");   RETURN
```

出力結果:

```
‘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]]
```

help コマンドを用いると, このように Asir の組み込み関数についての解説が表示される. 説明が専門家向けの記述もあるので, 用語については, 数学や数式処理関係の本を参照する必要な場合もある. たとえば, 無平方分解 とはなにか調べよ.

Asir のマニュアルは Web ページおよび asirgui のヘルプメニューでもみることが可能である.

参考:
Asir の中には起動時に必要な関数達をよみこんでしまうものもある. たとえば, OpenXM 版の `Risa/Asir` の場合次の起動メッセージを出力する. この Risa/Asir を OpenXM/Risa/Asir とよぶ.

```
bash-2.03$ asir
This is Risa/Asir, Version 20020802 (Kobe Distribution).
Copyright (C) 1994-2000, all rights reserved, FUJITSU LABORATORIES LIMITED.
Copyright 2000,2001, Risa/Asir committers, http://www.openxm.org/.
GC 6.1(alpha5) copyright 2001, H-J. Boehm, A. J. Demers, Xerox, SGI, HP.
PARI 2.2.1(alpha), copyright (C) 2000,
     C. Batut, K. Belabas, D. Bernardi, H. Cohen and M. Olivier.
OpenXM/Risa/Asir-Contrib(20020808), Copyright 2000-2002, OpenXM.org
help("keyword"); ox_help(0); ox_help("keyword"); ox_grep("keyword");
     for help messages (unix version only).
Loading ~/.asirrc
[873]
```

この例では, 最初のプロンプトは `[873]` であるが, この数は起動時によみこまれる Asir プログラムのサイズにより変化する. OpenXM/Risa/Asir の場合, Asir Contrib ライブラリを起動時に読み込むので, この数が 873 になっている. Asir Contrib ライブラリには, 行列計算, TeX への変換などさまざまな便利な関数が用意されている.

現在の Asir 配布形態は下のように 2 通りある. どの版でも, version number が 20001200 以上のものならこの本のサンプルプログラムはすべて動作する.

1. Head branch オリジナル版
2. Head branch OpenXM 版 (OpenXM/Risa/Asir, オリジナル + Asir-Contrib パッケージ )

## 1.4 Asirgui (Windows 版)

Windows 版は, シェルからではなく, `asirgui` のアイコン

をダブルクリックして立ち上げる. 立ち上がったウィンドウはキーボードからの入力を受け付ける状態にあるが, 打った文字がすべてそのまま Asir に渡されるわけではなく, 入力行編集に用いられる特殊な文字もある.

1. [→] または [CTRL]+[f] : カーソルを一文字右に動かす.
2. [←] または [CTRL]+[b] : カーソルを一文字左に動かす.
3. [↑] または [CTRL]+[p] : 一つ前の入力行を表示する.

4. ↓ または CTRL+n ：現在表示されている入力行の次の入力行を表示する.
5. CTRL+a ：カーソルを行頭に動かす.
6. CTRL+e ：カーソルを行末に動かす.
7. CTRL+d ：カーソルの上の一文字をけす.
8. CTRL+k ：カーソルより行末までの文字を消す.
9. !+ 文字列 : 文字列 で始まる入力行のうち, 最新のものを表示する.
10. !! : 直前の入力行を表示する.

カーソル移動コマンドによりカーソルの位置を動かし, 文字削除コマンドで文字を削除したあと, 通常の文字を入力することで, 打ち間違いの場合にも, 全部を打ち直すことなく修正できる. また, 入力された行は保存されていて, これをヒストリと呼ぶ. CTRL+p , CTRL+n , ! により表示された行に対し, さらに編集作業を続けることができる. これにより, 引数を少し変えて同じコマンドを実行する, などが少ない手間で繰り返せる.

## 1.5　エディタを利用してプログラムを書く (unix 編)

入力をファイルに書いておいて, `load` して実行することができる. 2 章で説明するエディタ emacs および Risa/Asir を同時に起動しておく (図 1.1 を見よ). このように二つソフトを立ち上げておいた状態で, 下の 1, 2, 3 を繰り返すのが一般的なプログラムの開発方法である.

1. エディタ emacs でファイルを編集,
2. emacs で変更したら, 変更内容をファイルにセーブ.
3. Risa/Asir 側で `load` して使う.

このような方法は C や Java のプログラムを書く場合や, TeX で文書作成する場合など, いろいろな場面で有効である.

エディタ emacs のつかいかたについては 2 章で解説しているので, 適宜参照するとよい. Emacs は長い歴史をもつ強力なスクリーンエディタである. Emacs Lisp と呼ばれるマクロ言語をもち, エディタの機能を変更していくことが可能である. さらに emacs のなかから メールを読み書きしたり, プログラムのコンパイル, デバッグをすることも可能であり, 多くのプログラマに愛用されてきた. なお emacs と双璧をなしてプログラマに人気のあるエディタとして `vi` もある. シンプルでプログラミングにとり大事な機能しかもたない軽快な `vi` 愛用者も多い.

とにもかくにも, Emacs を覚えておくといろいろな場面で便利なのでここでは emacs を使って説明しよう.

**入力例 1.3** エディタ emacs を立ち上げる. `emacs hoge.rr &` と入力するとファイル `hoge.rr` を編集できる. なお, すでに emacs が起動していたら, ファイル `hoge.rr` をコマンド CTRL+x CTRL+f `hoge.rr` で開くと (open すると), ファイル `hoge.rr` を編集できる.

```
bash-2.03$    emacs  hoge.rr &    RETURN
```

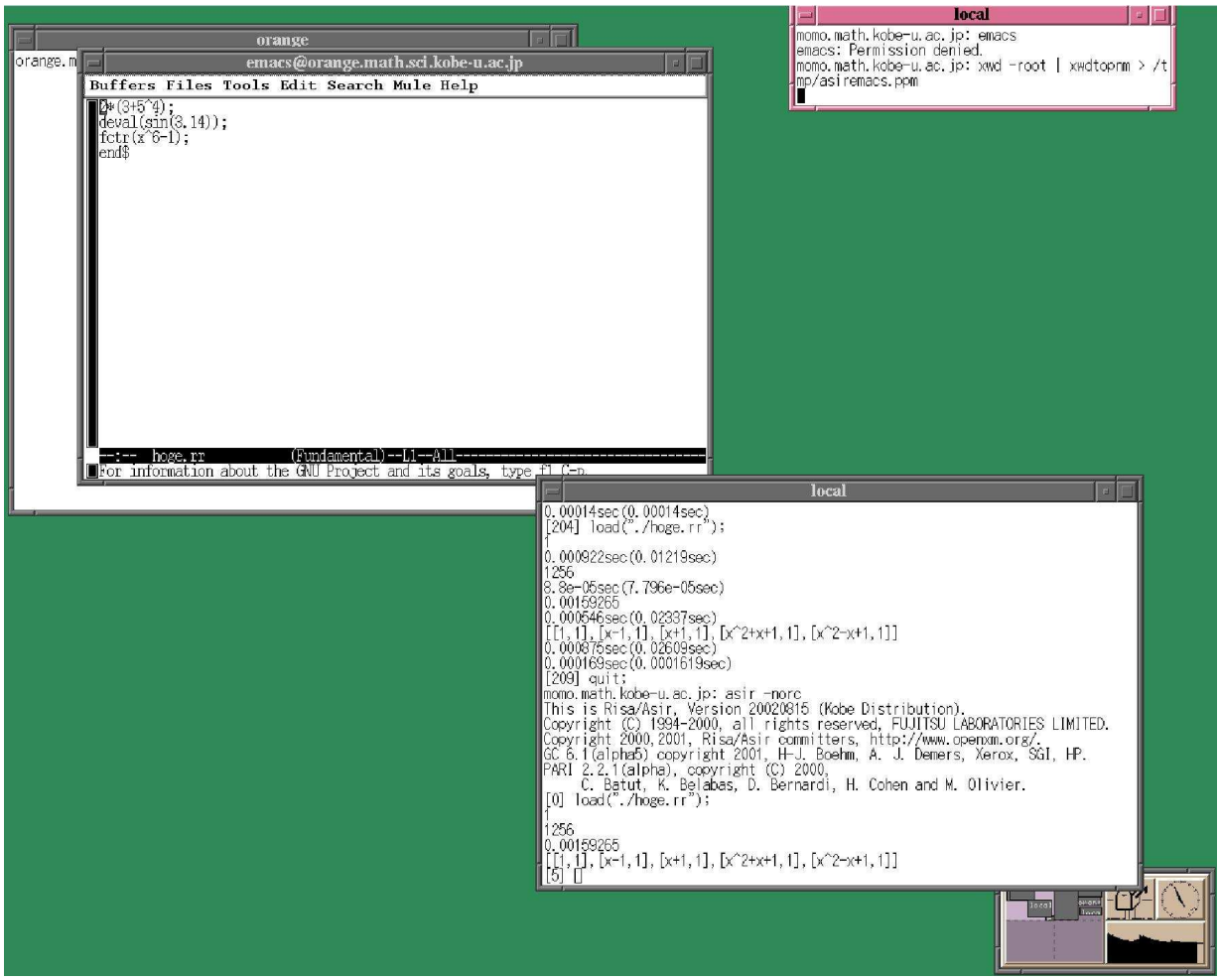



図 1.1: Asir と emacs

Emacs で次を入力する.

```
2*(3+5^4);
deval(sin(3.14));
fctr(x^6-1);
end$
```

入力がおわったら, コマンド CTRL+x CTRL+s でファイルをセーブ. 次に Asir の方で, load して読み込み実行.

```
bash-2.03$      asir      RETURN
                Asir の起動メッセージ
[0]             load("./hoge.rr");      RETURN
```

出力結果:

```
1
1256
0.00159265
[[1,1],[x-1,1],[x+1,1],[x^2+x+1,1],[x^2-x+1,1]]
[4]
```

出力の最初の 1 は load が成功しましたよ, といってる 1 である.

さて, emacs を立ち上げるとき & をつけて起動していることに気がついたであろうか?

UNIX は, もともと複数の人が同時に一つのコンピュータを使えるように設計された OS である. 「同時」といっても, CPU が一つしかない場合には, ある瞬間に実行されているのは一つのプログラムだが, UNIX は, 一つのプログラム (プロセス) に非常に短い時間 (例えば 10 ミリ秒) しか割り当てず, 時間が来ると強制的にそのプロセスを停止させ, 他のプロセスをスタートさせる. これを繰り返すことにより, あたかも多くのプロセスが同時に動いているように見えるのである. 言い換えれば UNIX は, アクティブなプロセスを待ち行列として管理している.

さて, 端末エミュレータも一つのプロセスであり, その中でユーザからの入力を待っているシェルも一つのプロセスである. シェルは他のプログラムを実行するのが仕事であるが,

*bash-2.03$* emacs RETURN

とすると, emacs は起動するものの, シェルのプロンプトは表示されない. 何人かの人が経験したように, この状態で端末エミュレータに文字を入力しても, 文字がオウム返しされるだけで何も起こらない. 実はこの状態ではシェルは emacs の終了を待っている. これを, 「emacs をフォアグラウンドで実行している」という. これに対し,

*bash-2.03$* emacs & RETURN

と実行すると,

```
[1] 60582 &
```

というようなメッセージを出してすぐにプロンプトが出る. これは, 呼び出したシェルと呼び出された emacs 双方がアクティブなプロセスとなっていることを意味する. この場合 「emcas をバックグラウンドで実行している」という.

フォアグラウンドからバックグラウンドへの移行も可能である. 何かのプロセスをフォアグラウンド実行しているシェルに, CTRL+z を入力すると, 実行されているプロセスが一旦停止し, シェルのプロンプトが出る. このとき bg を実行するとプロセスはバックグラウンドに移行する.

CTRL+z
*[1]+ Stopped emacs*
*bash-2.03$* bg RETURN
*[1]+ Stopped emacs &*
*bash-2.03$*

## 1.6 エディタを利用してプログラムを書く (Windows 編)

Windows でプログラムを書く場合も何らかのエディタが必要である. ここでは, Windows 付属のメモ帳 (notepad) を使うことにする. メモ帳以外でも書いたプログラムをテキストファイルとして保存できるエディタならなんでもよい. たとえば, Windows 用の emacs を利用してもよい. 日本語版の windows 用 emacs を meadow とよぶ.

**問題 1.5** `http://www.google.com` で meadow を配布しているホームページを検索して, meadow をダウンロード, インストールせよ.

メモ帳は, 通常は

スタート → (すべての) プログラム → アクセサリ

から起動できる. (`asirgui` から `shell("notepad");` と入力しても起動可能.) メモ帳を起動したら, プログラムを入力して,

ファイル → 名前を付けて保存

を選ぶ. すると保存する場所と名前を聞いてくるので, 適当な名前をつけて保存ボタンを押せばよい.

unix 上で, ファイルをロードして (読み込んで) 実行するには `load` コマンドを実行する必要があった. Windows の場合, `load` コマンドでもファイルをロード可能だが, `load` コマンドにファイルの正しい場所を示す引数を与えるのが面倒なので, `asirgui` の

ファイル → 開く

メニューから, ファイルを指定するのが楽である.

**例題 1.11** `Asir` のコマンドをいくつかファイルにメモ帳で入力し, セーブの後, ロードして実行しなさい. (図 1.2).

**問題 1.6** Windows 版では `asirgui` のファイルメニューの中の log を選択すると, Asir への入力および出力をファイルに保存できる. 本章の例を `Asir` で実行し, log をとることにより実行結果をファイルに保存しなさい.

## 1.7 参考: Asir について知っておくと便利なこと

### 1.7.1 起動時のファイル自動読み込み

Unix 版では `.asirrc` に

```
load("glib")$
end$
```

と書いておくと, 起動時に自動的にパッケージ `glib` がロードされる.

Windows 版でも環境変数 `HOME` で指定したディレクトリの下のファイル `.asirrc` に `load("glib")$` と書いておくと, 起動時に自動的にファイル `glib` を読み込むが, Asir メインディレクトリ (`get_rootdir();` で返されるディレクトリ) にファイル `.asirrc` をおいておくのが簡単である.

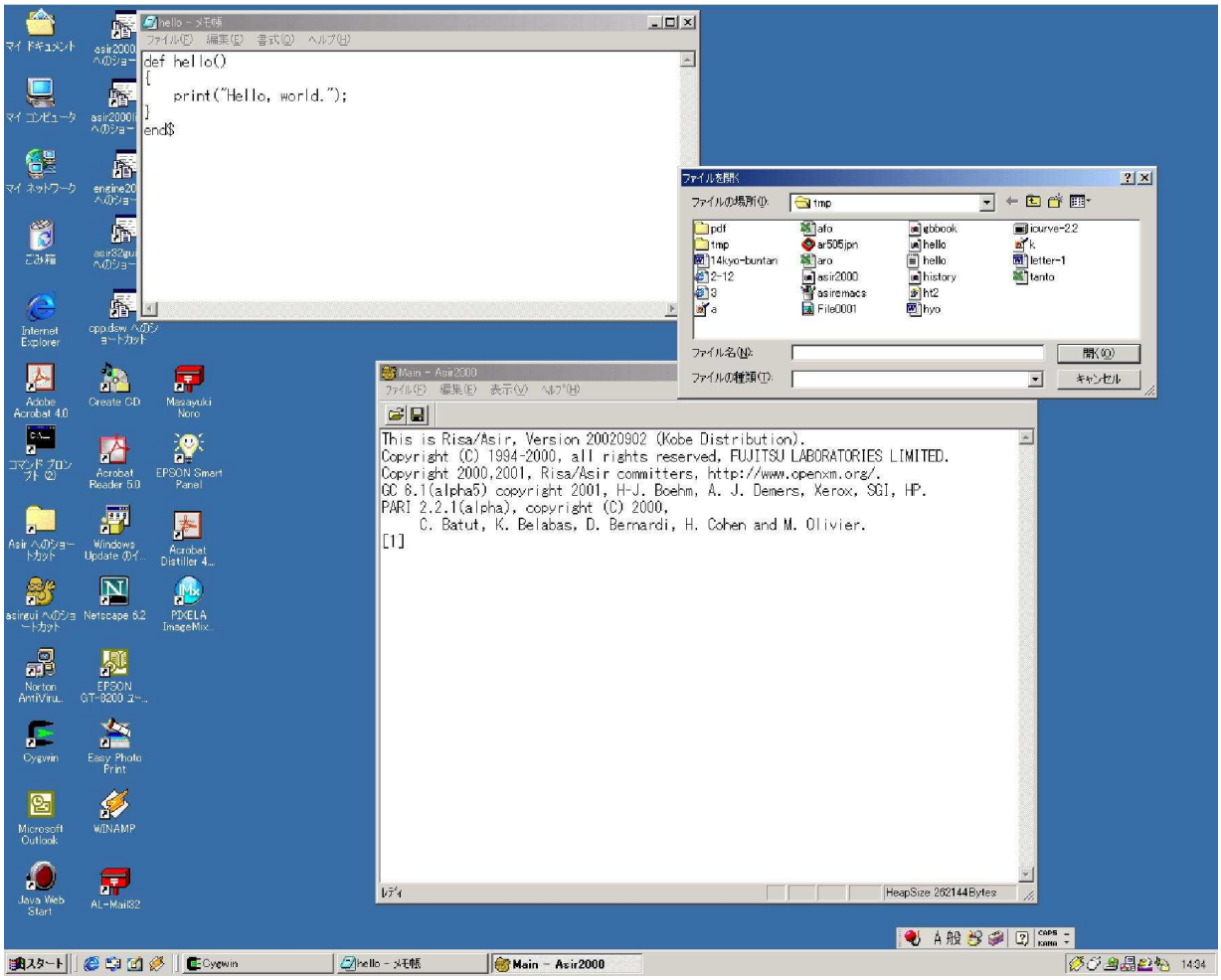



図 1.2: Asir とメモ帳

### 1.7.2 デバッガ

プログラムが文法的に正しくとも動くとは限らない. 実行時にエラーすると (debug) というプロンプトを出して止まる. これをうまく使えればプログラミングがとても楽になるが, とりあえずは `quit` で抜けられる.

### 1.7.3 ファイル読み込みディレクトリの登録 (unix 編)

ホームディレクトリにファイルを作り続けるとディレクトリが汚くなる. この場合, 例えばホームディレクトリに `risa` というディレクトリを作る.

| *bash-2.03$* | `mkdir risa` | RETURN |
|---|---|---|

次に bash ユーザの場合は `.bashrc` の最後に次を追加しておく.

```
ASIRLOADPATH=.:$HOME/risa
export ASIRLOADPATH
```

csh, tcsh ユーザの場合には

```
setenv ASIRLOADPATH .:$HOME/risa
```

を .cshrc に追加しておく. 自分がシェルとして bash をつかっているのか, csh, tcsh をつかっているのかわからない場合は両方のファイルに書いておけば良い. 新しい端末エミュレータを起動して, Asir ファイルを次のようにディレクトリ risa の中に作る.

```
bash-2.03$    cd risa    RETURN
bash-2.03$    emacs  test    RETURN
```

risa ディレクトリに入っているファイルは, カレントディレクトリがどこでも (asir を起動したディレクトリがどこであっても), 次のようにファイル名だけでロードできる.

```
bash-2.03$    cd /tmp    RETURN
bash-2.03$    asir    RETURN

[0]           load("test");    RETURN
1
```

ASIRLOADPATH が上のように設定されてない場合, Asir はカレントディレクトリでロードするファイルを探してくれない. カレントディレクトリのファイルを読む場合は, ./ と明示的にカレントディレクトリのファイルだと Risa/Asir に教える必要がある. これがコマンド load("./hoge.rr")$ の ./ の意味である.

## 1.8 章末の問題

1. 電源の on, off について注意すべき点はなにか?
2. OS にはどんな種類があるか?
3. お店ではどのようなコンピュータが売っているか?
4. Shift キー, Ctrl キーの役割はなにか?
5. エディタでファイルを編集, Asir の load コマンドでファイルをロードして実行. このとき編集したファイルはどこにセーブされるのか? セーブをわすれて load すると何がおこるか?
6. Asir book 版 CD または
   ftp://ftp.math.kobe-u.ac.jp/pub/OpenXM/asir-book/asir-book-ja.tgz には Prog ディレクトリがありそのなかに本書で説明しているプログラムが格納されている. このなかのたとえば, ccurve.rr をロードして, main(8); を実行してみよう (第 12 章を参照).

Risa/Asir ドリル ギャラリー : Happy Hacking Keyboard (PFU HHK Lite)

# 第2章　unix のシェルとエディターemacs

コンピューターが利用者（ユーザー）とやりとりをするには2通りのやり方がある．一つが graphical user interface でありもうひとつが文字ベースの user interface である．核となるシステムがありそれを覆っているのが user interface である．Windows も Unix も graphical user interface と文字ベースの user interface 両方を持っている．Unix での graphical user interface は X window システムとよばれている．

文字ベースの user interface は習得に少々時間がかかるが, いったん覚えると便利でやめられないし効率的である場合もおおい．文字ベースの user interface の代表的なものが シェルである．シェル言語は極めて論理的な代物なので, 論理的思考に強い人にはきわめてわかりやすくかつ使いやすい文字ベースのインターフェースである．

unix を始めて用いる読者は, この章は一度目は読みながすだけにして先を読みながら必要に応じて読みかえすといいであろう．また, この章も初歩しか書いていないので, 必要におうじて [1] や各種参考書などを参照するとよい．またディレクトリの木構造については, 付録 24 章も参照してほしい．

## 2.1　unix のシェルコマンド

1. 入力要求記号 (プロンプト) はコンピュータが何か入力を求めている時に表示される．入力要求記号は文字ベースのユーザインターフェースで良く使われる．入力要求記号は動いているソフトウエアの種類によってことなる．たとえば unix のシェルは `%` とか `$` とか `bash$` などを入力要求記号として使っている．Asir は `[数字]` を入力要求記号として使っている．MS-DOS のシェル (command.com) は　`a:¥`　とかを入力要求記号として使っている．以下で unix シェルに対するコマンドを説明する．

2. `man` コマンドはマニュアルを表示する．たとえば `man ls` と入力すると, `ls` コマンドの解説がみれる．

3. 新しいシェルウインドーを立ちあげたい時は、 `kterm &`　と入力する．ウインドーが開き、unix シェルのプロンプトがでる．

4. `ssh` ホストの名前：そのホストへリモートログインする．`ssh` は通信や login パスワードの暗号化を行ない通信する．

5. `asir` RETURN：Asir を起動する．`quit;` RETURN：Asir を終了する．

6. `w`：いま自分の計算機上にだれが login しているかをみるコマンド．

7. `ls`：カレントディレクトリのファイルの一覧を表示する．`ls -l`：カレントディレクトリのファイルの一覧を詳しく表示する．`ls -tl`：カレントディレクトリのファイルの一覧を詳しく変更時刻順に表示する．`ls -l | more` とすれば一ページずつ表示できる．スペースキーの入力で次のページ, `q` (quit) の入力で表示を修了する．`ls -a`：`.` ではじまる隠されたファイルも表示する．

8. Unix のパスは / で区切って書く. なお, Windows の日本語モードではパスの区切りは ¥ 記号である. Windows の英語モードではパスの区切りは　\　(バックスラッシュ) 記号である. ファイル名とパス名については, 第 24 章も参照せよ.

9. mkdir ディレクトリ名 : 新しいディレクトリを作る. cd ディレクトリ名 : そのディレクトリにカレントディレクトリを移す. cd .. : 一つ上のディレクトリへ移る. pwd : カレントディレクトリ (現在いるディレクトリ) を表示する.

10. chmod : ファイルの属性をかえる. chmod -w　ファイル名 : ファイルを書き込み禁止にする. chmod 600　ファイル名 : ファイルを自分だけしか読み書きできないようにする.

11. more

12. cp　ファイル名 1　　ファイル名 2 : ファイルのコピーをする.

13. rm　ファイル名 : ファイルを消す.

14. ps -ax : 現在動いているプロセスをすべて表示する. kill -9 プロセス番号 : プロセスを殺す.

15. CTRL+C : プログラムを stop する.

16. CTRL+Z : プログラムを中断する. kill %% : いま中断したプログラムをころす. fg : いま中断したプログラムを続行する (forground にもってくる).

17. ping マシン名 : マシン名で指定されるマシンがネットワークに接続されていて稼働中か調べる.

18. telnet マシン名 : マシン名で指定されるマシンにリモートログインする. 現在はセキュリティの確保のため, telnet の使用はすすめられない. かわりに ssh を使用する.

19. ftp : ethernet でつながった機械の間でのファイル転送.

20. passwd または yppasswd : パスワードの変更.

21. last | more : 最後に login した時刻を順に表示.

**例題 2.1**　現在のディレクトリにあるファイルをすべて表示しなさい. 新しいもの順に表示しなさい.

**入力例 2.1**

| | |
|---|---|
| *bash-2.03*　ls　RETURN | ファイルの一覧を表示. |
| *hoge.rr memo.txt* | |
| *bash-2.03*　ls -lt　RETURN | ファイルの一覧を変更時間順に表示. |
| *-rw-r–r– 1 taka taka 317 Nov 20 21:41 memo.txt* | |
| *-rw-r–r– 1 taka taka 16 Nov 20 21:39 hoge.rr* | |

**例題 2.2**　Unix ではシステムの全体の設定に関するファイルはディレクトリ /etc の下に格納されている. /etc の下で mo で始まるすべてのファイル名を表示せよ. ファイル /etc/motd の内容を表示せよ.

**入力例 2.2**

*bash-2.03* `ls /etc/mo*` RETURN | /etc/mo* で始まるファイルの一覧を表示.
*/etc/modules /etc/modules.conf /etc/motd*
*/etc/modutils:*
*0keep aliases arch om-inst paths setserial*
*bash-2.03* `more /etc/motd` RETURN | ファイル /etc/motd の内容を表示.
*Linux potato-610 ...*
*Most of the programs included with the Debian GNU/Linux system are*
*...*
*Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable law.*
これは起動時に表示されるメッセージである.
このファイルを編集すると起動時のメッセージを変更できる (root のみ)

**例題 2.3** Unix ではコマンドは, ディレクトリ`/bin`, `/usr/bin`, `/sbin`, `/usr/local/bin` などの下に格納されている. `/bin` の下で `r` で始まるすべてのファイル名を表示せよ.

**入力例 2.3**

*bash-2.03* `ls /bin/r*` RETURN | /bin/r で始まるファイルの一覧を表示.
*/bin/rbash /bin/readlink /bin/rm /bin/rmdir /bin/run-parts*

**問題 2.1** (参考課題)
ファイル `/etc/hosts`, `/etc/passwd` に何が書いてあるか `more` を用いて調べなさい. Unix の本を参照して意味を調べてみよう.

**問題 2.2** (Debian GNU Linux のみ, 上級課題)
ファイル `/etc/network/interfaces`, `/etc/init.d/networking` に何が書いてあるか調べなさい.

**問題 2.3** (FreeBSD のみ, 上級課題)
ファイル `/etc/rc.conf` に何が書いてあるか調べなさい.

**例題 2.4** 他のコンピュータが自分のコンピュータからインターネットを介してつながっているかしらべるにはコマンド `ping` を用いる. ホスト `www.math.kobe-u.ac.jp` が到達可能か `ping` で調べよ.

**入力例 2.4**

*bash-2.03* `ping www.math.kobe-u.ac.jp` RETURN | ping スタート
*PING apple.math.kobe-u.ac.jp (133.30.64.174): 56 data bytes*
*64 bytes from 133.30.64.174: icmp_seq=0 ttl=254 time=1.9 ms*
*64 bytes from 133.30.64.174: icmp_seq=0 ttl=254 time=1.4 ms*
CTRL + C | ping 中断
*bash-2.03* `ping 10.1.123.1` RETURN | 10.1.123.1 への ping
*PING 10.1.123.1 (10.1.123.1): 56 data bytes*
返事がない. つながっていない.

## 2.2 Emacs

1. `emacs` ファイル名 : emacs の起動. Emacs の仲間には, mule, xemacs などがある.
   注意: Emacs は, ほとんどの unix で, 標準でインストールされているかまたは簡単なコマンドでインストール可能である. Windows 用の mule は meadow と呼ばれる.

2. CTRL+x CTRL+f ファイル名 RETURN : ファイルの読み込み.

3. CTRL+x CTRL+c : emacs の終了.

4. CTRL+x CTRL+s : save. ファイルをセーブする.

5. 以下がカーソルの移動コマンド. CTRL+f : Forward, CTRL+b : Backword, CTRL+p : Previous, CTRL+n : Next. → ← ↑ ↓ キーでもカーソルは移動可能である.

6. CTRL+d : delete. カーソルの上の一文字をけす. CTRL+k : カーソルより行末までの文字を消す.

7. CTRL+g : 中断. これはとても便利.

8. CTRL+j : 一行したへ移動する. 言語に応じたインデントを自動的にしてくれる.

9. CTRL+SPACE : マークをつける. esc w : write to yank buffer. バッファにかく (copy). CTRL+y : yank. バッファの中身をカーソルのいちに書く (paste). CTRL+w : マークをつけた位置からカーソルまでの領域をバッファへ移す (copy).

10. CTRL+- : Undo. つまり、” 待った！ やりなおし”.

11. CTRL+x 2 : Window を二つにする. CTRL+x 1 : Window を一つにする. CTRL+x o : other window. 他のウインドーへ移る.

12. esc x shell RETURN : emacs のなかからシェルを立ちあげる.

13. esc x goto-line RETURN 行番号 : “行番号” へジャンプする. asir はプログラムの誤りがある場合その周辺の行番号を知らせる. たとえば `near line 11: parse error` の場合は 11 行目の周辺でエラーをさがす.

注意: `asirgui` の入力には Emacs 風のコマンドヒストリ機能がついている. たとえば, CTRL+f , CTRL+b でカーソルを移動して, 入力コマンドを編集できる. また CTRL+p (previous) で過去実行した `Asir` コマンド入力をよびだせる. Unix 版の Asir では, `fep asir` で Asir を起動すると, 同じようなコマンドヒストリ機能が利用可能である.

**例題 2.5** [02]　1 から 100 までの和を求めるプログラムを `s1.rr` という名前で emacs を用いて作りなさい. そしてこれを `Asir` へ読み込んで実行しなさい.

`s1.rr`

```
Re = 0$
for (K=1; K<=100; K++) {
  Re = Re + K;
}
print(Re)$
```

**入力例 2.5**

| | | |
|---|---|---|
| *bash-2.03* | `emacs s1.rr &` RETURN | s1.rr という名前でファイルを編集, Save (終了). |
| *bash-2.03* | `asir` RETURN | Asir を起動. |
| *This is Risa/Asir, Version ...* | | |
| ... | | |
| *[0]* | `load("./s1.rr");` RETURN | ファイルの s1.rr の読み込み. |
| ... | | |
| *5050* | | print(Re) が答を表示した. |

いままでの例題では RETURN 記号の入力も明示しておいたが, 以後の章では書かない. またシステムの出力を斜体, 入力をタイプライタフォントで示したが, 以後そうしない. 混乱はないものと思う.

**問題 2.4** (参考課題)
多くの unix システムでは TEX がインストールされている. emacs と TEX を用いて次の文書を作成せよ.

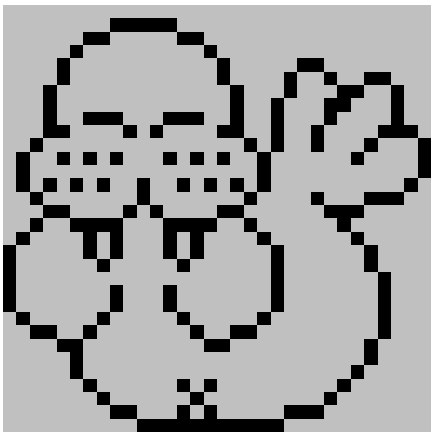
$a > 0$, $b > 0$ のとき, 次の不等式を証明せよ.

$$\frac{a+b}{2} \geq \sqrt{ab}$$

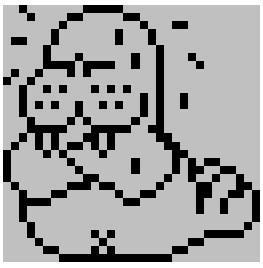
また等号が成り立つのはどのようなときか.

TEX への入力は以下のとおり.

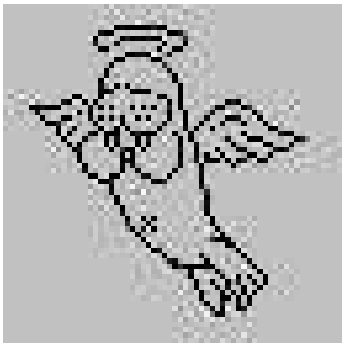
```
\documentclass{jarticle}
\begin{document}
$a >0$, $b>0$ のとき, 次の不等式を証明せよ.
$$  \frac{a+b}{2} \geq \sqrt{ab} $$
また等号が成り立つのはどのようなときか.
\end{document}
```



Risa/Asir ドリル ギャラリー：Risa/Asir アイコンの ホポ君の拡大図.



Risa/Asir ドリル ギャラリー：Risa/Asir アイコンの ホポ君バリエーション 1.



Risa/Asir ドリル ギャラリー：Risa/Asir アイコンの ホポ君バリエーション 2.

# 関連図書

[1] Kernighan, B.W., Pike, R. UNIX プログラミング環境. 1984. 日本語訳はアスキー出版局が出版.

Unix の考え方, Unix シェルを理解するには, 必読の名著であろう. Unix の誕生は 1969 年といわれている. Unix はベル研で誕生, 成長し, 多くの大学でさらに研究, 強化された. 1983 年にはカリフォルニア大学バークレー校から 4.2 BSD がリリースされた. これは TCP/IP の実装を含んでおり, 現在のインターネットでつかわれている多くの基本的なアイディアはここが原点であるといってもいいすぎではない. また MIT では 1980 年代なかばより, X Window System が研究された. 1990 年代にはいり, Linux や FreeBSD などのいわゆるパソコン用 (1984 年頃に初めて市場にでてきた 80386 以上の CPU と AT 互換機を対象), かつ配布に著作権上の制限がほとんどない unix が出現して, 多くの人にとりみじかなオペレーティングシステムとなった. 筆者の学科でもネットワークの基幹はこのようないわゆる PC unix を用いている.

この本の著者は Unix の設計者であり, 解説書の体裁でかかれてはいるが, この本から得るものはそれ以上のものであろう.

[2] Unix や Emacs (mule) についての手軽な参考書がいろいろある. これらについては, 本屋さんで自分の気にいったものを読めばよいであろう.

# 第3章　計算機の仕組み

Risa/Asir の本格的なプログラミングにはいる前に計算機の仕組みを簡単に学んでおこう. 現実のプログラミングはつねに計算機の構造にしばられている. したがって, 計算機の構造をある程度納得しておくことはプログラミングにとっても大事である. この章の内容をきちんと理解するのは簡単ではないので, とりあえずはこんなもんかと納得しておくことにして, 機会をみつけてこの章の内容を深く理解しようと努力してほしい. なお, 講義では TK80 エミュレータをプロジェクタでみせながら, 解説をした.

## 3.1　CPU, RAM, DISK

計算機は CPU, RAM, DISK, 周辺 LSI, 入出力装置とそれらを結ぶ各種通信路 ( バス 等) からなる. これらの話題を詳細にのべることはこの本の目的ではないが, プログラムを書くうえで, 計算機の仕組みについてそれなりのイメージをもっているのは大事である.

図 3.1 は単純化した, 計算機の仕組みの図である. 計算機は メモリ (図 3.3 の写真), CPU (central processing unit, 図 3.2 の写真も参照), および入出力機器をもつ.

1. CPU はメモリよりプログラムを読み込み逐次実行する. CPU は四則演算や条件判断をしてジャンプ等をおこなう.

2. すべての情報は 0 と 1 で表現される. コンピュータで画像や音楽を扱うことも可能であるが, これらも, 0 と 1 だけで (2 進数で) 表現されている.

3. 計算機は四則演算, “正か, 負か, 0 か, でない” の条件判断, “かつ, または” の論理演算に帰着するような処理しかできない.

メモリには番地が付いており, 一つの番地には 2 進数 8 桁分を記憶できる. 2 進数 8 桁分を 1 byte と呼ぶ. 2 進数の 11111111 は 10 進数の 255 なので, メモリの一つの番地には, 0 から 255 までの数を格納できる.

CPU はメモリよりプログラムを読み込み逐次実行する. たとえば, 1 から 100 までの和を求める次のプログラムを考えよう.

```
Re = 0$
for (K=1; K<=100; K++) {
  Re = Re + K;
}
print(Re)$
```

メモリには, 変数 Re, K に対応する適当な大きさの領域が確保される. for ループにはいるとまず, K に 1 が格納される. 次に条件 K<=100 が成り立つか CPU が判断して for ループを抜けるかどうか

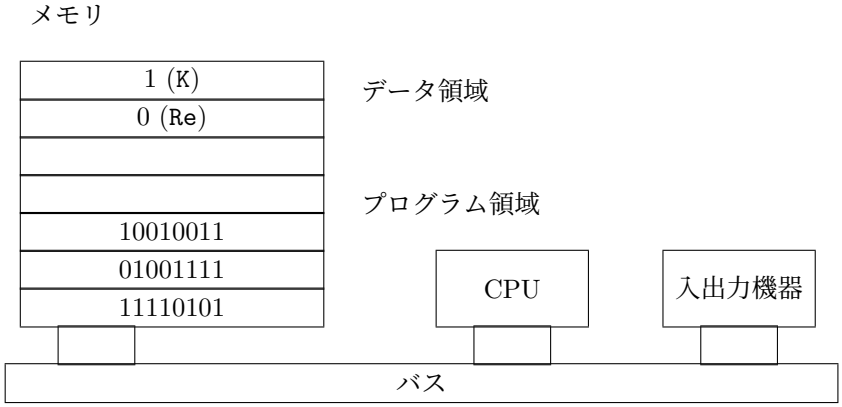



図 3.1: メモリ, CPU

決めている. この場合はループを抜ける必要はないので, メモリより, Re と K の値を読み込み, Re＋K を計算し, メモリの領域 Re へ計算結果の値を格納する. K の値を 1 増やしてから, 次に条件 K<=100 が成り立つか CPU が判断して for ループをぬけるかどうか決めている. この場合はループを抜ける必要はないので, メモリより, Re と K の値を読み込み, Re＋K を計算し, メモリの領域 Re へ計算結果の値を格納する. K の値を 1 増やす. 以下これを繰り返す. 最後にループを抜けたら, Re の値を印刷する.

ただしこの説明はすこし読者をだましている. CPU は Asir のプログラムを理解できるわけではない. CPU が理解できるのは, Asir よりとっつきにくい機械語である. すこしとっつきにくいが, 上の Asir とおなじように, 代入, 比較, 足し算などの機能をもつ.

機械語 (マシン語) は 1 と 0 の列である. これが順に解釈実行される. 機械語は CPU の種類によりことなる. Intel Pentium CPU とか, R3000 CPU とか PowerPC CPU とかいろんな CPU がある. 現在いわゆる IBM PC コンパチ機 (互換機) で利用されているのは, Intel 80386 系 CPU である. この系統の CPU は長い歴史をもっており, 以下のように発展してきた.

1. アドレス空間の大きさが $2^{16}$ の 8080 CPU.
2. アドレス空間の大きさが $2^{20}$ の 8086 CPU.
3. Intel Pentium CPU の直接の祖先であるアドレス空間の大きさが $2^{32}$ の 80386 CPU.

アドレス空間というのは, メモリの番地のとりえる値の範囲である. たとえばアドレス空間が $2^{16}$ のときは, 番地は 0 から $2^{16}-1$ までである.

Intel Pentium の起源ともいえる, Intel 8080 CPU は各種シミュレータが存在しており, 現在でも 8080 CPU のマシン語プログラミングを経験することが可能である.

マシン語は Risa/Asir とことなり, 2 進数の列で表現する. たとえば, TK-80 エミュレータ [1] に載っている Intel 8080 CPU では, マシン語命令 1100 1010 は, 一つ前の演算結果が 0 なら この命令の次にくる 16 桁の 2 進数をメモリの番地とみなして, その番地にジャンプせよという意味となる. このような数字の列をメモリにプログラムとして書き込んでいくことにより TK-80 をプログラミングする. TK-80 を実際実行するところを, マシン語に堪能な人にたのんで見せて見せてもらうとよいであろう. 全ての計算機は最終的にはこのマシン語を解釈実行している.

計算機内部では全てが 2 進数の数字で表現されている. 図 3.3 (メモリ) を良く見ると, 金色に光る縦の線が沢山見えるであろう. メモリが動作中は, 金色に光る縦の線ひとつひとつが 2 進数の 1 桁に対応することとなる. この部分にかかる電圧を目にみることができれば, 2 進数で現在どのアドレスにアクセスしているのか? どのようなデータを読んでいるのかを知る事が可能である. 第 10 章末の Risa/Asir ドリル ギャラリー のハードディスクの写真にも, 何本もの線がたばになったケーブルが写っているであろう. この線一本一本がハードディスクに転送されるデータやコマンドの 2 進数表現の 1 桁 1 桁に対応している.

## 3.2 計算機の歴史

計算機の歴史を述べるのは, 本書の趣旨でないが, Asir の実習をするうえで, 知っておくと助けとなる予備知識としてすこしだけ説明する.

パーソナルコンピュータの歴史は, いわゆる Apple 系のコンピュータ (Macintosh など) と, Intel の 80 系 CPU を搭載した, IBM PC コンパチ機 および NEC PC98 系などに大別できる.

ここでは, IBM PC コンパチ機について簡単に歴史を述べるにとどめる. IBM PC コンパチ機出現以前には, Intel 8080 (1973 年頃) またはそれの上位互換 CPU である Z80 (1975 年頃) を搭載したいろいろなコンピュータがあった. 日本では 1980 年代始めに NEC 8001, NEC 8801 シリーズが多く売れた. これらのコンピュータは Microsoft Basic を標準搭載する他に CP/M 80 というディスクオペレーティングシステム (基本ソフト) が稼働可能であった. パーソナルコンピュータ市場に進出を決めた, IBM は, 1981 年 Intel 8088 CPU を搭載した IBM PC を市場に 出した. さらにその仕様を公開したため, 多くのメーカーが全く同じ仕様のパーソナルコンピュータを売り出すこととなった. これが現在に続く IBM PC 互換機の誕生である. IBM は当初 CP/M 86 をディスクオペレーティングとして, 採用したかったが, CP/M を販売していた, デジタルリサーチ社とおりあわず, 結局マイクロソフトから MSDOS の供給をうけ PCDOS なる名前で売り出すこととなった. これが Windows の元祖 MSDOS である. MSDOS はこのような歴史的事情もあり, CP/M 80 に似ている部分も多い. たとえば MSDOS や Windows では, たとえば `command.com` みたいなファイル名を多く用いている. このファイル名ではドットの後ろの, 3 文字がファイルの形式をあらわす. たとえば, `.com` や `.exe` は実行可能形式のファイル, `.htm` は HTML ファイル, `.c` は C 言語のファイルである. この 3 文字を拡張子と呼ぶ. MSDOS ではファイル名を 8 文字 ドット 3 文字であらわす. これがいわゆる 8.3 形式である. この 8.3 形式は CP/M 80 に由来する. 現在でも, コマンドプロンプトを起動すれば, この MSDOS のコマンドをいろいろ使うことが可能であり, 筆者 (T) など愛用者もいる.

さて, Asir は MSDOS の末裔である Windows でも動作するが, unix オペレーティングシステム上で開発されている. unix オペレーティングシステムの歴史は省略するが, 1990 年代なかばより, Linux や FreeBSD などのいわゆるフリーの unix が IBM PC 互換機や Macintosh 上で快適に動作する. このような unix は, 無料で配布されており, またオペレーティングシステム全部のソースが公開されている. Asir はこのようなフリーの unix のなかの FreeBSD で開発されている. フリーの unix は, インターネットからや雑誌の付録として容易に取得可能である.

## 3.3 2 進数と 16 進数

計算機の扱える情報は 2 進数である. それらにたいする, 四則演算と記憶ですべての処理がすすむ. これを常に念頭において勉強していくのが, 理解の早道であろう.

2 進数で 8 桁の情報を 1 byte とよぶ. メモリは各番地に 1 byte の情報を格納できる. 2 進数は 0

と 1 の 2 つの記号を用いて数を表現する. 2 進数と 10 進数の対応は以下のとおり.

| | |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |

$a_n a_{n-1} \cdots a_0$, $a_i \in \{0, 1\}$ なる表示の 2 進数を 10 進数であらわすと,

$$\sum_{k=0}^{n} a_k 2^k$$

に等しい. たとえば, 2 進数 1101 は, $2^3 + 2^2 + 2^0$ に等しい.

2 進数を書くのは桁が多くて面倒なので, 代わりに普通 16 進数を使うことが多い. 16 進数 2 桁が 2 進数 8 桁に対応するので換算が簡単である. 16 進数では

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

と 16 個の記号を用いて数をあらわす. A が 10 進数の 10, B が 10 進数の 11, C が 10 進数の 12, D が 10 進数の 13, E が 10 進数の 14, F が 10 進数の 15 に対応する. したがって, 2 進数, 16 進数, 10 進数の対応は以下のようになる.

| | | |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | $A$ | 10 |
| 1011 | $B$ | 11 |
| 1100 | $C$ | 12 |
| 1101 | $D$ | 13 |
| 1110 | $E$ | 14 |
| 1111 | $F$ | 15 |

| | | |
|---|---|---|
| 10000 | 10 | 16 |
| 10001 | 11 | 17 |
| 10010 | 12 | 18 |
| 10011 | 13 | 19 |
| 10100 | 14 | 20 |
| 10101 | 15 | 21 |
| 10110 | 16 | 22 |
| 10111 | 17 | 23 |
| 11000 | 18 | 24 |
| 11001 | 19 | 25 |
| 11010 | 1$A$ | 26 |
| 11011 | 1$B$ | 27 |
| 11100 | 1$C$ | 28 |
| 11101 | 1$D$ | 29 |
| 11110 | 1$E$ | 30 |
| 11111 | 1$F$ | 31 |

16 進数は `0x` や H をつけて表すことも多い. また A から F を小文字で書くことも多い. たとえば `0x1E`, `0x1e`, 1EH, 1eH はすべて 16 進数の 1E を表す. $a_n a_{n-1} \cdots a_0$, $a_i \in \{0, 1, \ldots, F\}$ なる表示の 16 進数を 10 進数であらわすと,

$$\sum_{k=0}^{n} a_k 16^k$$

に等しい. ただし $a_i$ が 0xA のときは 10, $a_i$ が 0xB のときは 11, 等と解釈する. 2 進数と 16 進数の間の変換は容易であろう.

**問題 3.1** 次の 16 進数達を 10 進数, 2 進数で書け. 0xFF, 0x100, 0xFFFF, 0x10000, 0x81.

**問題 3.2** 16 進数で筆算をやることを試みよ.

計算機が数を有限桁の 2 進数で表現していることを実感する例として次の計算をやってみるのはおもしろいであろう. コマンド `pari(sqrt,n)` を用いると, `n` の平方根を指定した精度で計算できる. 詳しくは, 索引およびマニュアルの `eval` および `setprec` を見よ.

**例題 3.1** $2+\sqrt{3}$ と $\frac{1}{2-\sqrt{3}}$ の近似値を Asir で計算してみなさい.

```
[346] 2+pari(sqrt,3);
3.732050807568877293465305078 34
[347] 1/(2-pari(sqrt,3));
3.732050807568877292661930199 21
```

$\frac{1}{2-\sqrt{3}}$ の分母を有理化すると $2+\sqrt{3}$ に等しいはずだが, 近似的な数値計算ではこのように誤差が生じる.

Asir では教育用にメモリーを直接覗いたり操作したりする, peek と poke という関数がついている. これらの関数を用いるとメモリに数が格納されている様子を観察できる. これらの関数については 第 11 章を見よ.

## 3.4 章末の問題

1. 1 から 100 までの数を足すプログラムを参考にして, 1 から 100 までの範囲にある偶数を足すプログラムに変更しなさい.

2. 0x100, 0x200, 0x300, 0x400 をそれぞれ 10 進数になおしなさい.

3. (Windows のみ)
   スタート → プログラム → アクセサリ → コマンドプロンプト (MSDOS プロンプト)
   で コマンドプロンプトを立ち上げ `debug` と入力してみよう. debug が立ち上がる. たとえば `d 100` と入力すると 100H 番地からのメモリの内容を 16 進数でみれる. 詳しくは参考書 [2] を見てもらいたい.

```
[878] setprec(5000);
2003
[879] eval(@pi);
3.14159265358979323846264338327950288419716939937510582097494459230781640628620
99862803482534211706798214808651328230664709384460955058223172535940812848111745
02841027019385211055596446229489549303819644288109756659334461284756482337867831
65271201909145648566923460348610454326648213393607260249141273724587006606315588
17488152092096282925409171536436789259036001133053054882046652138414695194151160
94330572703657595919530921861173819326117931051185480744623799627495673518857527
24891227938183011949129833673362440656643086021394946395224737190702179860943702
77053921717629317675238467481846766940513200056812714526356082778577134275778960
91736371787214684409012249534301465495853710507922796892589235420199561121290219
60864034418159813629774771309960518707211349999998372978049951059731732816096318
59502445945534690830264252230825334468503526193118817101000313783875288658753320
83814206171776691473035982534904287554687311595628638823537875937519577818577805
32171226806613001927876611195909216420198938095257201065485863278865936153381827
96823030195203530185296899577362259941389124972177528347913151557485724245415069
59508295331168617278558890750983817546374649393192550604009277016711390098488240
12858361603563707660104710181942955596198946767837449448255379774726847104047534
64620804668425906949129331367702898915210475216205696602405803815019351125338243
00355876402474964732639141992726042699227967823547816360093417216412199245863150
30286182974555706749838505494588586926995690927210797509302955321165344987202755
96023648066549911988183479775356636980742654252786255181841757467289097777279380
00816470600161452491921732172147723501414419735685481613611573525521334757418494
68438523323907394143334547762416862518983569485562099219222184272550254256887671
79049460165346680498862723279178608578438382796797668145410095388378636095068006
42251252051173929848960841284886269456042419652850222106611863067442786220391949
45047123713786960956364371917287467764657573962413890865832645995813390478027590
09946576407895126946839835259570982582262052248940772671947826848260147699090264
01363944374553056820349625245174939965143142980919065925093722169646151570985 83
87410597885959772975498930161753928468138268683868942774155991855925245953959431
04997252468084598727364469584865383673622262609912460805124388439045124413654976
27807977156914359977001296160894416948685558484063534220722258284886481584560285
06016842739452267467678895252138522549954666727823986456596116354886230577456498
03559363456817432411251507606947945109659609402522887971089314566913686722874894
05601015033086179286809208747609178249385890097149096759852613655497818931297848
21682998948722658804857564014270477555132379641451523746234364542858444795265867
82105114135473573952311342716610213596953623144295248493718711014576540359027993
44037420073105785390621983874478084784896833214457138687519435064302184531910484
81005370614680674919278191197939952061419663428754440643745123718192179998391015
91956181467514269123974894090718649423196156794520809514655022523160388193014209
37621378559566389377870830390697920773467221825625996615014215030680384477345492
02605414665925201497442850732518666002132434088190710486331734649651453905796268
56100550810665879699816357473638405257145910289706414011097120628043903975951567
71577004203378699360072305587631763594218731251471205329281918261861258673215791
98414848829164470609575270695722091756711672291098169091528017350671274858322287
18352093539657251210835791513698820914442100675103346711031412671113699086585163
98315019701651511685171437657618351556508849099898599823873455283316355076479185
35893226185489632132933089857064204675259070915481416549859461637180270981994309
92448895757128289059232332609729971208443357326548938239119325974636673058360414
28138830320382490375898524374417029132765618093773444030707469211201913020330380
19762110110044929321516084244485963766983895228684783123552658213144957685726243
34418930396864262434107732269780280731891544110104468232527162010526522721116603
96665573092547110557853763466820653109896526918620564769312570586356620185581007
29360659876486117910453348850346113657686753249441668039626579787718556084552965
41266540853061434443185867697514566140680070023787765913440171274947042056223053
89945613140711270004078547332699390814545664645880797270826683063432858778569830 52
35808933065757406795457163775254202114955761581400250126228594130216471550979259
23099079654737612551765675135751782966645477917450112996148903046399471329621073
40437518957359614589019389713111790429782856475032031986915140287080859904801094
12147221317947647772622414254854540332157185306142288137585043063321751829798662
23717215916077166925474873898665494945011465406284336639379003976926567214638530
67360965712091807638327166416274888800786925602902284721040317211860820419000422
96617119637792133757511495950156604963186294726547364252308177036751590673502350
72835405670403867435136222247715891504953098444893330963408780769325993978054193
41447377441842631298608099888687413260472156951623
[880]
```

Risa/Asir ドリル ギャラリー：$\pi$ の近似値.

# 関連図書

[1] 榊 正憲, 復活! TK-80. アスキー出版局, 2000.
TK-80 の Windows 用エミュレータが付いている. このエミュレータが動くのをみると機械語ってどんなものというイメージがつかめる. 8080 CPU や 8086 CPU の機械語の理解をするには TK-80 のプログラミングとか, Windows の MSDOS プロンプトで debug を用いて簡単なプログラムをしてみるのをおすすめする (著者 (T)).

[2] 蒲地輝尚, はじめて読む 8086, アスキー出版局, 1987.
debug を用いた 8086 マシン語プログラミングにはこの本がお薦めである. 8086 のマシン語でプログラムを書いても, Intel Pentium は上位互換, かつ Windows は MSDOS に上位互換なのでそのまま動作するのが嬉しい (著者 (T)). ちなみにこの本は PC9801 向けにかかれているので, 6.9 の音を出す例題は, AT 互換機では動作しない. AT 互換機では

```
MOV AL,3
OUT 61,AL
```

で音がでる. 音を止めるには, `AL` レジスタに `0` をいれて同じことをやればよい. Windows 2000 や XP では, ハードウエアを直接操作するのは許されていないので, 残念ながら音はでない.

[3] 高山信毅, CP/M 80 の世界, 工学社, 1988.
パーソナルコンピュータの歴史のひとこま, CP/M 80 上のソフトについて詳しい.

[4] IBM PC の歴史については,

`http://www.atmarkit.co.jp/fsys/pcencyclopedia/004pc_history02/pc_hist03.html`

を見よ.


図 3.2: CPU (Pentium III)

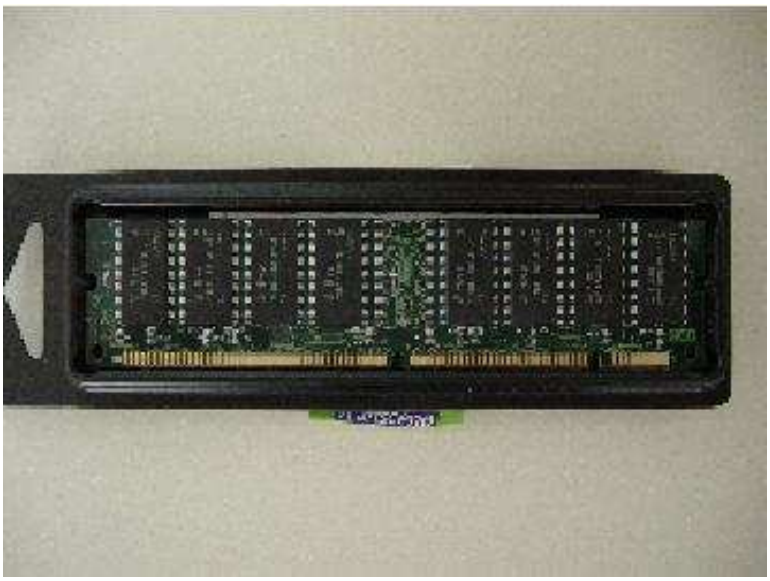


図 3.3: メモリー (256MB DIMM)

# 第4章　Risa/Asir 入門

## 4.1　Risa/Asir で書く短いプログラム

本書のもととなった講義では, 高校数学の教科書で紹介されている BASIC のプログラミングをひととおり勉強したあと, この本の内容にはいっていくことも多かった. BASIC はいろいろ批判もあるが, (1) 行番号, (2) 代入文や命令文, (3) goto 文, をもち, マシン語を勉強しなくても “計算機とはなにか” というイメージをある程度持つための訓練には最適の言語の一つであろう. Risa/Asir のユーザ言語は, C に似た文法をもつより抽象化された言語であるが, それが実際の計算機でどのように実行されるのか常に想像しながら使用するのと, そうでないのとは, 大きな違いがある. 達人への道を歩もうという人は, 抽象化された言語で書かれていようが, 実際の計算機でどのように実行されるのかをわかっていないといけない. プログラム実行中のメモリ使用の様子を手にとるように 描写できるようになれば, 計算機の達人への道は間違いなしである. 前節で計算機の仕組みの概略と 2 進数について説明したのはつねにこの “プログラム実行中のメモリ使用の様子を手にとるように描写できるように” ということを念頭において計算機のプログラミングをしてほしいからである.

訳のわからない説教で始めてしまったが, この意味はだんだんと分かるであろう. とにかく Risa/Asir で短いプログラムを書いてみよう.

くりかえしは以下のように for 文を用いる.

**例題 4.1** [02]

```
for (K=1; K<=5; K=K+1) { print(K); };
```

を実行してみなさい. このプログラムは `print(K)` を `K` の値を 1 から 5 まで変えながら 5 回実行する.

**入出力例 4.1**

```
[347]  for (K=1; K<=5; K=K+1) { print(K); };
1
2
3
4
5
[348] 0
[349]
```

**例題 4.2** [02]　1 から 100 までの数の和を求めるプログラム

```
S = 0;
for (K=1; K<=100; K++) {
   S = S+K;
}
print(S);
```

または

```
def main() {
  S = 0;
  for (K=1; K<=100; K++) {
      S = S+K;
  }
  print(S);
}
main();
```

を実行してみなさい. これらの内容を書いてあるファイル a.rr を作成して, load("./a.rr"); でロード実行するとよい (unix の場合). Windows の場合は, プログラムをファイルから読み込むには “ファイル → 開く” を用いる.

同じプログラムを BASIC で書くと以下のようになる.

```
10 S = 0
20 for K=1 to 100
30   S = S+K
40 next K
50 print S
```

条件分岐をするときには if 文を用いる. 次の例は, $b, c$ に数をセットすると 2 次方程式 $x^2+bx+c=0$ の解の近似値を求める. なお @i は虚数単位 $\sqrt{-1}$ である.

```
B = 1.0; C=3.0;
D = B^2-4*C;
if (D >= 0) {
   DQ = deval(D^(1/2));
   print([ -B/2+DQ/2, -B/2-DQ/2]);
}else {
   DQ = deval((-D)^(1/2));
   print([ -B/2+@i*DQ/2, -B/2-@i*DQ/2]);
}
```

条件分岐をするためにもちいる表現をまとめておく.

| 記号 | 意味 | 例 |
|---|---|---|
| == | ひとしいか? | X == 1 ( X = 1 か?) |
| != | ひとしくないか? | X != 1 ( X ≠ 1 か?) |
| ! | でない (否定) | !(X==1) ( X ≠ 1 か?) |
| && | かつ | (X < 1) && (X > -2) ( X < 1 かつ X > −2 か?) |
| \|\| | または | (X > 1) \|\| (X < -2) ( X > 1 または X < −2 か?) |

念のために, “かつ” と “または” の定義を復習しておく. 次の表では, T で 真 (True) を, F で 偽 (False) を表す.

| A | B | A && B |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

| A | B | A \|\| B |
|---|---|---|
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

なお Asir では 偽を 0, 真を 0 でない数で表現してよいし, 偽を false, 真を true と書いてもよい. たとえば,

```
if (1) {
   print("hello");
}else{
   print("bye");
}
```

は hello を出力する.

条件判断をかっこでくくって合成してもよい. たとえば

```
(1 || 0) && 1
```

は 1 すなわち真 (T) となる.

**問題 4.1** 次のプログラムで hello は何回表示されるか?

```
for (I=0; I<10; I++) {
   if (((I^2-6) > 0) && ( (I < 3) || ( I > 6))) {
      print("hello");
   }
}
```

**例題 4.3** 1 から 20 までの自然数 $N$ についてその 2 乗の逆数 $\frac{1}{N^2}$ の和を求めるプログラムを書き, 実際に計算機でどのように実行されているのかメモリと CPU の様子を中心として説明しなさい. メモリと CPU については, 第 3.1 節を見よ.
プログラムは以下のとおり. 参考のため BASIC のプログラムも掲載, 解説する.

| 場所 | 内容 |
|---|---|
| S | 0 |
| N | 1 |
| | |
| | |
| | |

図 4.1: メモリの図解

Asir 版

```
S=0;
for (N=1; N<=20; N++) {
  S = S+1/(N*N);
}
print(S);
```

BASIC 版

```
10 S = 0
20 for N=1 to 20
30   S = S+1/(N*N)
40 next N
50 print N
```

Asir は 17299975731542641/10838475198270720 なる答えを戻す. これを, 小数による近似値になおすにはたとえば函数 deval(S) (S の 64bit 浮動小数点数への変換) または eval(S*exp(0)) (S の任意精度浮動小数点数への変換) を用いる. 一方 BASIC (たとえば 16bit 版 UBASIC) は 1.5961632439130233163 を戻す. この違いは Asir は分数ですべてを計算しているのに対して, basic では 1/(N*N) を近似小数になおして計算していることによる.
“実際に計算機でどのように実行されているのかメモリと CPU の様子を中心として説明” するときに大事なポイントは以下のとおり.

1. プログラムもメモリに格納されており, それを CPU が順に読み出して実行している. 注意: この場合は正確にいえばインタプリタがその作業をしている. インタプリタはプログラムを順に読みだし, CPU が実行可能な形式に変換して実行している. (初心者にはこの違いはすこしむづかしいかも.)

2. S と N という名前のついたメモリの領域が確保され, その内容がプログラムの実行にともない時々刻々と変化している.

説明: 文 S=0 の実行で, メモリの S という名前のついた領域に数 0 (ゼロ) が格納される. (図 4.1 を見よ.)
for 文 の始めで, メモリの N という名前のついた領域に数 1 が格納される.
文 S = S+1/(N*N) では 1/(N*N) が CPU で計算されてその結果と S の現在の値 0 が CPU で加算されメモリ S に格納され S の値は更新される.
以下, 省略.

## 4.2 デバッガ

実行中エラーを起こした場合 Asir はデバッグモードにはいり, プロンプトが

(debug)

に変わる. デバッグモードから抜けるには quit を入力すればよい.

```
(debug) quit
```

実行中エラーを起こした場合, 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. 変数の値は

```
(debug) print 変数名
```

で表示できる.

list コマンドを用いるとエラーをおこしたあたりのプログラムを表示してくれるので便利である.

```
(debug) list
```

## 4.3 関数の定義

Risa/Asir ではいくつかの処理を関数としてひとまとめにすることができる. 関数は

```
def  関数名 () {
   関数の中でやる処理
}
```

なる形式で定義する. これは関数名にその手続きを登録しているだけで実際の実行をさせるには以下のように入力する必要がある.

```
関数名 ();
```

たとえば $b$, $c$ に数をセットすると 2 次方程式 $x^2 + bx + c = 0$ の解の近似値を求めるプログラムは次のようにして関数としてまとめるとよい. なおプログラム中で $b$, $c$ は変数 B, C に対応している. @i は虚数単位 $\sqrt{-1} = i$ である.

```
def quad() {
  B = 1.0; C=3.0;
  D = B^2-4*C;
  if (D >= 0) {
     DQ = deval(D^(1/2));
     print([ -B/2+DQ/2, -B/2-DQ/2]);
  }else {
     DQ = deval((-D)^(1/2));
     print([ -B/2+@i*DQ/2, -B/2-@i*DQ/2]);
  }
}
quad();
```

上のプログラムで 変数 B, C の値をいちいち変更するのは面倒である. そのような時には B, C を関数の引数とするとよい.

```
def quad2(B,C) {
  D = B^2-4*C;
  if (D >= 0) {
     DQ = deval(D^(1/2));
     print([ -B/2+DQ/2, -B/2-DQ/2]);
  }else {
     DQ = deval((-D)^(1/2));
     print([ -B/2+@i*DQ/2, -B/2-@i*DQ/2]);
  }
}
quad2(1.0,3.0);
quad2(2.0,5.0);
quad2(3.0,1.2);
```

B, C を関数 quad2 の引数 (argument) とよぶ. たとえば, quad2(1.0,3.0); を実行すると, 変数 B に 1.0, 変数 C に 3.0 が代入されて, 関数 quad2 が実行される. 上の例の場合には $b = 1.0$, $c = 3.0$ および $b = 2.0$, $c = 5.0$ および $b = 3.0$, $c = 1.2$ の 3 通りの 2 次方程式を解いていることになる.

プログラミングにはいろいろと格言があるが, その一つは,

> 困難は分割せよ.

である. 関数をもちいることにより, 処理をさまざまなグループに分解して見通しよく処理することが可能となる. 関数については, 第 8 章でくわしく説明する.

## 4.4 章末の問題

1. コマンドの末尾の `$` と `;` の働きの違いを説明しなさい.
2. 関数 deval の働きは何か? `print(deval(1/3+1/4))$` および `print(1/3+1/4)$` の出力を例として説明しなさい.
3. 与えられた数 $a, b, c$ に対して方程式 $ax^2 + bx + c = 0$ を解くプログラムを作りなさい.

# 関連図書

[1] 齋藤, 竹島, 平野: Risa/Asir ガイドブック SEG 出版, ISBN4-87243-076-X.
Risa/Asir の平易な入門書. Risa/Asir の開発の歴史についての記述もありおもしろい.

[2] 野呂: 計算代数入門, Rokko Lectures in Mathematics, 9, 2000. ISBN 4-907719-09-4.
`http://www.math.kobe-u.ac.jp/Asir/ca.pdf` から, PDF ファイルを取得できる.
`http://www.openxm.org` より openxm のソースコードをダウンロードすると, ディレクトリ `OpenXM/doc/compalg` にこの本の TeX ソースがある.

[3] D.E. Knuth: The Art of Computer Programming, Vol2. Seminumerical Algorithms, 3rd ed. Addison-Wesley (1998). ISBN 0-201-89684-2.
日本語訳はサイエンス社から, “準数値算法” という書名で出版されている. 乱数, 浮動小数, (多倍長) 整数, 多項式 GCD, 因数分解などに関するアルゴリズムについて広範かつ詳細に書かれている. アルゴリズムだけでなくその実装法についても得るところが多い.

因数分解をするための Berlekamp のアルゴリズムについては, 他に藤原良, 神保雅一: 符号と暗号の数理 (共立) などを参考にするといいであろう.

# 第5章　制御構造

## 5.1　条件判断と繰り返し

プログラムは以下のような条件判断と繰り返しのための文を組み合わせておこなう. 具体的な使い方の例は次の節からを参照.

1. for（初期設定; 終了条件; ループ変数の更新）{ ループ内で実行するコマンド列 }.
2. if（条件）{ 条件が成立するときに実行するコマンド列 }
3. if（条件）{ 条件が成立するときに実行するコマンド列 }
   else { 条件が成立しないときに実行するコマンド列 }
4. while（条件）{ 条件が成立するときに実行するコマンド列 }
5. do { コマンド列 } while（条件）　条件がみたされる限り do 内のコマンド列を繰り返す.
6. break;　break は 繰り返しをするループのなかから飛び出すのに用いる. 次の節の例 5.5 を参照.

上で “コマンド列” が, 1 個のみのときは { } を略してよい. たとえば,

```
if (A) { print("yes!"); }    は
if (A)  print("yes!");      と書いてよい.
```

## 5.2　プログラム例

**例 5.1** まずは読み書きそろばんプログラムから.

ファイル名 cond1.rr

```
 1:  def main(A,B) {
 2:        print(A+B);
 3:        print("A kakeru B=",0);
 4:        print(A*B);
 5:  }
 6:  end$
```

左のプログラムはあたらえられた二つの数の和と積を出力するプログラムである. 関数の引数として数 A, B を読み込む. 2, 3 行目で和と積を計算して出力する. { と } かこんだものがひとかたまりの単位である. 実行は自前の関数 main() に数字をいれて評価すればよい. 字さげ (インデントという) をしてわかりやすく書いていることに注意.

実行例

```
[0] load("cond1.rr");
[1] main(43,900)$
943
A kakeru B=38700
```

なお, cond1.rr のロードが失敗する場合は, load("./cond1.rr"); と入力する (第 1.7.3 節). Windows では “ファイル → 開く” を用いて読み込む.

**例 5.2** 次に if をつかってみよう.

プログラム

```
/* cond2.rr */
def main(A,B) {
      if ( A > B ) {
         C=A;
      }else{
         C=B;
      }
      print(C);
}
end$
```

出力結果

```
[0] load("cond2.rr");
[1] = main(2,-54354)$
2
```

main(A,B) は A と B を比較して大きい方を印刷する. 次のように自前の関数の戻り値として, 大きい方を戻してもよい. 詳しくは, 8 章で説明するが, print と return は違う. print は画面に値を印刷するのに対して, return は関数の値を戻す働きを持つ. return の値が画面に印刷されるのは, 下で説明しているように ; の副作用である.

プログラム

```
/* cond2.rr */
def main(A,B) {
      if ( A > B ) {
         C=A;
      }else{
         C=B;
      }
      return(C);
}
end$
```

main のあとに ; (セミコロン) をつけて戻り値を印刷するようにしていることに注意. セミコロンの代わりに $ を用いると, 戻り値をプリントしない.
字下げ (インデント) の仕方にも気をつけよう. if や for を用いるときは, 読みやすいようにインデントをおこなうべきである. たとえば, 左のプログラムを

```
def main(A,B){if(A>
    B){C=A;}else{
C=B;}return(C);}
```

のように書いてもよいが読みにくい.

出力結果

```
[0] load("cond2.rr");
[1] = main(2,-54354);
2
```

/* と */ でかこまれた部分はコメントであり, プログラムとはみなされない. プログラム全体で利用される定数は, define 文で宣言しておくとよい. たとえば, #define AAA 10 と書いておく

と, 以下 AAA があらわれるとすべて 10 でおきかえられる. この置き換えは, プログラムのロード時におこなわれるので, AAA=10 とするより実行速度の点で有利である.

**例 5.3** 次に for を使って繰り返しをやってみよう.

プログラム

```
/* cond3.rr */
def main() {
      Result = 0;
      for (K = 1; K<= 10; K++) {
          Result = Result + K^2;
      }
      print(Result);
}
end$
```

左のプログラムは $\sum_{k=1}^{10} k^2$ を計算するプログラムである.

**問題 5.1** [05] $\sum_{k=1}^{n} k^2$ の表を $n = 1$ より 100 に対して作れ.

実行例

```
[0] load("cond3.rr");
[1] main()$
  385
```

**例 5.4** 上のプログラムでは, $\sum_{k=1}^{n} k^2$ の計算をいろいろな $n$ に対して計算するのにいちいちプログラムを書き換えないといけない. この問題点は関数の引数というものを使うと簡単に解決できる. 関数とその引数については 8 章で詳しく議論するが, この例に関しては下の例で十分了解可能であろう.

プログラム

```
/* cond3a.rr */
def main(N) {
      Result = 0;
      for (K = 1; K<= N; K++) {
          Result = Result + K^2;
      }
      print(Result);
}
end$
```

左のプログラムは $\sum_{k=1}^{\mathtt{N}} k^2$ を計算するプログラムである.
N の実際の値は, 例のように main の後に与えればよい.

実行例

```
[0] load("cond3a.rr");
[1] main(10)$
  385
[2] main(20)$
 2870
```

**例 5.5** つぎのプログラムは break による for ループからの脱出の例.

プログラム

```
def main() {
  for (X=-990; X<=990; X++) {
    if (X^2-89*X-990 == 0) {
       print(X);
       break;
    }
  }
}
main()$
end$
```

このプログラムは 2 次方程式 $x^2-89x-990=(x-99)(x+10)$ の整数解を一つしらみつぶし探索で探すプログラムである. つまり, $x$ の値を順番に変えて, 方程式を実際にみたすか調べている. 解 $-10$ が発見された時点で, break コマンドで for ループを抜け出してプログラムを終了する.

**例 5.6** つぎのプログラムは線のひきかたと・の打ち方の解説.

プログラム

```
/* cond4.m */
load("glib");
def main0() {
  glib_open();
  glib_window(0,0,1,1);
  glib_putpixel(0.8,0.5);
  glib_line(0,0,1,0.5);
  glib_line(0,0,1,1);
}
end$
```

繰り返しをつかってグラフィックスを書く前にちょっとトレーニングを. 左のプログラムは・をうって, 線分を二本か書くプログラムである. glib_window では, グラフィックを表示する座標系の設定をしている. 引数の 0,0,1,1 は 座標 (0,0) を画面の 左上に, 座標 (1,1) を画面の右下にせよという意味である. glib で始まる関数については, 本章末の 5.3 節の解説を参照.

**例 5.7** つぎに, グラフィック画面で for の働きをみてみよう.

```
/* cond5.rr */
load("glib")$
def main() {
  glib_open();
  glib_window(0,0,100,100);
  for (K=1; K<=100; K = K+8) {
     glib_line(0,0,70,K);
  }
}
end$
```

プログラムを実行するには, ロードしたあと, main(); と入力する. 左のプログラムは傾きが段々おおきくなっていく線分達を描く.
K = K+8 は K += 8 と書いた方が簡潔である.

**例 5.8** 次はグラフを何枚かつづけて書くプログラム.

```
/* cond7.rr */
load("glib")$
def main() {
  for (A=0.0; A<=2; A += 0.8) {
     plot(sin(5*x)+A*sin(2*x),
          [x,0,10*3.14]);
  }
}
end$
```

プログラムを実行するには, ロードしたあと, `main();` と入力する. このプログラムは

$$\sin 5x + a \sin 2x$$

のグラフを $a = 0, 0.8, 1.6$ について三枚描くプログラムである.

**例 5.9** Taylor 展開は関数を多項式で近似する方法である. $n$ 次の Taylor 展開のグラフを描くプログラムを書いて, Taylor 展開がもとの関数に収束していく様子を観察してみよう.

```
/* taylor.rr */
load("glib")$
def taylor(N) {
  glib_open();
  glib_window(-5,-2,5,2);
  glib_clear();
  F = 0;
  for (I=0; I<=N; I++) {
    F=F+
     (-1)^I*x^(2*I+1)/fac(2*I+1);
  }
  print("sin(x) の Taylor 展
開 :",0);
  print(2*N+1,0);
  print(" 次までは ");
  glib_line(-5,0,5,0);
  glib_line(0,-5,0,5);
  print(F);
  for (K=-5; K<=5; K = K+0.03) {
     glib_putpixel(K,subst(F,x,K));
  }
}
print("Type in, for example,
      taylor(2);taylor(4);")$
end$
```

`taylor(N);` と入力すると, このプログラムは

$$\sin x$$

のテイラー展開

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

を `2*N+1` 次まで計算して, グラフを描く. たとえば `taylor(4);` と入力してみよう. `subst(F,x,K)` は式 `F` の中の `x` を数 `K` で置き換えた結果を戻す.

**例 5.10** Fourier 展開は関数を三角関数で近似する方法である. $n$ 次の Fourier 展開のグラフを描くプログラムを書いて, Fourier 展開がもとの関数に収束していく様子を観察してみよう. Fourier 展開は, JPEG 画像の処理などに使われている.

```
load("glib")$
def fourier(N) {
  glib_open();
  glib_window(-5,-5,5,5);
  glib_clear();
  F = 0;
  for (I=1; I<=N; I++) {
     F=F+(-1)^(I+1)*sin(I*x)/I;
  }
  F = 2*F;
  print("x の Fourier 展
開 :",0);
  print(N,0);
  print(" 次までは ");
  glib_line(-5,0,5,0);
  glib_line(0,-5,0,5);
  glib_line(deval(-@pi),deval(-@pi),
            deval(-@pi),deval(@pi));
  glib_line(deval(@pi),deval(-@pi),
            deval(@pi),deval(@pi));
  print(F);
  for (K=-5; K<=5; K = K+0.03) {
     glib_putpixel(
      K,deval(subst(F,x,K)));
  }
}
print("Type in, for example,
       fourier(4); fourier(10);")$
end$
```

`fourier(N);` と入力すると, このプログラムは

$$x$$

の Fourier 展開

$$x = 2\sum_{n=1}^{\infty}(-1)^{n+1}\frac{\sin(nx)}{n}$$

を N 次まで計算して, グラフを描く. `deval(F)` は F を double の精度で (11 章を見よ), 数値計算する. `subst(F,x,K)` では `2*sin(0.5)-sin(1.0)` みたいな式に変形されるだけなので, `deval` による評価が必要である.

**例 5.11** 次に `for` の 2 重ループを作ってみよう.

プログラム

```
/* cond6.m */
def main() {
   for (I=1; I<=3; I++) {
      print("<<<");
      for(J=1;J<=2;J++) {
         print("I=",0);
         print(I);
         print("J=",0);
         print(J);
      }
      print("  >>>");
   }
}
end$
```

for の中に for を入れることもできる. 実行例をよくみて I, J の値がどう変わっていっているか見て欲しい.

実行例

```
main();
<<<
I=1
J=1
I=1
J=2
  >>>    つづきは右
```

```
<<<
I=2
J=1
I=2
J=2
  >>>
<<<
I=3
J=1
I=3
J=2
  >>>
```

**例 5.12** 次に不定方程式 $x^2 + y^2 = z^2$ の整数解を for を用いたしらみつぶし法で探してみよう.

プログラム

```
/* cond77.rr */
def main() {
  for (X=1; X<10; X++) {
    for (Y=1; Y<10; Y++) {
      for (Z=1; Z<10; Z++) {
          if (X^2+Y^2 == Z^2) {
             print([X,Y,Z]);
          }
       }
    }
  }
}
end$
```

実行例

```
[346] load("cond77.rr");
1
[349] main();
[3,4,5]
[4,3,5]
0
```

**問題 5.3**　引数 $N$ の階乗を返す関数を作れ.

**問題 5.4**　引数 $N, I$ に対し, 2 項係数 $\binom{N}{I}$ を返す関数を作れ. .

**問題 5.5**　引数 $N$ が素数ならば 1, 合成数なら 0 を返す関数を作れ. (整数 $I$ に対し, isqrt($I$) は $\sqrt{I}$ を越えない最大の整数を返す. )

**問題 5.6**

```
def main() {
  for (K=0; K<5; K++) {
    print(K);
  }
}
```

この関数を while を使って書き換えよ.

**問題 5.7**　for (K=1;K<=100;K++) { X=12; Y = X*X;}
と for (K=1;K<=100;K++) { X=12345678790123456789; Y = X*X;} の早さを cputime(1); を用いて比べなさい. 早さが違うときはその理由も考えなさい.

$1 \le x < 10, 1 \le y < 10, 1 \le z < 10$ の範囲で, 全部のくみあわせをしらみつぶしに調べてみることにより, 整数解を探そうというプログラムである.

**問題 5.2** [15]　もっと早く整数解を見つけられるようにプログラムを改良せよ. ちなみに, この方程式の解は理論的によくわかっているので, その結果を使うのは反則.

## 5.3 glib について

この節のプログラムの中で, グラフィック関連のコマンドは `load("glib");` コマンドをまず最初に実行しておかないと実行できないものがある.

```
[0]    load("glib");    RETURN
```

Windows 版 asirgui では, ファイルメニューの “開く” から, glib を選択して読み込んでもよい.

glib をロードすることにより, 次の関数が使えるようになる.

| | |
|---|---|
| glib_window(X0,Y0,X1,Y1) | 図を書く window のサイズを決める.<br>画面左上の座標が (X0,Y0), 画面右下の座標が (X1,Y1) であるような座標系で以下描画せよ.<br>ただし x 座標は, 右にいくに従いおおきくなり,<br>y 座標は 下に いくに従い大きくなる (図 5.1). |
| glib_clear() | 描画画面をクリアする. |
| glib_putpixel(X,Y) | 座標 (X,Y) に点を打つ. |
| glib_line(X,Y,P,Q) | 座標 (X,Y) から 座標 (P,Q) へ直線を引く |
| glib_print(X,Y,S) | 座標 (X,Y) に文字列 S を書く (英数字のみ). |


図 5.1: 座標系

色を変更したいときは, | 記号で区切ったオプショナル引数 color を使う. たとえば,

```
glib_line(0,0,100,100|color=0xff0000);
```

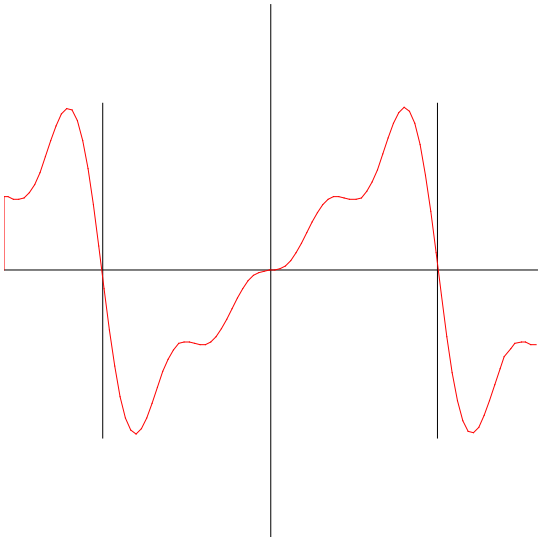
と入力すると, 色 0xff0000 で線分をひく. ここで, 色は RGB の各成分の強さを 2 桁の 16 進数で指定する. この例では, R 成分が ff なので, 赤の線をひくこととなる. なお, 関数 glib_putpixel も同じようにして, 色を指定できる.

さて, 図 5.1 で見たようにコンピュータプログラムの世界では, 画面の左上を原点にして, 下へいくに従い, $y$ 座標が増えるような座標系をとることが多い. 数学のグラフを書いたりするにはこれでは不便なことも多いので, glib では,

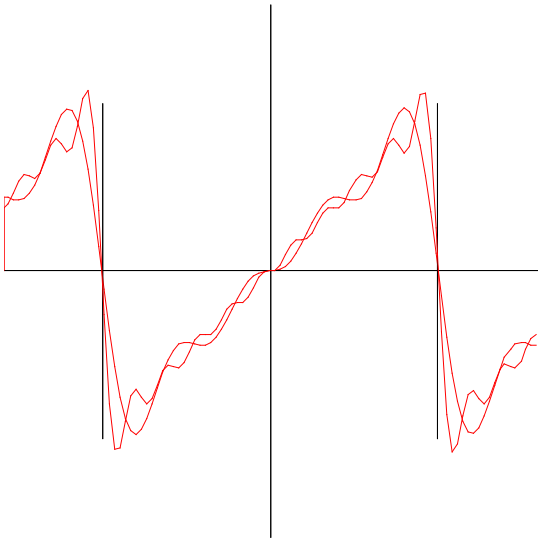
```
Glib_math_coordinate=1;
```

を実行しておくと画面の左下が原点で, 上にいくに従い $y$ 座標が増えるような数学での座標系で図を描画することが可能である.

次の図は, $f(x) = x$ の Fourier 展開を 4 次までとったものの数学座標系でのグラフを `glib` を用いて書いたものである.



次の図は, $f(x) = x$ の Fourier 展開を 4 次までとったものと 10 次までとったもののグラフを重ねて書いたものである. $x$ に収束していく様子を観察できる.



# 第6章　制御構造とやさしいアルゴリズム

## 6.1　2 分法とニュートン法

計算機では整数の四則計算の組み合わせで, より複雑な計算をしているとおもってほぼまちがいない. たとえば $\sqrt{a}$ の近似計算を考えてみよう. この数を近似計算するにはいろいろな方法があるが, 一つの方法は, $\sqrt{a}$ は $y = x^2 - a$ と $y = 0$ の交点をもとめることである. 一般に $f(x)$ を連続微分可能関数とし,

$$y = f(x)$$

と $y = 0$ の交点を近似的に求めることを考えてみよう.

点 $x = x_n$ における $y = f(x)$ の接線の方程式は,

$$y - f(x_n) = f'(x_n)(x - x_n)$$

である. したがって, この接線の方程式と $y = 0$ との交点は,

$$x_n - f(x_n)/f'(x_n)$$

となる. 数列 $x_k$ を次の漸化式できめよう.

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

いま, $f(x) = 0$ の根 $r$ に十分近い数 $x_0$ をとり, 上の漸化式で, 数列 $x_k$ を決めていけば, $f'$ が 0 でない限り, $x_k$ は $r$ に収束していくであろう. これは, 厳密に証明せずとも, 図 6.1 をみれば納得できるであろう. このように $f(x) = 0$ の根を求める方法を Newton 法とよぶ. Newton 法は, 出発点とする十分近い解を見付けることができれば, 非常に収束が早い. しかしながら, やや安定性に欠ける.

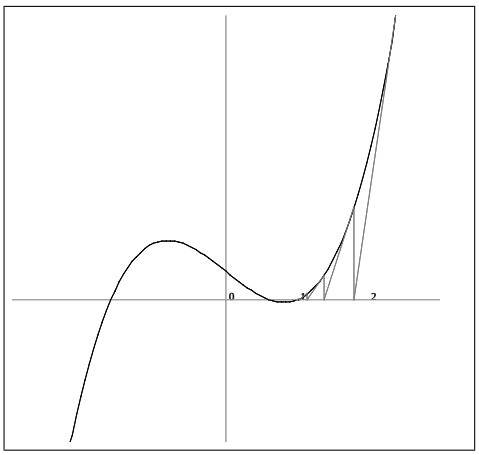



図 6.1: Newton 法が収束する例

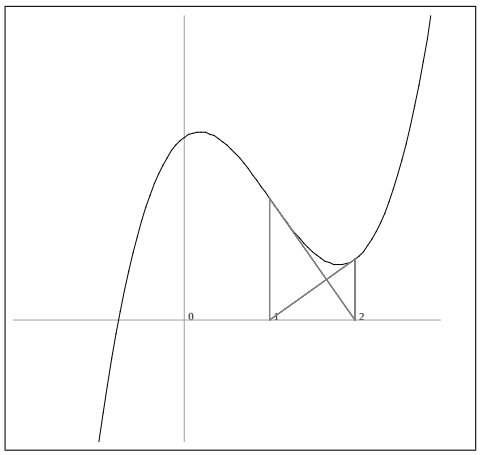



図 6.2: Newton 法が収束しない例

たとえば図 6.2 を見られたい. この図は, $f(x) = x^3 - 3x^2 + x + 3$ に対し, $x_1 = 1$ からスタートした Newton 法の挙動を示している. この場合, $x_n$ は 1 と 2 という値を繰り返しとることになる. これは極端な例だが, 極値の周辺で, 不安定な状況が起こることは容易に想像がつくであろう.

初期値さえ適切ならば, Newton 法は高速に $f(x) = 0$ の解を求めることができる. Newton 法で $\sqrt{x}$ の近似値を求めるプログラムは以下のとおり.

```
def sqrtByNewton(A) {
  Epsilon = 0.0001;
  P = 0.0;
  Q = deval(A);
  while (!( (Q-P > -Epsilon) &&
            (Q-P < Epsilon))) {
     P = Q;
     print(Q);
     Q = P-(P*P-A)/(2.0*P);
  }
  return(Q);
}
```

たとえば, $\sqrt{2}$ は次のように計算できる.

```
[422] sqrtByNewton(2);
2
1.5
1.41667
1.41422
1.41421
```

このプログラムでは P が $x_k$ だとすると,

```
Q = P-(P*P-A)/(2.0*P);
```

の実行で Q に $x_{k+1}$ の値がはいる. while ループでは, 次の繰り返しにはいると P=Q を実行するので P に $x_{k+1}$ が代入される. `Q = P-(P*P-A)/(2.0*P);` の実行で Q にこんどは $x_{k+2}$ の値がはいる. これを繰り返すことにより数列 $x_i$ を計算している.

while の条件の意味を説明しよう. このプログラムでは P が $x_k$ だとすると, Q には $x_{k+1}$ がはいっている. したがってこのプログラムで while ループをぬけだすための条件は $|x_{k+1} - x_k| <$ Epsilon である. 計算の精度をあげるためには, Epsilon の値を小さくすればよい.

Newton 法ほど早くないが, 安定性のある方法として, 2 分法 (bisection method) がある.

$f$ が連続関数とするとき, $f(a) < 0$ かつ $f(b) > 0$ なら $f(\alpha) = 0$ となる根 $\alpha$ が区間 $(a, b)$ に存在するという事実を思いだそう. この事実をもちいて, 根の存在範囲を狭めていくのが, 2 分法である.

次のプログラムでは, while ループの中において, 区間 [A,B] につねに解が存在することが保証されている. つまり, F(A) $< 0$, F(B) $> 0$ が常に成り立つように変数 A, B の値が更新されていく. 更新のやりかたは以下のとおり.

```
C = (A+B)/2;              A, B の中点の計算
FC = subst(F,x,C);        C における F(x) の値の計算
if (FC > 0) B = C;
else if (FC < 0) A = C;
```

```
def bisection(A,B,F) {
  Epsilon = 0.00001;
  A = deval(A); B=deval(B);
  if (subst(F,x,A) >= 0 || subst(F,x,B) <= 0)
    error("F(A)<0 and F(B)>0 must hold.");
  while (true) {
    C = (A+B)/2;
    if ((A-B > -Epsilon) && (A-B < Epsilon))
      return(C);
    FC = subst(F,x,C);
    if (FC > 0) B = C;
    else if (FC < 0) A = C;
    else if (FC > -Epsilon && FC < Epsilon)
      return(C);
  }
}
```

左のプログラムでは, 根の存在区間の幅が Epsilon よりも小さくなった時にループを抜けて, その区間の中点の値を返す. 実行すると次のようになる.

```
[205] bisection(1,2,x^2-2);
1.41421
```

**問題 6.1** (1) 上のプログラムには解説がほとんどない. プログラムの仕組みを解読して解説せよ. (2) 2 分法で根がもとまる様子を観察できるようにプログラムを改造せよ. Newton 法での収束の様子と比較せよ.

上の問題で, 収束の様子を調べたが, Newton 法でもとまった近似解にどの程度誤差があるかは, 次のようにして理論的に調べることが可能である.

根を $\alpha$, $x_n$ をニュートン法にあらわれる根の近似値の列, 誤差を

$$\varepsilon_n = x_n - \alpha$$

とおく.

以下, $\varepsilon_{n+1}$ は, $\frac{\varepsilon_n^2}{2}\frac{f''(\alpha)}{f'(\alpha)}$ にほぼ等しいということをテイラー展開を用いて証明する.

$x_n = \alpha + \varepsilon_n$ なので, テイラー展開の公式から, $\alpha < \xi_i < x_n$ を満たす定数 $\xi_i$ が存在して,

$$\begin{aligned} f(x_n) &= f(\alpha) + \varepsilon_n f'(\alpha) + \frac{\varepsilon_n^2}{2} f''(\alpha) + \frac{\varepsilon_n^3}{6} f'''(\xi_1) \\ f'(x_n) &= f'(\alpha) + \varepsilon_n f''(\alpha) + \frac{\varepsilon_n^2}{2} f'''(\xi_2) \end{aligned}$$

がなりたつ. 定義より,

$$\begin{aligned} \varepsilon_{n+1} &= x_{n+1} - \alpha \\ &= x_n - \frac{f(x_n)}{f'(x_n)} - \alpha \\ &= \varepsilon_n - \frac{f(x_n)}{f'(x_n)} \end{aligned}$$

がなりたつ. これにテイラー展開の式を代入すると,

$$\varepsilon_{n+1} = \varepsilon_n - \frac{\varepsilon_n f'(\alpha) + \frac{\varepsilon_n^2}{2} f''(\alpha) + \frac{\varepsilon_n^3}{6} f'''(\xi_1)}{f'(\alpha) + \varepsilon_n f''(\alpha) + \frac{\varepsilon_n^2}{2} f'''(\xi_2)}$$

$$
\begin{aligned}
&= \frac{\varepsilon_n f'(\alpha) + \varepsilon_n^2 f''(\alpha) + \frac{\varepsilon_n^3}{2} f'''(\xi_2) - \varepsilon_n f'(\alpha) - \frac{\varepsilon_n^2}{2} f''(\alpha) - \frac{\varepsilon_n^3}{6} f'''(\xi_1)}{f'(\alpha) + \varepsilon_n f''(\alpha) + \frac{\varepsilon_n^2}{2} f'''(\xi_2)} \\
&= \frac{\varepsilon_n^2}{2} \frac{f''(\alpha) + \varepsilon_n f'''(\xi_2) - \frac{\varepsilon_n}{3} f'''(\xi_1)}{f'(\alpha) + \varepsilon_n f''(\alpha) + \frac{\varepsilon_n^2}{2} f'''(\xi_2)}
\end{aligned}
$$

最後の式は, 近似的に $\frac{\varepsilon_n^2}{2}\frac{f''(\alpha)}{f'(\alpha)}$ に等しい. $n$ ステップ目での誤差が $\varepsilon_n$ なら $n+1$ ステップ目では誤差が $\varepsilon_n^2$ のオーダーとなる.

したがって, ニュートン法の誤差は, 2 乗, 4 乗, 8 乗, ... と急速に減っていく. たとえば, $f(x) = x^2 - 2$ の時, 正の根 $\alpha$ は約 1.14213 である. このとき $\frac{f''(\alpha)}{2f'(\alpha)}$ の値は $-0.177$ である. $|x_n - \alpha| < 10^{-p}$ なら, ほぼ $|x_{n+1} - \alpha| < 1.77 \times 10^{-2p}$ となる.

実際, 精度の高い浮動小数点計算を遂行する次のプログラムで, 誤差の様子をプリントすると以下のようになる.

```
setprec(20);
def sqrtByNewton(A) {
  Epsilon = (1/10)^20;
  P = 0;
  Q = A;
  while (!( (Q-P > -Epsilon) &&
            (Q-P < Epsilon))) {
     Q=eval(Q*exp(0)); print(Q);
     P = Q;
     Q = P-(P*P-A)/(2*P);
  }
  return(Q);
}
```

出力結果

```
[431] sqrtByNewton(2)$
2.00000000000000000000000000000000000000
1.50000000000000000000000000000000000000000
1.41666666666666666666666666666666666666670
1.41421568627450980392156862745098039216
1.41421356237468991062629557889013491012 5
1.41421356237309504880168962350253024362 0
[432] eval(2^(1/2));
1.41421356237309504880168872420444308449 3
```

関数 setprec で, 何桁の有効数字で計算をおこなうかを指定できる. setprec(n) の $n$ を大きくすると, 高い精度で近似計算をおこなう. sin などの関数の近似値を高い精度で計算するには, 次のように関数 eval を用いる.

```
eval(sin(@pi/3));
```

参考: 1 変数代数方程式の解法については, Frisco の 最終報告書の Algorithmic Research/Task 3.4: Numerical solving/ 3.4.1 Univariate solving
http://www.nag.co.uk/Projects/Frisco/frisco/node7.htm にいろいろな方法の比較がある. pari(root, 多項式) は Schoenhage アルゴリズムを用いて根を求めている.

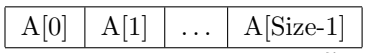
## 6.2　最大値と配列

配列というのは添字つきの変数で自由に代入や参照が可能である. 他の言語では array と呼ばれるが, Asir では数学的なオブジェクトである vector, matrix を 単なる入れ物として用いて array の代用品としている.

- 普通の変数

  名前のついた入れ物と思える. 変数が式の中に現れるとその中に入っている値と置き換わる. また, 代入式の左辺に現れると, 右辺の値がその入れ物の中身になる (上書き). 普通の変数は今までのプログラムで利用してきた. たとえば前節最後のプログラムの `Epsilon`, `P`, `Q`, `A` 等は (普通の) 変数である.

- 配列

  普通の変数 (入れ物) が長さ `Size` 個だけつながったもの. 

添字は 0 から始まる (要注意!) ので, 長さが `Size` なら `A[0]` から `A[Size-1]` までの入れ物があることになる.

1. 配列の作り方

```
[0] A = newvect(5);                長さ 5 の配列 (vector) を生成して A に代入
[ 0 0 0 0 0 ]                      最初は 0 ベクトル
[1] B = newvect(3,[1,2,3]);        長さ 3 の配列を初期値指定して生成
[ 1 2 3 ]                          B[0] = 1, B[1] = 2, B[2] = 3
```

2. 配列要素の取り出し

```
[3] B[2];                          B の添字 2 に対応する要素
3
[4] C = (B[0]+B[1]+B[2])/3;        式の中に現れる配列要素アクセス
2
[5] C;                             C には 2 が入っている
2
```

3. 配列要素への代入

```
[6] A[0] = -1;                     A の先頭に -1 を代入
-1
[7] A;                             A が配列なら, A の値は配列全体
[ -1 0 0 0 0 ]
```

4. 配列の長さの取り出し

```
[8] size(A);                       長さを取り出す組み込み関数
[5]                                リストを返す
[9] size(A)[0];                    リストの先頭要素が長さ
5
```

配列についてまとめると, 次のようになる.

- 配列は長さ固定の連続した入れ物.
- 長さ `N` の配列は `newvect(N)` で生成. 添字は 0 以上 `N-1` 以下.
- 各要素の書き換えは自由.

配列 (asir ではベクトルと呼ぶ) のなかのデータの最大値を求めてみよう.

```
/* cond8.rr */
def main() {
    Total=5;
    A = newvect(Total);
    for (K=0; K<Total; K++) {
      A[K] = random() % 100;
    }
    print(A);
    /* search the maximum */
    Max = A[0];
    for (K=0; K<Total; K++) {
      if (A[K] > Max) {
         Max = A[K];
      }
    }
    print("Maximum is ",0);
    print(Max);
}
```

まず, 乱数でデータを A へ格納してから, 最大値を探す. 実行結果は次のようになる.

```
[450] load("cond8.rr");
1
[453] main();
[ 58 10 55 62 2 ]
Maximum is 62
0
```

newvect(N) はサイズ N のベクトルを生成する関数である. random() は乱数を戻す関数である.

A % N は A を N でわった余り.

平均値を求めるプログラムも同じように書ける.

```
/* cond9.rr */
def main() {
    Total=5;
    A = newvect(Total);
    for (K=0; K<Total; K++) {
      A[K] = random() % 100;
    }
    Sum = 0;
    print(A);
    for (K=0; K<Total; K++) {
      Sum = Sum + A[K];
    }
    print("Average is ",0);
    print(Sum/Total);
}
```

左が平均値を求めるプログラムである. 実行例は

```
[445] load("cond9.rr");
1
[448] main();
[ 77 90 39 46 58 ]
Average is 62
0
```

## 6.3 効率的なプログラムを書くには?

例として $e = \sum_{i=0}^{\infty} 1/i!$ を部分和で近似することを考える. $e(N) = \sum_{i=0}^{N} 1/i!$ とする. そのままプログラム化すると

```
def factorial(N) {
  A = 1;
  for (I=1; I<=N; I++) A *= I;
  return A;
}

def e1(N) {
  E = 0;
  for ( I = 0; I <= N; I++ )
    E += 1/factorial(I);
  return E;
}
```

実はこの `e1` 関数はとても無駄が多い (何故か?) ので改良してみよう. $i! = i \cdot (i-1)!$ だから, `1/factorial(I)` の分母を `1/factorial(I-1)` から計算できる.

```
def e2(N) {
  F = 1;
  E = 1;
  for ( I = 1; I <= N; I++ ) {
    F *= I;
    E += 1/F;
  }
  return E;
}
```

```
E = 1/0!

F = I!
E = 1+1/1!+...+1/I!
```

実はこれでもまだ効率が無駄が多い. これは, 有理数計算特有の事情である. $a/b + c/d$ の計算には後の節で説明する整数 GCD の計算が必要になるが, $e(I-1)$ の分母は明らかに $I!$ を割るので GCD 計算は無駄になる. よって, $e(I) = a(I)/I!$ とおいて, $a(I)$ の漸化式を求めよう. $e(I) = e(I-1) + 1/I!$ より $a(I) = I \cdot a(I-1) + 1$. よって, 次のプログラムが書ける.

```
def e3(N) {
  A = 1;
  for ( I = 1; I <= N; I++ )
    A = I*A+1;
  return A/factorial(N);
}
```

**問題 6.2** 三つのプログラムを実際に書いて, 結果, 計算時間を比較せよ.

```
[...] cputime(1);
```

を実行すると, 計算時間が表示されるようになる.

**問題 6.3** $e(N) = N/D$ ($N$, $D$ は整数) と書けているとする.

- `idiv(A,B)` : `A/B` の整数部分を返す.
- `nm(Q)` : 有理数 `Q` の分子を返す.
- `dn(Q)` : 有理数 `Q` の分母を返す.

これらの組み込み関数を使って $e(N)$ の 10 進小数展開を $K$ 桁求めよ. (小数点は不要. $K$ 桁の表示できればよい.) 例えば $e(10) = 9864101/3628800$ で, 10 桁求めると 2718281801.

**問題 6.4**

$$\arctan x = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{x^{2n-1}}{2n-1}$$

と

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}$$

(マチンの公式) を使って $\pi$ の有理数近似値を求める関数を書け.

## 6.4 章末の問題

問題 1: 配列にはいったデータのなかで, $i$ 番目に大きいデータを戻すプログラムを書きなさい. (素朴な方法だと, 配列の大きさが大きくなるとなかなか実行が終らない. 第 14 章を学習してから改めてこの問題を考えてみるとよい. )
問題 2: 数学の本に書いてあることから, 適当な題材をえらびプログラムに書いてみなさい. 外積のような面倒なものの計算をやらせてもいいし, ヤコビ行列式の計算をやらせてみてもいい. 微分は関数 `diff` で可能. なお Risa/Asir では $\frac{f}{g}$ なる形の有理式の計算もできるが, 通分は 重たい計算なので自動的にはやらない. 関数 `red` を用いて通分を行う必要がある. たとえば次の例をみよ.

```
[344] H=(x-1)/(x^2-1);
(x-1)/(x^2-1)
[345] red(H);
(1)/(x+1)
```

問題 3:
次のプログラムの出力値を答えよ.

```
def main() {
   C = 10;
   D = (C > 5? 100 : 200);
   print(D);
}
```

問題 4:
次のプログラムの出力を書きなさい.

```
def main() {
  A = newmat(10);
  I=0; J = 0;
  while (I<9) {
     A[++I] = J;
     J++;
  }
  print(A[3]);
}
```

問題 5:
3 次方程式 $x^3 + Ax^2 + Bx + C = 0$ の実根を Newton 法で求めるプログラムを書け.
ヒント: この問題は本格的なパッケージ関数の開発へのチャレンジである. ささやかながら “プログラムの開発” の経験ができる. プログラムの開発は段階をおって進むものである. 次のような順番にしたがい開発をすすめよう.

1. レベル 1　とりあえず一根を求める. 反復が停止しないなどという失敗も有り得る.
2. レベル 2　実根の個数をあらかじめ調べ, 場合分けして初期値を適切に設定し, 重根がない場合に全実根を求める.
3. レベル 3　重根をとりあつかう.

ヒント 2: $f(x) = x^3 + Ax^2 + Bx + C$ に対し, $f(x) = 0$ の解を Newton 法で求める場合の初期値は次のように決められる.

1. $\forall x, f'(x) > 0$ の場合
2. $\exists x, f'(x) = 0$ の場合
   $f'(\alpha) = 0$, $f'(\beta) = 0$ $(\alpha < \beta)$ とすると $f(x)$ は $x = \alpha$ で極大, $x = \beta$ で極小.
   (a) $f(\beta) > 0$ の場合解は一つ. $x_0 < \alpha$ からスタート.
   (b) $f(\alpha) < 0$ の場合解は一つ. $x_0 > \beta$ からスタート.
   (c) $f(\alpha) > 0, f(\beta) < 0$ の場合解は三つ. $x_0 > \beta$, $x_0 < \alpha$, $x_0 = (\alpha + \beta)/2$ のそれぞれからスタート.

ヒント 3: 前のヒントのように初期値を選べば, 重根を持たない場合にはちゃんと解が得られる. $f'(x)$ が常に正かどうかは高校数学でおなじみの判別式でわかる. 重根があるかどうかは微妙な問題なので, ない場合に動くプログラムが書けていればそれでよい.

注意すべき点は, 初期値, 係数全てが有理数の場合, Newton 法の全ての計算が有理数で行われてしまい, 異常に時間がかかった挙げ句巨大な分母分子を持つ値が得られることがある. このようなことを避けるために, 初期値は `X=eval(A*exp(0))` あるいは `X=deval(A)` として, 強制的に浮動小数に変えるとよい.

数値計算を多用すると次のようなエラーに出会うことがある.

```
***   the PARI stack overflows !
 current stack size: 65536 (0.062 Mbytes)
 [hint] you can increase GP stack with allocatemem()
```

このようなエラーが出た場合には,

```
[295] pari(allocatemem,10^7)$
```

などを実行して, pari の使用できるメモリを増やすこと.

# 第7章　ユークリッドの互除法とその計算量

## 7.1　素因数分解

$a$, $b$ を自然数とするときその最大公約数 (Greatest Common Divisor, GCD) とは, $a$ と $b$ を共に割り切る数で最大のものである. この数は $a$ と $b$ を素因数分解すれば求まるが, 実は素因数分解で GCD を求めるのは効率がわるい. この章では, GCD 計算の高速アルゴリズムである Euclid の互除法を説明し, あわせて計算の効率を議論する計算量 (computational complexity) の理論への入門を試みる.

**例題 7.1** 入力された自然数 $n$ の素因数分解を求めよ.

```
def prime_factorization(N) {
   K = 2;
   while (N>=2) {
     if (N % K == 0) {
        N = idiv(N,K);
        print(K,0); print(", ",0);
     }else{
        K = K+1;
     }
   }
   print(" ");
}
```

N % K は N を K で割った余り. idiv(N,K) は N を K で割ったときの商をあらわす. このプログラムでは, K で試しに N をわってみて割り切れたら, 因子として出力する.

N が 60 の時の変数の変化:

| K | 2 | 2 | ··· |
|---|---|---|---|
| N | 60 | 30 | ··· |
| 行番号 | 3 と 4 の間 | 3 と 4 の間 | ··· |

問: この表を完成し, プログラムの動きを説明せよ.

## 7.2　計算量

N % K が 前章のプログラムで何回程度実行されるか考えてみよう. このプログラムが最悪の動作をするのは, N が素数の時である. N が素数の時には, K は N に達するまで 1 づつ増えつづける. したがって, N 回余りの計算が実行される. よって N の 2 進数であらわした桁数を $m$ とすると, $O(2^m)$ 回程度の余りの計算が実行されることになる.

ここで $O(2^m)$ は $O$-記法とよばれ, $m$ が十分大きい時, $[C2^m + (C'2^m$ より小さい数$)]$ 程度の大きさであるような数をあらわす. ここで $C$ は定数である. 例えば, $2m^2 - m = O(m^2)$, $100m \log m + 5000m = O(m \log m)$ である.

プログラムやアルゴリズムの実行時間やメモリ使用量を入力データサイズの関数で評価することを計算量の評価という. 実行時間 (時間計算量) を調べるには, 一番多く実行される文の実行回数を目安にするとよいであろう.

たとえば, 上のプログラム `prime_factorization(N)` の場合は `N` に素数をいれた場合, ループが O(N) 回実行される. したがって次のような定理がなりたつのはほぼ明らかであろう.

**定理 7.1** 素因数分解アルゴリズム `prime_factorization(N)` の最悪時間計算量は $O(2^m)$ である. ここで `N` を 2 進数であらわしたときの桁数を $m$ とする.

アルゴリズムの性能表示は $O(m), O(m^2), O(\log m), O(m \log m), O(e^m)$ など $O$-記法を利用しておこなわれる. $O(\log m), O(m), O(m \log m), O(m^2)$ がどの程度違うか表にしてみてみよう. ここで $\log m$ は 2 を底とする $m$ の対数である.

| $m$ | 10 | 100 | 100,000 | 1,000,000 |
|---|---|---|---|---|
| $\log m$ | 3(くらい) | 6 | 16 | 19 |
| $m \log m$ | 30 | 600 | 1,600,000 | 19,000,000 |
| $m^2$ | 100 | 10000 | 100 億 | 1 兆 |

この表を見ればわかるように, $O(m)$ のアルゴリズムと $O(m^2)$ のアルゴリズムは $m$ が大きくなると性能差がきわめて大きくなる. いわんや $O(m)$ と $O(2^m)$ の差はきわめて大きい. 上にしめした素因数分解のアルゴリズムは $O(2^m)$-アルゴリズムであり, これを元にした, GCD 計算ももちろん $O(2^m)$-アルゴリズムとなる. GCD 計算には, これとはくらべものにならないくらい早い $O(m)$ のアルゴリズムがある. これが, ユークリッドの互除法である.

## 7.3　互除法

$a, b$ の GCD を $\gcd(a, b)$ であらわそう. 便宜上, $\gcd(a, 0)$ は $a$ と定義する. ユークリッドの互除法は次の定理を基礎としている.

**定理 7.2** $a, b$ を自然数として $a \geq b$ と仮定しよう. このとき

$$\gcd(a, b) = \gcd(b, r)$$

がなりたつ. ここで, $q$ を $a$ わる $b$ の商, $r$ を $a$ わる $b$ の余り, すなわち,

$$a = bq + r, \quad r < b$$

が成立していると仮定する.

証明: $d$ が $a, b$ の公約数なら, $r = a - bq$ なので, $d$ は $r$ を割り切る. また $b$ も割り切る. したがって, $d$ は $b$ と $r$ の公約数である.

$d'$ が $b$ と $r$ の公約数なら, 同じ理由で, $d'$ は $a$ と $b$ の公約数である.

したがって, $a, b$ の公約数の集合と $b, r$ の公約数の集合は等しい. とくに GCD 同士も等しい. 証明おわり.

**例題 7.2** 18 と 15 の GCD を上の定理を利用して求めよ.

$$18 \div 15 = 1 \ \cdots 3$$
$$15 \div 3 = 5 \ \cdots 0$$

である. したがって, 上の定理より.

$$\gcd(18,15) = \gcd(15,3) = \gcd(3,0) = 3$$

となる.

この GCD 計算方法をユークリッドの互除法という. プログラムに書くと次のようになる. 次の関数 `e_gcd(A,B)` は数 `A` と数 `B` の GCD を互除法で求める.

```
def e_gcd(A,B) {
  if (B>A) {
    T = A; A = B; B=T;
  }
  while (B>0) {
    R = A%B;
    A=B; B=R;
  }
  return(A);
}
```

さてこのアルゴリズムの計算量を考察しよう. 命令 `R = A%B` が何回実行されるのか考えればよいであろう. (最悪) 計算量を求めるには, プログラムが最悪の振舞をするデータが何かわかれば計算量の評価ができる. 実は互除法での最悪の場合のこの回数はフィボナッチ数列で実現できる. 次の漸化式

$$F_{k+2} = F_{k+1} + F_k, \quad F_0 = F_1 = 1$$

できまる数列をフィボナッチ数列とよぶ.

**定理 7.3** 互除法による $a$ と $b$ の GCD 計算が $n$ 回の `R = A%B` の計算で終了したとする. このとき, $b$ の値によらず, $a \geq F_n$ が成立する.

証明: $a_n = a$, $a_{n-1} = b$ とおく. 互除法の各ステップにでてくる数を次のように $a_k$, $a_{k-1}$ とおく.

$$a_n \div a_{n-1} = q_n \ \cdots \ a_{n-2}$$
$$a_{n-1} \div a_{n-2} = q_{n-1} \ \cdots \ a_{n-3}$$
$$\cdots$$
$$a_1 \div a_0 = q_1 \ \cdots \ 0$$

このように定義すると

$$a_{k+2} = q_{k+2}a_{k+1} + a_k, \quad q_{k+2} \geq 1$$

がなりたつ. また $a_0 \geq 1$, $a_1 \geq 2$ がなりたつ. よって, フィボナッチ数列の漸化式とくらべることにより,

$$a_k \geq F_k \quad k = 0, 1, 2, \ldots, n$$

が成り立つ. 証明おわり.

この証明により, $a = F_n$, $b = F_{n-1}$ に互除法を適用すると, $n$ 回の `R = A%B` の計算が必要なことも分る.

$F_n$ の一般項を計算することにより, 次の定理が得られる.

**定理 7.4** $m$ 桁の数の GCD の計算は, 互除法で $O(m^2)$ 時間でできる.

**問題 7.1** $F_n$ の一般項を計算し, 上の定理の証明を完成せよ.

上の結果により, 互除法による GCD 計算は素因数分解による GCD 計算にくらべ圧倒的に早いことがわかる.

**問題 7.2** $a = 1000000000000283$ と $b = 3256594799$ の GCD を互除法および素因数分解を用いて求めよ. 時間を計測せよ. 1 時間待っても答えがでないときはあきらめよ.

インターネットなどで用いられている暗号は素因数分解に計算時間がかかる — 計算量が大きい — という経験則に立脚して設計されている.

## 7.4　参考: 領域計算量と時間計算量

整数 $n$ の素数判定には自明な方法がある.

- 2, 3, ..., $n-1$ で割ってみる.
- 2, 3, ..., $\lfloor\sqrt{n}\rfloor$ で割ってみる. ($\lfloor x\rfloor$ は $x$ を越えない最大の整数.)
- 2 以外は奇数で割る.

いずれにせよ $\sqrt{n}$ に比例する回数の割算が必要である. たとえば $n \simeq 2^{1024} \simeq 10^{308}$ のとき, $\sqrt{n} \simeq 2^{512} \simeq 10^{154}$. 計算機では 1 クロックを単位として各種の命令が実行されている. 現在の一般的な CPU のクロックは 1GHz ($10^9$Hz) 程度, すなわち単位操作を一秒間に $10^9$ 回できる程度なので, 1 クロックで 1 回割算ができても $10^{145}$ 秒 $\simeq 3.17 \times 10^{137}$ 年程かかることになる.

$2^{1024}$ 程度の数の素因数分解は, もっとよいアルゴリズムを用いても困難であり, それが RSA 暗号の安全性の根拠となっている.

素朴には, 計算量とは, ある大きさのデータに対して, 計算にどれだけ時間 (ステップ数) あるいは領域 (メモリ量) がかかるかを示す量である. ここでいくつか曖昧な点がある. この点をもうすこし詳しく考察しよう.

- データの大きさ
  - 大きな数も小さな数も一つの数と見る
  - メモリ上で必要な量を単位とする.
- 1 ステップとは
  - 数の計算は 1 ステップと見る.
  - 計算機の命令を 1 ステップと見る.

例えば浮動小数演算を用いる場合には, どちらで考えても同じだが, 近似なしに正確な値を有理数で求めていく場合には注意が必要である. たとえば, 2 つの $n$ 桁の整数

$$a = \sum_{i=0}^{n-1} a_i \cdot 10^i, \quad b = \sum_{i=0}^{n-1} b_i \cdot 10^i$$

を筆算の要領で計算することを考える.

$$ab = \sum_{i=0}^{n-1} ab_i \cdot 10^i$$

を使って, $ab_i$ を計算してから $i$ 桁左にシフトしながら足して行く. 一桁の数 $u$ に対し $au = \sum_{i=0}^{n} m_i \cdot 10^i$ を計算するアルゴリズムは次の通り.

```
c ← 0
for ( i = 0; i < n; i++ )
    t ← a_i b + c
    m_i ← t mod 10
    c ← ⌊t/10⌋
end for
m_n ← c
```

この計算で, かけ算は $n$ 回, 10 による割算が $n$ 回必要である.

次に, $ab_i$ をシフトして足していく計算を考える. これは, 繰り上がりを無視すれば $n$ 桁の数 $\sum_{i=0}^{n} f_i \cdot 10^i$, $\sum_{i=0}^{n} g_i \cdot 10^i$ の加算と考えてよいから次のアルゴリズムで計算できる.

```
c = 0
for ( i = 0; i < n; i++ )
    t ← f_i + g_i + c
    if t ≥ 10 f_i ← t − 10, c ← 1
    else f_i ← t, c ← 0
end for
(繰り上がりの処理)
```

これは, 本質的には加算が $n$ 回で計算できる. 以上の計算が $n$ 回必要になるから, $n$ 桁の数の積の計算には $n^2$ に比例するステップ数が必要になることが分かる.

$n$ 桁の整数を二つかける筆算アルゴリズムの計算量は $O(n^2)$ であった. このアルゴリズムはさらに改良可能である. 簡単な高速化アルゴリズムとして, Karatsuba アルゴリズムを紹介する. まず, 二桁の数 $a = a_1 \cdot 10 + a_0$ $b = b_1 \cdot 10 + b_0$ の積を考える. 普通にやると

$$ab = a_1 b_1 \cdot 10^2 + (a_1 b_0 + a_0 b_1) \cdot 10 + a_0 b_0$$

とかけ算が 4 回現れる. これを

$$ab = a_1 b_1 \cdot 10^2 + ((a_1 - a_0)(b_0 - b_1) + a_1 b_1 + a_0 b_0) \cdot 10 + a_0 b_0$$

と変形するとかけ算は $a_1b_1$, $a_0b_0$, $(a_1 - a_0)(b_0 - b_1)$ の 3 回になる. (加算は増える.) これを繰り返す. $2^n$ 桁の整数 $a$, $b$ をかける場合の計算量を $T(2^n)$ とする.$N = 2^{n-1}$ として $a = a_1 \cdot 10^N + a_0$, $b = b_1 \cdot 10^N + b_0$, $0 \leq a_i, b_i \leq 10^N$ と書いて, 上の考察を適用する. 一桁の乗算, 加減算のコストをそれぞれ $M$, $A$ とすると,

$$T(2^n) = 3T(2^{n-1}) + 4 \cdot 2^n A.$$

(第 2 項は「倍長」整数の加算コストを表す. $T(1) = M$ より, この漸化式を解いて,

$$T(2^n) = (M + 8A) \cdot 3^n - 8A \cdot 2^n.$$

よって, $T(2^n) = O(3^m)$. これより

$$T(n) = O(3^{\log_2 n}) = O(n^{\log_2 3}).$$

$\log_2 3 \simeq 1.58$ より漸近的には筆算よりよいアルゴリズムと言える.

なお, 一変数多項式の積でも同じアルゴリズムが使える. かなり低い次数 (20 次程度) から, 通常の $O(n^2)$ アルゴリズムより高速になる.

## 7.5　章末の問題

1. (計算量 — computational complexity — の理論への導入) `prime_factorization` の `N % K` が何回実行されたかを数えることができるように改良し, 入力する数 `N` をいろいろ変えて実行しこの余りを求める演算が何回実行されるか記録しなさい. それをグラフにまとめなさい (方眼紙またはそれに準ずるものを用いてきれいに書こう).

2. フィボナッチ数 $F_0, \ldots, F_{200}$ を求めるプログラムを書きなさい.

3. $\gcd(F_k, F_{k-1}) = 1$ であることを証明せよ.

4. 1000000000000283 は二つの素数に分解する. この分解を試みよ.

## 7.6　章末付録: パーソナルコンピュータの歴史 — CP/M80

Intel 8080 CPU をのせた TK-80 シュミレータで 8080 CPU のマシン語のプログラムを楽しんだであろうか？その後, Intel 8080 CPU は上位互換の Zilog Z80 CPU に市場でとってかわられることになる. ベストセラーとなった, NEC PC8801 シリーズは Zilog Z80 CPU を搭載し, ROM (Read Only Memory) に書き込まれた N88 Basic が電源投入と共に起動した. N88 Basic は Mirosoft Basic をもとに NEC が機能拡張した Basic 言語であり, 大人気を博した. PC8801 シリーズでは CP/M80 という Digital Research 社により開発されたディスクオペレーティングシステムも実行することが可能であった.

現在, たとえば FreeBSD 上の `cpmemu` なる CP/M 80 エミュレータを用いることにより, この CP/M 80 上の Microsoft Basic (MBASIC) を楽しむことが可能である. CP/M 80 の多くの商用ソフトは現在 http://deltasoft.fife.wa.us/cpm に保存されており, 自由に取得することが可能である.

GCD を計算するユークリッドのアルゴリズムを FreeBSD 上の cpmemu 上の MBASIC で実装してみよう.

```
bash$ ls
mbasic.com
bash$ cpmemu

A0>dir
A: MBASIC  .COM
A0>mbasic
BASIC-80 Rev. 5.21
[CP/M Version]
Copyright 1977-1981 (C) by Microsoft
Created: 28-Jul-81
35896 Bytes free
Ok
10 input a,b
20 r=a mod b
30 if r = 0 then goto 50
40 a=b: b=r: goto 20
50 print b
60 end
list
10 INPUT A,B
20 R=A MOD B
30 IF R = 0 THEN GOTO 50
40 A=B: B=R: GOTO 20
50 PRINT B
60 END
Ok
run
? 1234,1200
 2
Ok
system

A0>unix
Cp/M BIOS COLDBOOT takes you back to FreeBSD
bash$
```

CP/M 80 では, MBASIC 以外にさまざまなプログラム言語が動作した. そのなかでもっとも人気を博したのが ターボパスカル (turbo Pascal) である. 言語パスカルは 1970 年代前半にチューリッヒ工科大学の Niklaus Wirth により設計された美しい言語である. ターボパスカルでは, エディタとコンパイラが統合されており, さらに 100 行ほどのプログラムは 1 秒程度でコンパイル, 実行可能であった. 現在の高速なパソコンを利用している人にはこの感動はつたわらないかもしれないが, 当時

の Z80 CPU のパソコンでこのスピードは驚異的であった. ちなみに他の会社のコンパイラで同じプログラムをコンパイルしても, 1 分以上かかるのが通例であった.

```
0               :  1
1               :  1
2               :  2
3               :  3
4               :  5
5               :  8
6               :  13
7               :  21
8               :  34
9               :  55
10              :  89
11              :  144
12              :  233
13              :  377
14              :  610
15              :  987
16              :  1597
17              :  2584
18              :  4181
19              :  6765
20              :  10946
21              :  17711
22              :  28657
23              :  46368
24              :  75025
25              :  121393
26              :  196418
27              :  317811
28              :  514229
29              :  832040
30              :  1346269
31              :  2178309
32              :  3524578
33              :  5702887
34              :  9227465
35              :  14930352
36              :  24157817
37              :  39088169
38              :  63245986
39              :  102334155
40              :  165580141
41              :  267914296
42              :  433494437
43              :  701408733
44              :  1134903170
45              :  1836311903
46              :  2971215073
47              :  4807526976
48              :  7778742049
49              :  12586269025
50              :  20365011074
51              :  32951280099
52              :  53316291173
53              :  86267571272
54              :  139583862445
55              :  225851433717
56              :  365435296162
57              :  591286729879
58              :  956722026041
59              :  1548008755920
```

```
60              :  2504730781961
61              :  4052739537881
62              :  6557470319842
63              :  10610209857723
64              :  17167680177565
65              :  27777890035288
66              :  44945570212853
67              :  72723460248141
68              :  117669030460994
69              :  190392490709135
70              :  308061521170129
71              :  498454011879264
72              :  806515533049393
73              :  1304969544928657
74              :  2111485077978050
75              :  3416454622906707
76              :  5527939700884757
77              :  8944394323791464
78              :  14472334024676221
79              :  23416728348467685
80              :  37889062373143906
81              :  61305790721611591
82              :  99194853094755497
83              :  160500643816367088
84              :  259695496911122585
85              :  420196140727489673
86              :  679891637638612258
87              :  1100087778366101931
88              :  1779979416004714189
89              :  2880067194370816120
90              :  4660046610375530309
91              :  7540113804746346429
92              :  12200160415121876738
93              :  19740274219868223167
94              :  31940434634990099905
95              :  51680708854858323072
96              :  83621143489848422977
97              :  135301852344706746049
98              :  218922995834555169026
99              :  354224848179261915075
100             :  573147844013817084101
101             :  927372692193078999176
102             :  1500520536206896083277
103             :  2427893228399975082453
104             :  3928413764606871165730
105             :  6356306993006846248183
106             :  10284720757613717413913
107             :  16641027750620563662096
108             :  26925748508234281076009
109             :  43566776258858444738105
110             :  70492524767089125814114
111             :  114059301025943970552219
112             :  184551825793033096366333
113             :  298611126818977066918552
114             :  483162952612010163284885
115             :  781774079430987230203437
116             :  1264937032042997393488322
117             :  2046711111473984623691759
118             :  3311648143516982017180081
119             :  5358359254990966640871840
```

Risa/Asir ドリル ギャラリー : フィボナッチ数

```
[349] pari(nextprime,10^130);
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000001113
[350] pari(nextprime,10^131);
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000063
[351] pari(nextprime,10^132);
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000169
[352] pari(nextprime,10^133);
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000021
```

Risa/Asir ドリル ギャラリー : 素数の生成

# 関連図書

[1] 木田, 牧野, Ubasic によるコンピュータ整数論, 日本評論社
素因数分解についてのさまざまなアルゴリズムの解説および Ubasic によるプログラムが収録してある. 素因数分解について興味をもったら是非読んでほしい本の 1 冊である.

# 第8章　関数

あるひとまとまりのプログラムは関数 (function) としてまとめておくとよい．　計算機言語における関数は数学でいう関数と似て非なるものである．関数を手続き (procedure) とか サブルーチン (subroutine) とかよぶ言語もある．

## 8.1　リストとベクトル (配列)

ベクトルは, 6.2 節ですでに使用したが, 関数の説明にはいるまえにリスト (list) とベクトル (vector) について補足説明しておこう．リストおよびベクトルともにさまざまなアルゴリズムを実現するための重要なデータ構造である．リストについて詳しくは 13 章で議論する．適宜参照してほしい．

リスト型のデータはたとえばいくつかのデータをまとめて扱いたい場合にもちいる．リストはベクトルとことなり, 実行中にサイズを変更できるが, $k$ 番目の要素に代入したりできない．それ以外の扱いはベクトルに似ている．

リスト構造の起源は Lisp 言語である．Lisp 言語ではすべてのデータをリストとして表す．Lisp 言語は一番長い歴史をもつ言語にもかかわらず現在でもいろいろな場面で利用されている．たとえば Emacs マクロとよばれるものは, Emacs Lisp と呼ばれる Lisp 言語で記述されている．Lisp ではリストを括弧を用いて生成するが, Asir ではリストは [, ] を用いて生成する．

```
[345] A=[10,2];
[10,2]
[346] A[0];
10
[347] A[1];
2
[348] B=["dog","cat","fish"];
[dog,cat,fish]
[349] B[0];
dog
```

変数 A には長さ 2 のリスト [10,2] が代入された．10 が 0 番目の元, 2 が 1 番目の元であり, それぞれ A[0], A[1] で取り出せる．リストの元は, 数字である必要はなく, 文字列でもよいし, 別のリストであってもよい．

Asir のベクトルは C 等の言語では 配列 (array) とよばれるデータ型に似てる．Asir でもベクトルを配列と呼ぶ場合もあるので, ベクトルと配列は同一のものだとおもってよい．C の配列とことなり, Asir の配列は何でもいれることが可能なので k 番目の要素に代入が可能で, 高速に扱うことのできる大きさが固定されたリスト構造と思っておけばよい．リストに対しては代入, たとえば A[0] = 2 といった操作ができないが, ベクトルにたいしてはその操作も可能である．新しいベクトルは, たとえばコマンド

```
A=newvect(3,[10,2,5]);
```

で生成する. ここで 最初の 3 は, ベクトルの長さである. コマンド

```
newvect(3);
```

で長さ 3 で要素がすべて 0 のベクトルを生成する. ベクトルからリストを生成するには組み込み関数 vtol を用いればよい.

与えられた変数に格納されたデータが, リストかベクトルかを区別したい場合, 組み込みの type 関数を用いる. 変数 A に代入されているものが, ベクトルである場合, type(A) は 5 を戻す. 変数 A に代入されているものが, リストである場合, type(A) は 4 を戻す.

ベクトルがでてきたついでに, 行列についても簡単に説明しておこう. Asir では, たとえば A=newmat(2,2,[[1,2],[3,4]]); で $2 \times 2$ 行列 $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ を生成し, 変数 A に代入する. 行列の (I,J) 成分は A[I][J] で参照する. ただし, 普通の数学の添字とことなり, 添字は 0 から始まる.

## 8.2　関数と局所変数

関数を用いる最大の利点は, 関数を一旦書いてしまえば, 中身をブラックボックスとして扱えることである. 大規模なプログラムを書くときは複雑な処理をいくつかの関数に分割してまず各関数を十分テストし仕上げる. それからそれらの関数を組み合わせていくことにより, 複雑な機能を実現する. このようなアプローチをとることにより, “困難が分割” される.

簡単な関数の例をとり関数の書き方を説明しよう.

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

関数 nsum(N) は $\sum_{i=1}^{N} i$ の値を計算して戻す. N を関数の引数 (argument) とよぶ.
例:

```
[445] nsum(10);
55
[446] nsum(100);
5050
```

関数の戻り値 (return value) は return 文で与える. いまの場合は変数 S の値である. なお, print と return は違う. print は画面に値を印刷するのに対して, return は関数の値を戻す働きを持つ. print 文では, 関数の値を戻すことはできない.

関数のなかで利用されている変数と引数は, その関数の実行中のみ生成される変数であり, さらにその関数外部の同名の変数の値を変えない. このように一時的に生成される変数を局所変数 (local variable) とよぶ. 関数を用いて処理を分割したとしても, 関数の中で変数の値を変更したら, その関数の外の同じ名前の変数の値もかわってしまうとしたら, 分割した利点がすくない. そこでででてきた概念がこの “局所変数” の概念である. 上のプログラム例では, N, S, I が局所変数である. 局所変数はその関数のなかだけで有効な変数である. これを, “局所変数のスコープはその関数のなかだけ” という言いかたをする. 局所変数の考え方は, 計算機言語の歴史では大発明の一つである.
例:

```
[447] S=3;
3
[448] N=4;
4
[449] nsum(10);
55
[450] S;
3
[451] N;
4
```

[447] の S と関数 nsum のなかの変数 S は別物である. したがって, nsum の終了時点で関数 nsum のなかの変数 S の値は 55 であるが, [450] で S の値を表示させてみてもやはり 3 のままである. 引数の N についても同様である. 図 8.1 も参照.

nsum(10) のよびだし直前

| 場所 | 内容 |
|---|---|
| S | 3 |
| N | 4 |
| | |
| | |
| | |

nsum(10) の終了直前

| 場所 | 内容 |
|---|---|
| S | 3 |
| N | 4 |
| S(nsum の S) | 55 |
| N(nsum の N) | 10 |
| | |

図 8.1: メモリの図解

Asir では関数内部の変数は自動的に局所変数とみなされる. ただしこのような計算機言語はむしろ例外に属し, 多くの計算機言語では局所変数は宣言しないといけない. たとえば C 言語で, nsum を書くとつぎのようになる.

```
#include <stdio.h>
int nsum(int N) {
  int S;
  int I;
  S=0;
  for (I=1; I<=N; I++) {
    S= S+ I;
  }
  return(S);
}
main() {
  printf("%d\n",nsum(10));
  printf("%d\n",nsum(100));
}
```

このプログラムは C 言語で 1 から 10 までの和, 1 から 100 までの和を計算して印刷するプログラムの例である. int S;, int I が局所変数の宣言である. この文は実行はされない. unix 上ではこのプログラムをたとえば, local.c なる名前で save し,

```
bash$ cc local.c
bash$ ./a.out
```

でコンパイル, 実行できる.

関数の中から外で定義された変数を変更したいときもあるかもしれない. そのようなときは, 関数の先頭で, extern 宣言すればよい.

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

```
[444] S=10;
[445] nsum(10);
55
[446] S;
55
```

関数 nsum の中の変数 S は, 関数 nsum の外の変数 S と同一の変数なので, [446] で S の値が 55 になってる.

ことなる関数同士の局所変数は互いに無関係である.

```
def nsum(N) {
  S=0;
  for (I=1; I<=N; I++) {
    S= S+ I;
  }
  return(S);
}
def make_table(N) {
  for (I=1; I<=N; I++) {
     print(I,0); print(" :  ",0);
     print(nsum(I));
  }
}
```

```
[347] make_table(4);
1 :  1
2 :  3
3 :  6
4 :  10
0
```

関数 make_table(N) は $\sum_{i=1}^{p} i$ の表を $p = 1,\ldots,$N に対して作成する. make_table の N, I と nsum の N, I は別ものである. 図 8.2 を見よ.

make_table(4) の実行中で nsum(1) を呼ぶまえ.

| 場所 | 内容 |
|---|---|
| N(make_table の N) | 4 |
| I(make_table の N) | 1 |
| | |

make_table(4) の実行中で nsum(3) の終了直前.

| 場所 | 内容 |
|---|---|
| N(make_table の N) | 4 |
| I(make_table の N) | 3 |
| S(nsum の S) | 6 |
| N(nsum の N) | 3 |
| | |

図 8.2: メモリの図解

関数はかならずしも値を戻す必要はない.

```
def hello(N) {
  for (I=0; I<N; I++) {
     print("Hello!");
  }
}
```

関数 hello(N) は N 回 Hello を画面に表示する関数である.

```
[346] hello(3);
Hello!
Hello!
Hello!
0
[347] hello(3)$
Hello!
Hello!
Hello!
[348]
```

## 8.3 プログラム例

**例題 8.1** 最大値を返す関数を定義しよう.

プログラム func1.rr

```
/* func1.rr */
def max(A,B) {
  if (A>B) return(A);
  else return(B);
}
end$
```

max(A,B) は A と B の大きい方を戻す関数である.

実行例

```
[123] load("func1.rr");
1
[124] max(10,20);
20
```

**例題 8.2** 互除法で GCD を計算する関数はつぎのようにかける.

```
def abs(A) {
  if (A<0) return(-A);
  return(A);
}
def mygcd(A,B) {
  if (abs(B)>abs(A)) {
    T=A; A=B; B=T;}
  while (B != 0) {
    R = A % B;
    A = B; B = R;
  }
  return(A);
}
```

```
[349] mygcd(13,8);
1
[350] mygcd(8,6);
2
```

abs(A) は A の絶対値を戻す関数である.

**例題 8.3** 次は局所変数の考えかたの復習問題である. 出力がどうなるか間違いなくいえないといけない.

プログラム

```
/*func2.rr*/
def main() {
  I = 0;
  print(I);
  foo();
  print(I);
}
def foo() {
  I = 100;
}
main();
end$
```

左のプログラムの実行結果は

```
[48] load("func2.rr");
1
[49]
0
0
```

となる. けっして、100 とは表示されない. たとえ foo を

```
def foo() {
  I=100;
  return(I);
}
```

としたろころで同じである.

**例題 8.4** 次に 漸化式 $x_n = 2x_{n-1} + 1,\ x_0 = 0$ できまる数列の $n$ 項めをもとめるプログラムを作ろう.

プログラム

```
/* func4.rr */
def xn(N) {
    Re=0;
    for (K=0; K<N; K++) {
      Re = 2*Re+1;
    }
    return(Re);
}
```

実行例は以下のとうり.

```
[345] load("func4.rr")$
[346] xn(1);
1
[347] xn(10);
1023
[348] xn(20);
1048575
```

**例題 8.5** 次にリストを引数としてその要素の最大値と最小値を戻す関数をつくろう. 考え方は前節のプログラムと同じである. 二つの値を戻すために, 戻り値はリストかベクトルにするとよい.

```
/* func3.rr */
def minmax(A) {
  N = length(A);
  if (length(A) == 0) return(0);
  Max = Min = A[0];
  for (K=1; K<N; K++) {
     if (Max < A[K]) {
        Max = A[K];
     }
     if (Min > A[K]) {
        Min = A[K];
     }
  }
  return([Max,Min]);
}
end$
```

答えは [ 最小値, 最大値 ] のリストの形にして戻す. このような関数を多値関数と呼ぶ時もある. 実行例は以下のとうり.

```
[345] load("func3.rr")$
[346] minmax([1,4,2,6,-3,-2]);
[6,-3]
```

length(L) はリスト L のサイズ (長さ) を戻す関数である. ちなみに, ベクトル V のサイズ (長さ) をもどすには size(V)[0] とすればよい.

さて, 前の節で引数はその関数の実行中のみ存在する変数であると説明した. ベクトルやリストが引数として関数にわたされたときは内部的にはその先頭アドレスが関数にわたされベクトルやリスト自体は複製されていないことを了解しておく必要がある. このことを明確に理解するには, C のポインタや機械語の間接アドレシングの仕組みをきちんと勉強する必要があるかもしれない.
次のプログラムは, あたえらたベクトルのすべての成分を 1 にかえる.

```
def vector_one(V) {
  N = size(V)[0];
  for (I=0; I<N; I++) {
    V[I] = 1;
  }
}
end$
```

実行例:

```
[349] A=newvect(10);
[ 0 0 0 0 0 0 0 0 0 0 ]
[350] vector_one(A);
1
[351] A;
[ 1 1 1 1 1 1 1 1 1 1 ]
```

このようにベクトル A のすべての要素が 1 にかきかえれた.
size(V)[0] は ベクトル V の長さを戻す.

**問題 8.1 (10)** 上のプログラムで関数 vector_one() の最後の行に

```
V = 0;
```

を加えると,

```
[351] A;
[ 1 1 1 1 1 1 1 1 1 1 ]
```

となるだろうか. それとも,

```
[351] A;
0
```

となるだろうか? 関数実行中のメモリの様子を考えてどちらか答えよ.

図 8.3 にあるように, 関数にベクトルがわたされてもその複製は作成されない. これが普通の引数と違う点である.

vector_one(A) のよびだし直前

| 場所 | 内容 |
|---|---|
| A | A[0](先頭) のアドレス |
| A[0] | 0 |
| A[1] | 0 |
| ... | ... |
| A[9] | 0 |
| | |

vector_one(A) の終了直前

| 場所 | 内容 |
|---|---|
| A | A[0](先頭) のアドレス |
| A[0] | 1 |
| A[1] | 1 |
| ... | ... |
| A[9] | 1 |
| V | A[0](先頭) のアドレス |

図 8.3: メモリの図解

プログラム言語 Pascal では引数の配列を関数内で複製する (clone) か複製しないかを指示できる. キーワード var をつけると複製せず, つけないと複製する. プログラム言語 C や Java の場合は複製はされない. 複製は明示的におこなう必要がある. Asir でもベクトルやリストの複製は明示的におこなう必要がある. 代入演算子を用いても複製されていないことに注意. たとえば, A がベクトルとするとき B に代入しても複製はつくられない. 次の例を見よ.

```
[347] A=newvect(10);
[ 0 0 0 0 0 0 0 0 0 0 ]
[348] B=A;
[ 0 0 0 0 0 0 0 0 0 0 ]
[349] B[0]=100;
100
[350] B;
[ 100 0 0 0 0 0 0 0 0 0 ]
[351] A;
[ 100 0 0 0 0 0 0 0 0 0 ]
```

A がベクトルとするとき複製 (close) はつぎのようにおこなう.

```
N=size(A)[0];
B=newvect(N);
for (I=0; I<N; I++) {
   B[I] = A[I];
}
```

**問題 8.2 (20)**

1. 与えられたベクトルを複製する関数 clone_vector をつくりなさい.
2. あたなたの作った clone_vector は要素がベクトルのときはどのような動作をするか? たとえば, 引数に newvect(3,[1, newvect(2,[10,20]),3]) を与えたときはどのように動作するか?

## 8.4 デバッガ (より進んだ使い方)

4.2 節では, 不幸にしてエラーが起きてデバッグモードに入ってしまった場合の抜け方や, エラーの原因となった変数の値などを調べる方法のみ説明したが, ここではデバッガをより積極的に使う方法を紹介する. 詳しくは, "野呂, 下山, 竹島 : Asir User's Manual" (cf. 26.3 節) の第 5 章を参照してほしい.

### 8.4.1 ブレークポイント, トレースの使用

どこがおかしいかはまだ分からないが, とりあえず, 実行中のある時点での変数の値を見て考えよう, ということはよくある. このような場合, プログラム中に print の呼び出しを入れることで値を見ることはできるが,

- 後で外すのが面倒.
- どんどん表示が流れていくので結局何がおこっているのか分からない.

などの欠点がある. このようなときブレークポイント, トレースが便利である.

たとえば, 行列の積を計算するつもりで次のプログラムを書いたとしよう.

プログラム

```
def mat_mul(A,B)
{
  SA = size(A);
  SB = size(B);
  if ( SA[1] != SB[0] )
    error("mat_mul:size mismatch");
  N = SA[0]; L = SA[1]; M = SB[1];
  C = newmat(SA[0],SB[1]);
  for ( I = 0; I < N; I++ )
    for ( J = 0; J < M; J++ ) {
      for ( K = 0; K < L; K++ )
        T += A[I][K]*B[K][J];
        C[I][J] = T;
    }
  return C;
}
```

実行結果

```
[100] A = newmat(2,2,[[1,2],[3,4]]);
[ 1 2 ]
[ 3 4 ]
[101] mat_mul(A,A);
[ 7 17 ]
[ 32 54 ]
```

手で計算してみると, (0,0) 成分以外は全部おかしい. そこでデバッグモードに入ってブレークポイントを設定する.

```
[102] debug;
(debug) list mat_mul
1       def mat_mul(A,B)
2       {
3               SA = size(A);
4               SB = size(B);
5               if ( SA[1] != SB[0] )
6                       error(‘‘mat_mul : size mismatch’’);
7               N = SA[0]; L = SA[1]; M = SB[1];
8               C = newmat(SA[0],SB[1]);
9               for ( I = 0; I < N; I++ )
10                      for ( J = 0; J < M; J++ ) {
(debug) list
11                              for ( K = 0; K < L; K++ )
12                                      T += A[I][K]*B[K][J];
13                              C[I][J] = T;
14                      }
15              return C;
16      }
17      end$
(debug)
```

```
(debug) stop at 11
(0) stop at "./mat_mul":11
(debug) quit
[103] mat_mul(A,A);
stopped in mat_mul at line 11
in file "./mat"
11     for ( K = 0; K < L; K++ )
(debug) print [I,J]
[I,J] = [0,0]
(debug)
```

とりあえず, 11 行目にブレークポイントを設定して実行してみる. 11 行目で止まったら, (0,0) 成分を表示してみる.

```
(debug) cont
stopped in mat_mul at line 11
in file "./mat"
11    for ( K = 0; K < L; K++ )
(debug) print [I,J]
[I,J] = [0,1]
(debug)
```

(0,0) 成分は正しいので, (0,1) 成分の計算まで行く. これはコマンド `cont` (continue) を使う.

```
(debug) next
stopped in mat_mul at line 12
in file "./mat"
12     T += A[I][K]*B[K][J];
(debug) print T
T = 7
(debug)
```

次の行まで行く. これはコマンド `next` を使う. 止まったら, `T` の値を表示してみると, 積和計算用の変数 `T` が既に値を持っている. 要するに, 単なる `T` の初期化のし忘れだった.

この例はあまりに人工的であるが, 使い方の雰囲気は分かってもらえると思う. なお, トレースというのは, デバッグモードに入らずに, 指定された場所で値を表示する機能である.

```
(debug) trace T at 13
(0) trace T at "./mat_mul":13
(debug) quit
[101] mat_mul(A,A);
7
17
32
54
[ 7 17 ]
[ 32 54 ]
[102]
```

13 行目にきたら, T の値を表示するように指示した.

### 8.4.2　実行中断

ブレークポイントで止まってくれるのはまだマシな方で, いつまでたっても止まらないプログラムを書いてしまうことはよくある. 次の例は, N の階乗を計算するつもりのプログラムである.

```
def factorial(N)
{
  F = 1;
  for ( I = 1; I <= N; J++ )
    F *= I;
  return F;
}
end$
```

これを実行すると止まらない.

```
[100] factorial(3);
interrupt ?(q/t/c/d/u/w/?)
```

Ctrl-C を打ってみる. このプロンプトの意味は, ? を入力してみれば表示される. ここでは d と入力して, デバッグモードに入る.

```
(debug) where
#0 factorial(), line 5 in
"./factorial"
(debug) list factorial
1   def factorial(N)
2   {
3       F = 1;
4       for ( I = 1; I <= N; J++ )
5           F *= I;
6           return F;
7   }
8   end$
(debug) print I
I = 1
(debug)
```

ずいぶん待っているのに, I が増えていないので, 回りをじっと眺めると, I++ となるべきところが J++ となっている. 試しに J の値を見てみると

```
(debug) print J
J = 4120134
```

ととんでもないことになっている.

この例も人工的であるが, 実際にはよくあることである. ちょっと長めのプログラム中でこのような些細なミスを発見するのは, プログラムを眺めているだけでは見つかりにくいが, 中断からデバッグモード移行, という方法を使うと一発で見つかる場合が多い.

余談 (by N): (N) は Asir に限らずデバッガ依存型で, いい加減にプログラムを書いてはデバッガを頼りに修正していくという素人的プログラミングを長年続けている. 著者 (T) は, どうもそうではないらしく, どうやらソースコードをじっと眺めてたちどころにバグを見つけ出す, というプロフェッショナルなプログラミングをしているらしいが, 見たことがないので定かではない. もっとも, バグが入ったプログラムを書くようでは真のプロとは言えないという話もあるので, 五十歩百歩かもしれない.

## 8.5 章末の問題

**問題 8.3** 次のプログラムの出力値を答えよ. プログラムの実行中のメモリ内での変数の様子を説明せよ.

```
def main() {
  I = 0;
  K=tenten(I);
  print([I,K]);
}
def tenten(I) {
  I = 10;
  return(I);
}
main();
end$
```

**問題 8.4** 次のプログラムの出力値を答えよ. プログラムの実行中のメモリ内での変数の様子を説明せよ.

```
def main() {
  I = 1;
  K=tenten(I);
  print([I,K]);
}
def tenten(N) {
  I = 2*N;
  return(I);
}
main();
end$
```

**問題 8.5** 次のプログラムの出力値を答えよ. プログラムの実行中のメモリ内での変数の様子を説明せよ.

```
def main() {
  A = newvect(10);
  for (I=0; I<10; I++) A[I] = 0;
  tentenArray(A);
  print(A);
}
def tentenArray(B) {
  B[3] = 1000;
}
main();
end$
```

**問題 8.6** 摂氏をカシに変換する関数をつくり, 変換表を印刷しなさい.

**問題 8.7** [20] (この問題は unix 上のみで, `asirgui` ではできない.)

1. 文字画面上の 点 (x,y) に * を印刷する関数をつくりなさい. カーソルを (X,Y) に移動するには,
   `S=asciitostr([0x1b])+"["+rtostr(X)+";"+rtostr(Y)+"H"; print(S,0);` とすればよい. スクリーンを消去するには
   `S=asciitostr([0x1b])+"[2J"; print(S,0);` とすればよい. 0x1b (エスケープコード) で始まる文字列をエスケープシーケンスといい画面にこの文字列をおくり込むと, 画面を消去したり, 色をかえたり, カーソルを指定の位置へ移動できる. なお漢字コード等もエスケープコードを利用している. これについては第 10 章で説明する.
2. この関数をもちいて, 2 次関数のグラフを * を用いて書くプログラムをつくれ.

**問題 8.8** 長さ N の ベクトルを二つうけとり, 内積を計算する関数を書きなさい.

**問題 8.9** Asir では `diff(F,x)` で, `F` の `x` についての偏微分を計算することが可能である. `x` のあとに変数を続けて書くと, 偏微分をくりかえし行うことが可能である. たとえば, `diff(F,x,y,y)` は $\frac{\partial^3 \mathtt{F}}{\partial x \partial y^2}$ は意味する. `diff` について詳しくは Asir のマニュアルを見よ.
`diff` を用いて, あたえられた変数変換の Jacobi 行列式を計算するプログラムを作成しなさい.

**問題 8.10** Taylor 級数を用いて $\sin x$ の近似値を計算する関数を作成しなさい.

参考 (by T): 関数を有効に利用することにより, プログラムを分割して開発していくことが可能である. 過去に作成したプログラムや他人の開発したプログラムを利用しやすくなる. このような考えの一つの到達点が, オブジェクト指向プログラミングであるが, 筆者 (T) はオブジェクト指向を標榜するプログラミングを “穴埋め式プログラミング” とふざけて呼ぶことがある. 穴埋め問題というのはおなじみであるが, それのプログラム版である. 穴埋めしているうちにプログラムができてしまう. たとえば, Java AWT (Application Window Toolkit) でプログラミングするときは, あらかじめ与えらたプログラムをまさに穴埋めしながらプログラミングする. たとえば, `mouseDown` という関数を記述すれば, mouse を押したときの動作を定義できるが, 別に書かなくてもすむ. このとき `mouseDown` の” 穴を埋めた” のである.

補足 (by N): C++ の場合, 「穴」と呼ぶべきものは「仮想関数」というものであろう. この場合, 「穴埋め」= 「オーバーライド」である. 埋めるべき穴がちゃんと見えている場合は問題ないが, そもそも自分が実現したい機能が仮想関数として既に提供されているかどうかは, あるオブジェクトに関するマニュアルを隅から隅まで読んでみないとわからない場合が多い. これをさらにややこしくするのが「継承」という機能である. これは, 豊富な機能を持つオブジェクトを, 基本的なオブジェクトへの機能追加によって階層的に作りだすという操作を実現するためのものである. このため, 自分の実現したい機能があるオブジェクトのマニュアルになくても, さらに上 (下?) の階層まで遡って調べることになる. 面倒になると, 探すのをあきらめて, 自分で書いてしまったりする. こうして, C++ のスタイルに馴染めない old C プログラマ (N) は, 常に「書かなくてもいいプログラムを書いてしまったのではないか」という不安を感じながら Windows プログラミングをしているのだ.

# 第9章　常微分方程式の数値解

## 9.1　差分法

$f'(t)$ は $h$ が十分小さいとき,

$$\frac{f(t+h)-f(t)}{h}$$

で近似できる. 同じように $f''(t)$ は $h$ が十分小さいとき,

$$\frac{f(t+h)-2f(t)+f(t-h)}{h^2} \tag{9.1}$$

で近似できることをたとえば Taylor 展開を用いて証明できる. 微分をこのように近似することにより, 微分方程式の近似解を, 数列の漸化式を解くことによりもとめることが可能である.

たとえば, 単振動の方程式

$$\frac{d^2}{dt^2}y + y = 0$$

を数値的に解くことを考えてみよう. 解く前に, この方程式の物理的意味を復習しておこう. 古典物理で習うように,

$$\text{質量} \times \text{加速度} = \text{力}$$

なる関係式がなりたつ. この関係式を Newton の運動方程式という. 物体の時刻 $t$ における位置を $q(t)$ とおくと, 速度は $q'(t)$, 加速度は $q''(t)$ である.

1 次元的に単振動をするバネについた質量 1 の物体 W を考えよう. 時刻 $t$ における, $W$ の位置を $y(t)$ とすることにする. ただし, バネが自然な長さにあるとき $y=0$ とする. フックの法則によると, 自然な長さから $y$ だけのびた (負のときはちじんだとみなす) とき物体 $W$ にかかる力は $-ky$ である. ここで $k$ はバネできまる定数. ここで $k=1$ と仮定して Newton の運動方程式を適用すると, 単振動の方程式

$$y'' + y = 0$$

を得る.

$A$, $B$ を定数として一般解は $A\cos(t)+B\sin(t)$ であるが, この方程式を数値解法で解こう. 数値解法の利点は, cos や sin で解を書けないときでも, 微分方程式の近似解がわかることである. 式 (9.1) より, $h$ が十分小さいとき,

$$\frac{y(t+h)-2y(t)+y(t-h)}{h^2} + y(t)$$

は大体 0 に等しい. したがって,

$$y(t+h) = 2y(t) - y(t-h) - h^2 y(t)$$

が近似的になりたつとしてよいであろう. $Y_k = y(kh)$, $k=0,1,2,\ldots$ で数列 $Y_k$ を定める. このとき,

$$Y_{k+2} = 2Y_{k+1} - Y_k - h^2 Y_k$$

なる漸化式が近似的になりたつとしてよい. したがって, 漸化式

$$y_{k+2} = 2y_{k+1} - y_k - h^2 h_{k+1} \tag{9.2}$$

を満たす数列を決めることにより, 解の近似をもとめることが可能であると予想できる. このように解の近似を求める方法を差分法とよぶ. 漸化式 (9.2) を元の微分方程式の差分化, または差分スキームとよぶ.

数列の値を決めるには初期条件が必要であるが, それは, 微分方程式の初期条件 $y(0) = a, y'(0) = b$ を用いて関係式

$$y(0) = a = y_0,\ y'(0) = b \simeq \frac{y_1 - y_0}{h}$$

できめてやればよい.

**例題 9.1** 強制振動の方程式

$$\frac{d^2}{dt^2}y + y = \sin(at)$$

を数値的に解いてみよう. $\sin(at)$ が外部から単振動する物体に与えられる力である.

```
load("glib")$

def osci() {
  glib_window(0,-5,50,5);
  glib_clear();
  glib_line(0,0,50,0);
  glib_line(0,-10,0,10);

  X1 = 0.5; X2 = 0.501; A=0.5;

  Dt = 0.07; T = 0;
  while (T<50) {
     X3 = 2*X2-X1+Dt*Dt*(eval(sin(A*T))-X2);
     glib_putpixel(T,X1);
    /*     print([T,X1]); */
     T=T+Dt;
     X1=X2; X2=X3;
  }
}

print("Type in osci()")$
end$
```

これで, 微分方程式を近似的に解く問題が, 漸化式をみたす数列を求める問題になったのであるが, このような近似解が本当の解に収束するかとか, 全くことなる解しかとらえられない不安定現象が起こる場合があるとかの議論をやらないといけない.

## 9.2 不安定現象

差分法をもちいると時々本来の解からは似ても似つかない解を得ることがある. そのような例を一つ示そう.

方程式

$$\frac{d}{dt}y = -2y, \quad y(0) = 1$$

を考える. 本来の解は $e^{-2t}$ である.

$$\lim_{h\to 0}\frac{f(t+h)-f(t-h)}{2h} = f'(t)$$

であるので, $h$ を十分小さい数とするとき, 次のような差分スキーム (差分方程式) で近似解をもとめてもいいであろう.

$$y_{k+2} = y_k - 4hy_{k+1}, \quad y_0 = 1,\ y_1 = y_0 + (-2)h$$

ここで, 本来の解の原点での微分係数が $-2$ なので, この事実を用いて, $y_1$ をきめている. このとき数列 $y_k$ は, 時刻 $hk$ での真の解の値を近似していると予想できる.

しかるに, プログラム `unstable.rr` (図 9.1) の出力がしめすように, $k$ の値が小さいときは真の解を近似してるが, $k$ の値が大きくなると, 解は振動をはじめ最後には大きな数に爆発してしまう. このような現象を差分法の誤差の爆発現象 (blow up of errors of a difference scheme) という.

`unstable.rr` の出力. 時間と真の解との誤差を表示している.

```
      略
[Time, (approx sol)-(true sol)] : [0.29,-0.000212873]
[Time, (approx sol)-(true sol)] : [0.3,0.000149232]
[Time, (approx sol)-(true sol)] : [0.31,-0.000217378]
[Time, (approx sol)-(true sol)] : [0.32,0.000159362]
      略
[Time, (approx sol)-(true sol)] : [6.86,90.7806]
[Time, (approx sol)-(true sol)] : [6.87,-92.6143]
[Time, (approx sol)-(true sol)] : [6.88,94.4851]
[Time, (approx sol)-(true sol)] : [6.89,-96.3937]
The solution is blown up.
```

このように, $h$ の値を決めたとき, $k$ が小さいうちは差分法の近似解が真の解に近いが, $k$ が大きくなったら, 真の解の近似を与える保証はない. $h$ の値を決めたとき, $k$ がある範囲におさまるうちは差分法が真の解に十分近い解を与えることを次の節で証明しよう.

一般に, 差分法の解で真の解を大きい $k$ まで近似するのは簡単ではない. 前節で紹介した単振動の方程式の差分化は十分大きい $k$ に対しても解が近似になっているような例である. 大きい $k$ まで近似するには, 差分法の解ともとの微分方程式の定性的性質がにている必要がある. 保存量等があるとき元の方程式と差分化した漸化式が同じような保存則をみたしている等の性質があれば, 定性的性質が同じになる可能性も高いが, 厳密な数学的解析が必要である.

## 9.3 解の収束定理

**定理 9.1** 微分方程式

$$y'(t) = f(t, y(t)), \quad y(0) = P$$

を考える. ある数 $L$ が存在して, 任意の, $u, v \in \mathbf{R}$ および $t \in [0, b]$ に対して $f(t, z)$ はリプシッツ連続の条件

$$|f(t, u) - f(t, v)| < L|u - v|$$

```
load("glib")$

def un() {
  glib_window(0,-5,10,5);
  glib_clear();
  glib_line(0,0,10,0);
  glib_line(0,-10,0,10);

  X1 = 1.0;
  Dt = 0.01; T = 0.0;
  X2 = 1.0-2*Dt;
  while (T<50) {
     X3 = X1 - 4*Dt*X2;
     glib_line(T,X1,T+Dt,X2);
     print("[Time, (approx sol)-(true sol)] : ",0);
     print([T,X1-deval(exp(-2*deval(T)))]);
     T=T+Dt;
     X1=X2; X2=X3;
     if (X3 > 100 || X3 < -100) {
        print("The solution is blown up.");
        break;
     }
  }
}

print("Type in un()")$
end$
```

図 9.1: プログラム `unstable.rr`

をみたすと仮定する. さらに, 2 回連続微分可能な解 $y(t)$ が $t \in [0, b]$ の間で存在すると仮定する. $n$ を数, $h = b/n$ とおくとき数列 $\{y_k\}$ を漸化式

$$y_{k+1} = y_k + hf(x_k, y_k), \quad y_0 = P$$

で定義する. このとき, $n$ に関係しないある数 $C$ が存在して,

$$\max_{0 \le k \le n} |y_k - y(kh)| \le Ch \tag{9.3}$$

となる.

定理の意味を説明しよう. $C$ は $n$ に依存しないので, 式 (9.3) の右辺は, $n = b/h \to \infty$ のとき 0 に収束する. したがって, $n$ が十分大きいとき (すなわち $h$ が十分小さいとき), 差分法で求まる数列 $y_k$ は真の解 $y(kh)$ に $0 \le kh \le b$ の区間で十分近い.

定理の証明をしよう. $y(t)$ を真の解とし, $t_k = kh$, $Y_k = y(t_k)$ とおく. $y(t_k + h)$ に Taylor 展開の公式を 2 次まで適用すると, 等式

$$y(t_k + h) = y(t_k) + y'(t_k)h + \frac{h^2}{2!}y''(t_k + \theta_k(h)h) \tag{9.4}$$

が成立する. ここで, $\theta_k(h)$ は $k$ と $h$ に依存し, $0 < \theta_k(h) < 1$ を満たす定数である. (9.4) を微分方程式を用いてかきなおすと,

$$Y_{k+1} = Y_k + f(t_k, Y_k)h + \frac{h^2}{2!}y''(t_k + \theta_k(h)h)$$

となる. 差分方程式の方の解 $y_k$ と差をとると,

$$y_{k+1} - Y_{k+1} = y_k - Y_k + hf(t_k, y_k) - hf(t_k, Y_k) - \frac{h^2}{2!}y''(t_k + \theta_k(h)h)$$

を得る. よって不等式

$$|y_{k+1} - Y_{k+1}| \le |y_k - Y_k| + hL|y_k - Y_k| + h^2M \tag{9.5}$$

を得る. ここで $f$ のリプシッツ連続性および, 区間 $[0, b]$ で $y''(t)/2!$ が有界であり, ある定数 $M$ で上からおさえられることを用いた. (9.5) の右辺の $|y_k - Y_k|$ を $k$, $k-1$ に対する場合の不等式 (9.5) で上からおさえることにより,

$$|y_{k+1} - Y_{k+1}| \le (1 + hL)^2|y_{k-1} - Y_{k-1}| + (1 + hL)h^2M + h^2M$$

をえる. これを繰り返して,

$$\begin{aligned} & |y_{k+1} - Y_{k+1}| \\ \le\ & (1 + hL)^{k-1}|y_0 - Y_0| + \left\{(1 + hL)^{k-2} + \cdots + (1 + hL) + 1\right\} h^2L \\ =\ & \frac{1 - (1 + hL)^{k-1}}{1 - (1 + hL)}h^2M \\ =\ & \frac{1 - (1 + hL)^{k-1}}{L}hM. \end{aligned}$$

ここで

$$|1 - (1 + hL)^{k-1}| \le 1 + \frac{(1 + hL)^n}{1 + hL} \le 1 + (1 + hL)^n = 1 + \left(1 + \frac{bL}{n}\right)^n \le 1 + e^{bL}$$

なので, $C = \frac{1+e^{bL}}{L}M$ とおくと, 定理の不等式がみたされる. 証明おわり.

同様の差分化および証明法は連立常微分方程式

$$\frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \\ \cdot \\ y_q \end{pmatrix} = \begin{pmatrix} f_1(t, y_1, \ldots, y_q) \\ f_2(t, y_1, \ldots, y_q) \\ \cdot \\ f_q(t, y_1, \ldots, y_q) \end{pmatrix}$$

でも通用する.

## 9.4　例

以下は常微分方程式を数値的に解く例である. どのような差分スキームをもちいているか, プログラムを読んで述べよ.

**例題 9.2**

$$\frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

を数値的に解く.

```
load("glib")$
def diff1(X1,X2) {
   glib_open();
   glib_window(-10,-10,10,10);
   glib_line(-10,0,10,0);
   glib_line(0,-10,0,10);
   Dt=deval(0.001); T=deval(0.0);
   while (T < 10) {
        Y1=X1+Dt*X2;
        Y2=X2-Dt*X1;
        glib_putpixel(X1,X2);
        T=T+Dt;
        X1=Y1; X2=Y2;
   }

}

print("Type in, for example, diff1(2,3); ")$
end$
```

**例題 9.3**

$$\frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & -0.1 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

を数値的に解く.

```
load("glib")$
def diff2(X1,X2) {
   glib_open();
   glib_window(-10,-10,10,10);
   glib_line(-10,0,10,0);
   glib_line(0,-10,0,10);
   Dt=deval(0.001); T=deval(0.0);
   while (T < 50) {
        Y1=X1+Dt*X2;
        Y2=X2+Dt*(-X1-0.1*X2);
        glib_putpixel(X1,X2);
        T=T+Dt;
        X1=Y1; X2=Y2;
   }

}

print("Type in, for example, diff2(2,3); ")$
end$
```

**例題 9.4** 強制振動の方程式

$$\frac{d^2}{dt^2}y + y = \sin(at)$$

を数値的に解いてみよう. $\sin(at)$ が外部から単振動する物体に与えられる力である. この例題は, 5 章においてすでに紹介した.

このように, 微分方程式を数値的に解く事を数値解法という. この方法は, ゲームから, LSI の設計, 飛行機の設計までさまざまな場面で利用されている.
最後は カオス的力学系の例である.

**例題 9.5** Lorentz 方程式

$$\begin{aligned} p_1' &= -ap_1 + ap_2 \\ p_2' &= -p_1p_3 + bp_1 - p_2 \\ p_3' &= p_1p_2 + cp_3 \end{aligned}$$

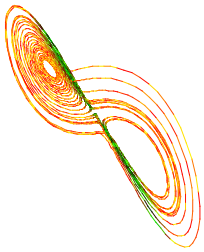
の解を眺めてみよう. ここで $'$ は $t$ についての微分をあらわす. $A, B, C$ はパラメータである.

```
load("glib")$

def lorentz() {
  glib_window(-25,-25,25,25);
  A=10; B=20; C=2.66;
  P1=0; P2 = 3; P3 = 0;
  Dt = 0.004; T = 0;
  while (T <50) {
      Q1=P1+Dt*(-A*P1+A*P2);
      Q2=P2+Dt*(-P1*P3+B*P1-P2);
      Q3=P3+Dt*(P1*P2-C*P3);
      glib_putpixel(Q1,Q2);
      T=T+Dt;
      P1=Q1; P2=Q2; P3=Q3;
  }
}

end$
```

画面に描き出される次のような不思議な軌道に感動していただきたい.



**章末問題**

1. 減衰振動の方程式 $y'' + ay' + y = 0$ を数値的に解きなさい. パラメータ $a$ を変えるとどうなるか?
2. $\sin(kx)$ および $\cos(kx)$ を組み合わせて面白いグラフを作ろう.
3. Lorentz 方程式の解を, パラメータ, 初期値を変えて観察しよう.
4. 連立常微分方程式の場合に差分法の解の収束証明をしなさい.
5. 差分法の解の収束定理において, 誤差が $Ch$ でなく $C'h^2$ となるような差分スキームを考えなさい (2 次のルンゲクッタ法).

# 第10章　入出力と文字列の処理, 文字コード

## 10.1　文字コード

### 10.1.1　アスキーコード

文字をどのようにして 2 進数として表現するかは, 規格が決められている. アルファベットについてはアスキーコードが標準としてつかわれている. アスキーコードでは 7 ビットを使用してコードを表現する. 次の表がその対応表である.

| | | | | | |
|---|---|---|---|---|---|
| 20 | | 40 | @ | 60 | ‘ |
| 21 | ! | 41 | A | 61 | a |
| 22 | " | 42 | B | 62 | b |
| 23 | # | 43 | C | 63 | c |
| 24 | $ | 44 | D | 64 | d |
| 25 | % | 45 | E | 65 | e |
| 26 | & | 46 | F | 66 | f |
| 27 | ’ | 47 | G | 67 | g |
| 28 | ( | 48 | H | 68 | h |
| 29 | ) | 49 | I | 69 | i |
| 2a | * | 4a | J | 6a | j |
| 2b | + | 4b | K | 6b | k |
| 2c | , | 4c | L | 6c | l |
| 2d | - | 4d | M | 6d | m |
| 2e | . | 4e | N | 6e | n |
| 2f | / | 4f | O | 6f | o |
| 30 | 0 | 50 | P | 70 | p |
| 31 | 1 | 51 | Q | 71 | q |
| 32 | 2 | 52 | R | 72 | r |
| 33 | 3 | 53 | S | 73 | s |
| 34 | 4 | 54 | T | 74 | t |
| 35 | 5 | 55 | U | 75 | u |
| 36 | 6 | 56 | V | 76 | v |
| 37 | 7 | 57 | W | 77 | w |
| 38 | 8 | 58 | X | 78 | x |
| 39 | 9 | 59 | Y | 79 | y |
| 3a | : | 5a | Z | 7a | z |
| 3b | ; | 5b | [ | 7b | { |
| 3c | < | 5c | \ | 7c | \| |
| 3d | = | 5d | ] | 7d | } |

```
3e   >      5e   ^      7e   ~
3f   ?      5f   _      7f
```

20H が空白, 20H 未満は制御コードと呼ばれている. たとえば, unix では 0AH が改行を表すために使われている. 制御コードの意味は OS によってことなる. MSDOS や Windows では 0DH, 0AH の 2 バイトが改行を表すために, Macintosh では 0DH が改行を表すために利用されている.

**問題 10.1 (05)**　次の文章をアスキーコードで表すとどうなるか？

Do not cry over the spilt milk.

**問題 10.2 (05)**　次はどんな文章か?

```
46 72 65 65 42 53 44 20 33 2e 33 2d 52 45 4c 45
41 53 45 20 28 47 45 4e 45 52 49 43 29 20 23 30
3a 20 53 61 74 20 4a 61 6e 20 32 39 20 30 39 3a
34 33 3a 34 39 20 4a 53 54 20 32 30 30 30 0a
```

### 10.1.2　漢字コードと ISO2022

漢字については, JIS コード, シフト JIS コード, EUC コードなどがある. (ねじの大きさと同じでれっきとした JIS 規格がある. 本屋さんへいって JIS の規格書 の情報編をみてみよう. ) JIS コード, EUC コードは 国際的なコード系の規約である, ISO-2022 に準拠している.

シフト JIS コードは MSDOS パーソナルコンピュータや Windows マシン, Macintosh などで広く利用されているが, その設計は多くの言語が混在するなどの状況を考えて設計されておらず, また現在ではあまり利用されない俗にいう “半角カナ” が頻繁に利用されていた 1980 年代始めの時代状況をひきづっており, 長い目でみたときには規格として耐えうる設計とはいえない. したがって国際規格としては認められていない. Web ブラウザ等の文字化け現象の一因はシフト JIS コードが広まってしまったことにも一因がある. Windows マシンなどは将来的には シフト JIS を漸進的に捨てて, Unicode を利用しようとしているようにみえる.

日本語 EUC コードは unix 系のコンピュータでおもに使われているコード系であり, 漢字は 2 バイトつまり１６ビットを使用して表現する. ここでは EUC コード表の一部をあげる.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| b1a1 | 院 | b1a3 | 隠 | b1a4 | 韻 | b1a5 | 吋 |
| b1a6 | 右 | b1a7 | 宇 | b1a8 | 烏 | b1a9 | 羽 |
| b1aa | 迂 | b1ab | 雨 | b1ac | 卯 | b1ad | 鵜 |
| b1ae | 窺 | b1af | 丑 | b1b0 | 碓 | b1b1 | 臼 |
| b1b2 | 渦 | b1b3 | 嘘 | b1b4 | 唄 | b1b5 | 欝 |
| b1b6 | 蔚 | b1b7 | 鰻 | b1b8 | 姥 | b1b9 | 厩 |
| b1ba | 浦 | b1bb | 瓜 | b1bc | 閏 | b1bd | 噂 |

では国際的な文字コード規格である, ISO 2022 について簡単に説明しよう. ISO-2022 について詳しくは [1] を参照. ISO-2022 (拡張版) では 21H から 7EH を GL 領域, A1H から FEH を GR 領域とよぶ. ISO-2022 では, 1BH から始まるエスケープシーケンスとよばれるコードを用いて, GL および GR 領域に各国においてさだめられたコードをはりつける. コードの張り付けには G0, G1, G2,

G3 という中間的な経由張り付け領域を経由しておこなうことになっているがここではふれない. [1] を参照.

たとえば, 1BH, 2DH, 42H, 1BH, 7EH なるエスケープシーケンスは, ISO 8859-2 (通称 ISO Latin-2) を GR 領域にはりつけよという指示である. たとえば, 文字 ö は ISO 8859-2 では F6H なるコードをわりあてられている. 例としてあげると,

1BH, 2DH, 42H, 1BH, 7EH, (ISO 8859-2 を GR へ) F6H, F6H, F6H ( ö が３つ)

は ö が 3 つを意味する.

ISO 2022-JP では, GL には (G0 を経由して) JIS X0208 ローマ字またはアスキー文字, または JIS X 0208 の漢字コード (73 年版, 83 年版などあり) をエスケープシーケンスで切替えながら表示する. たとえば

1BH, 24H, 42H (JIS X 0208 83 年版へ), 31H, 39H (厩), 1BH, 28H, 42H (アスキーコードへ), 41H, 42H, 43H

は “厩 ABC” という文字列となる.

日本語 EUC コードは GL に アスキーコードを, GR に JIS X0208 漢字コードを常によびだした状態のコードである. GR に JIS X0208 をよびだした場合には, JIS0208 の最上位ビットを 1 にする. したがって, “厩 ABC” を 日本語 EUC コードで表すと,

B1H, B9H (厩), 41H, 42H, 43H

となる. 実際, 31H は 2 進数では, 0011 0001 なので, 最上位ビットを 1 にすると, 1011 0001 となり, 16 進数では B1H である.

JIS コード, shift JIS コード, 日本語 EUC コード間の変換をおこなうプログラムとして, unix では `nkf` などのプログラムがある.

**問題 10.3 (05)** 次の文章を ISO-2022-jp および シフト JIS コードになおしなさい.

たびにやんでゆめはかれのをかけめぐる. Basho 作.

```
------------------------------------------------------
EUC (Extended Unix Coding system) によるひらがな.

a4a2 a4a4 a4a6 a4a8 a4aa 200a
あ　い　う　え　お
a4ab a4ad a4af a4b1 a4b3 200a
か　き　く　け　こ
a4b5 a4b7 a4b9 a4bb a4bd 200a
さ　し　す　せ　そ
a4bf a4c1 a4c4 a4c6 a4c8 200a
た　ち　つ　て　と
a4ca a4cb a4cc a4cd a4ce 200a
な　に　ぬ　ね　の
a4cf a4d2 a4d5 a4d8 a4db 200a
は　ひ　ふ　へ　ほ
a4de a4df a4e0 a4e1 a4e2 200a
```

```
ま　み　む　め　も
a4e4 a4a4 a4e6 a4a8 a4e8 200a
や　い　ゆ　え　よ
a4e9 a4ea a4eb a4ec a4ed 200a
ら　り　る　れ　ろ
a4ef a4a4 a4a6 a4a8 a4aa 200a
わ　い　う　え　お
a4f3 a1ab a1a3 0a0a
ん　゛　。

-------------------------------------------------
SJIS (Shifted JIS Code) によるひらがな.

82a0 82a2 82a4 82a6 82a8 200a
あ　い　う　え　お
82a9 82ab 82ad 82af 82b1 200a
か　き　く　け　こ
82b3 82b5 82b7 82b9 82bb 200a
さ　し　す　せ　そ
82bd 82bf 82c2 82c4 82c6 200a
た　ち　つ　て　と
82c8 82c9 82ca 82cb 82cc 200a
な　に　ぬ　ね　の
82cd 82d0 82d3 82d6 82d9 200a
は　ひ　ふ　へ　ほ
82dc 82dd 82de 82df 82e0 200a
ま　み　む　め　も 8
2e2 82a2 82e4 82a6 82e6 200a
や　い　ゆ　え　よ
82e7 82e8 82e9 82ea 82eb 200a
ら　り　る　れ　ろ
82ed 82a2 82a4 82a6 82a8 200a
わ　い　う　え　お
82f1 814a 8142 0a0a
ん　゛　。
```

### 10.1.3　全角文字と￥記号

Emacs などのエディタで 1 をそのまま入力した場合と, 1 を入力したあとかな漢字変換してでてくる １ では異なることがわかるであろうか？

1　　１

上で, 前者の 1 が半角の 1, 後者の １ が全角の 1 である.

**問題 10.4** 半角の 1, 後者の １ が全角の 1 を Emacs で入力してみて, Emacs の上でカーソルを移動していくと, 点滅しているカーソルの大きさが全角と半角で異なることをたしかめよ. また全角の 1 は半角の 1 の 2 倍の文字幅をもつことをたしかめよ. Windows の場合メモ帳 (notepad) などのテキストエディタを用いると確かめることができるが, ワープロソフトの場合は文字のフォントのデザインで幅がきまるので, 全角の文字幅が半角の 2 倍とは限らない.

半角の 1 と全角の 1 はことなる文字であり, 対応する文字コードは半角 1 に対してはアスキーコードの 31H, 全角 1 には JIS 漢字コードの 23H 31H が対応している. なお, 23H 31H は EUC コードでは A3H B1H, Shift JIS コードでは 82H 50H である. アスキーコード表にある英数字や多くの特殊記号には対応する文字の全角版が JIS 漢字コードに存在している. これらは対応するアスキー文字とはことなるものである. したがってたとえば, 電子メールアドレス `hoge@math.kobe-u.ac.jp` を全角文字で ｈｏｇｅ@ｍａｔｈ．ｋｏｂｅ－ｕ．ａｃ．ｊｐと書くとアドレスエラーになるし, プログラムを全角文字でかくともちろんエラーになる. 空白文字にも半角空白 (アスキーコードの 20H) と全角空白 (JIS コードの 21H 21H) があり時々トラブルの原因になる.

JIS X0201 規格は日本の文字コードの基本である. LR 領域に対応する方はアスキーコードとほぼ同一であるが, アスキーコードの \ が JIS X0201 では ¥ 記号になっている. C 言語や TeX の教科書で同じものに対して, \ と ¥ 両方の書き方があるのはそのせいである.

## 10.2 入出力関数

Asir では, キーボード入力を取り扱うため, `purge_stdin()` と `get_line()` なる関数を用意している. たとえば, キーボードから読み込んだ数字の 2 倍を表示する関数は次のようになる. この関数は, 単に RETURN だけの入力で終了する. また入力された文字列のアスキーコードも画面に表示する.

```
def nibai() {
   S = " ";
   while (strtoascii(S)[0] != 10) {
      purge_stdin();
      S = get_line();
      print(strtoascii(S));
      A = eval_str(S);
      print(2*A);
   }
}
```

`strtoascii(S)` は文字列 `S` を対応するアスキーコードのリストに変換する関数である. RETURN キーのみを入力した場合には, `S` には 0xA (改行コード) のみがはいる. `eval_str(S)` は文字列 `S` を Asir のコマンドとして評価する. この関数の逆関数は `rtostr` である.
たとえば, `P=rtostr((x+1)^2);` とすると, `P` には文字列としての `x^2+2*x+1` がはいる.

関数 `open_file()`, `get_byte()`, `close_file()`, `put_byte()` を用いるとファイルの読み書きができる. ファイルはまず open (ひらく) してから, 読まないといけない. open して, そのファイルを識別する番号を戻すのが, 関数 `open_file(S)` である. `S` にはファイル名を文字列として与える. ファイルが存在しないなど open に失敗すると, エラーでとまる. 成功した場合は, 0 以上の数字を戻す. これがファイル識別子 (file descriptor) である. 関数 `get_byte(`ファイル識別子`)` は, 指定されたファイルより 1 バイト読み込む. 関数 `close_file(`ファイル識別子`)` でファイルの利用終了を宣言する. ファイルへの書き込みについては, マニュアルを参照されたい. 利用例は 10.4 節をみよ.

## 10.3　文字列の処理をする関数

関数 asciitostr(), strtoascii() をもちいることにより, アスキーコードと文字列の変換が可能である.
例:

```
[346] asciitostr([0x41]);
A
[347] strtoascii("abc");
[97,98,99]
```

文字列の結合は + 演算子を用いる. たとえば "abc"+"ABC" は abcABC を戻す.

**問題 10.5** (C 言語を知ってる人向け). カーニハン, リッチーによる有名な本 “C プログラミング言語” の第 1 章で議論されているファイル入出力に関するプログラムを Asir で記述せよ.

## 10.4　ファイルのダンプ

ファイルはディスクの中に格納されている 1 バイトデータの一次元的な列に名前をつけたものである. プログラム, 図形データや文書はファイルとして格納される.

次のプログラム dump.rr はファイルの中身をバイトデータの列として表示するプログラムである. こういったことをやるプログラムをダンププログラムという. このプログラムでは次の二つの関数を定義している.

1. toHex(N) : 整数 N を 16 進法の文字列に直して戻す.
2. dump(F) : ファイル F をダンプする.

まずは toHex の定義.

```
extern HexTab$
HexTab=newvect(16,
 ["0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f"])$

def toHex(N)
{
  return HexTab[ishift(N,4)]+HexTab[iand(N,0xf)];
 /* return HexTab[idiv(N,16)]+HexTab[N%16]; */
}
```

extern 宣言した変数は局所変数としてあつかわれない. すべての関数の中から共通の変数として参照可能となる. 関数 ishift については, 11 章を参照. iand(A,B) は A および B を 2 進数で表記したときの, bit 毎 (桁毎) の and の結果を戻す. 関数 toHex(N) は N ($0 \leq$ N $\leq 255$) を 16 進表記に直す. なおこの関数は, ishift や iand を使わないでコメントにあるように書いてもよい.
つぎに dump の定義.

```
def dump(FileName) {
   Fp = open_file(FileName);
   if (Fp < 0) error("Open failed.");
   for ( I = 1; (C=get_byte(Fp)) >= 0; I++ ) {
     print(toHex(C),0);
      if ( !(I%16) )
        print("");
      else
        print(" ",0);
   }
  /* XXX */
  if ( (I-1)%16 )
    print("");
}
end$
```

**例題 10.1**　ファイル dump.rr のダンプをとってみなさい.

**入力例 10.1**

```
[346] load("dump.rr");      (Windows では ‘‘開く’’ で読み込む)
1
[352] dump("dump.rr");      ファイル dump.rr を 16 進数列で表示.
(Windows では dump("c:/dump.rr"); で c ドライブ直下のファイル dump.rr を読める)
20 20 0a 20 20 65 78 74 65 72 6e 20 48 65 78 54
61 62 24 0a 20 20 48 65 78 54 61 62 3d 6e 65 77
76 65 63 74 28 31 36 2c 0a 20 20 09 5b 22 30 22
2c 22 31 22 2c 22 32 22 2c 22 33 22 2c 22 34 22
2c 22 35 22 2c 22 36 22 2c 22 37 22 2c 22 38 22
2c 22 39 22 2c 22 61 22 2c 22 62 22 2c 22 63 22
         以下略
```

20 20 は空白 0a は改行, 20, 20 は空白, 65, 78, 74, 65, 72, 6e は extern である.

## 10.5　章末の問題

問題 1:
(a) 次のプログラムの出力値を答えよ. ただし 'A' のアスキーコードは 0x41 (16 進数の 41) である.

```
def main() {
   print(strtoascii("A"));
   N = strtoascii("B")[0]-strtoascii("A")[0];
   print(N);
}
```

(b) 0x41 を 10 進数になおすといくつか?

問題 2:
0, ..., 9 のアスキーコードはいくつかしらべるプログラムを書け.

問題 3:
次のプログラムの出力を答えよ.

```
def main() {
  A = strtoascii("105 cats");
  Ndigit = newvect(10);
  Nother = 0;
  for (I=0; I<10; I++) {
    Ndigit[I] = 0;
  }
  Zero = strtoascii("0")[0];
  Nine = strtoascii("9")[0];
  for (I=0; I<8; I++) {
    C = A[I];
    if ( C >= Zero && C <= Nine) {
      Ndigit[C-Zero] = Ndigit[C-Zero]+1;
    }else{
      Nother++;
    }
  }
  print("Nother=",0); print(Nother);
  print(Ndigit);
}
```

問題 4:
次のプログラムの出力値を答えよ.

```
def main() {
  A = newvect(10);
  A[0] = strtoascii("A");
  A[1] = 0x41;
  A[2] = 0x61;
  A[3] = 0x20;
  A[4] = strtoascii(".");
  A[5] = 0;
  A[6] = strtoascii("A");
  for (I=0; I<7; I++) {
    print(A[I],0); print(" ");
  }
  print("");
  print(asciitostr(A));  /* This generates an error for now. */
}
```

問題 5:
次のプログラムの出力を書きなさい.

```
def strToUpper(T) {
  S = strtoascii(T);
  /* print(S); */
  S = newvect(length(S),S);
  for (I=0; I < size(S)[0]; I++) {
   if (S[I] >= 0x61 && S[I] < 0x7f) {
     S[I] = S[I] - 0x20;
   }
  }
  return(asciitostr(vtol(S)));
}
def main() {
  A = "Hello World";
  B = strToUpper(A);
  print(B);
}
```

問題 6:
適当なファイルを選んで, それがどのようにファイルとして格納されているのか dump.rr で調べよ. テキストファイル や メールのファイル (初級コース), ペイントの作成する BMP ファイル (中級コース).

問題 7 (研究課題):
ローマ字, かな変換をおこなうプログラムを作成せよ.

Risa/Asir ドリル ギャラリー：ハードディスク (Harddisks). 大きい方はデスクトップコンピュータ用, 小さい方はラップトップコンピュータ用.
テキストファイルはハードディスクに文字コードを用いて格納されている.

# 関連図書

[1] 安岡孝一, 安岡素子, 文字コードの世界, 東京電機大学出版局, 1999.
世界の文字コードについての解説書. ISO-2022 について詳しくはこの本をみよ.

[2] B.W.Kernighan, D.M.Ritchie, C Programming Language (2nd Edition), Prentice Hall, 1988.
日本語訳: プログラミング言語 C (第 2 版), 共立出版
プログラミング言語 C のもっとも基本的かつ重要な文献. 世界的なベストセラーでもある. この本ではさまざまな例題とともに文字列処理も解説している.

Risa/Asir は C の文法と似た文法でプログラムを記述する. Maple や Mathematica などの数式処理システムは独自の言語でプログラムを記述しないといけない. 習得には少々時間がかかる. それに比較して, Risa/Asir は C や Java を知ってる人は, すぐプログラムが書ける. また, C や Java を知らない人は, 簡単にプログラムがためせる Risa/Asir で練習してから, C や Java を覚えると習得が早いであろう.

# 第11章　数の内部表現

## 11.1　peek と poke

関数 peek(D) を用いると, アドレス D に格納された 1 バイト (16 進 2 桁, 2 進 8 桁) のデータを覗く (peek) ことができる. 関数 poke(D,N) を用いると, アドレス D にデータ N を直接書き込む事が可能である. ここで, N は 0 から $2^8 - 1 = 254 =$ 0xff の範囲のデータである.

通常のプログラムでこのような関数を利用する必要はないが, 計算機プログラミングの達人になるには これらの関数を援用して, メモリの様子が手にとるように理解できるようになるまで頑張る必要があろう.

なお, poke であやまった場所にデータを書き込んでしまうと, システム全体が異常な動作におちいるので注意. どうしてそのような事態になるか説明できるだろうか？

**例 11.1** 文字列はその文字のアスキーコードがそのままメモリーに格納されている. このことを実例で確かめてみよう.

```
[469] X="abc";
abc
[470] D=get_addr(X);
139968704
[471] hex_dump(D,10);
0857c0c0: 07000000 98aa5308 0000
0
[472] type(X);
7
[473] hex_dump(0x0853aa98,10);
0853aa98: 61626300 00000000 656e
0
```

1. 変数 X に 文字列 abc を代入.
2. 変数 X の実体が存在するアドレスを get_addr で取り出し, 変数 D にいれる.
3. D から 10 バイト分の領域に何があるか hex_dump で調べてみる.
4. 07 00 00 00 の 07 は文字列を表すタグ (印).
5. 98 aa 53 08 は アドレス 08 53 aa 98 (後ろから読む) を意味する. ここにデータの実体が存在. Asir では 0x ではじまる数字および a から f の列は 16 進数を表す.
6. アドレス 08 53 aa 98 から 10 バイト分の領域をしらべると, 61 62 63 00 なるデータが存在しており, 順番に a, b, c のアスキーコードである.

上の例では, hex_dump の出力 98 aa 53 08 より アドレス 08 53 aa 98 (後ろから読む) を読みとり, 手動でアドレスを入力する必要があった. 次の関数 get_body_addr はこのアドレスを一気に求める.

```
def get_body_addr(X) {
  A = get_addr(X);
  A3=peek(A+7);
  A2=peek(A+6);
  A1=peek(A+5);
  A0=peek(A+4);
  return(ishift(A3,-24)+ishift(A2,-16)+ishift(A1,-8)+A0);
}
```

最後の ishift(A3,-24)+ishift(A2,-16)+ishift(A1,-8)+A0 は
A3*0x1000000+A2*0x10000+A1*0x100+A0 としてもよい. これはたとえば 2 進数を $-8$ bit 左にシフトすることと, その数を $2^8$ 倍 $(2^8 = 0x100 = 256)$ するのは同じことなので明らかであろう.

**例 11.2** get_body_addr を用いて, メモリに格納された文字列を直接書き換えよう.

```
[469] X="abc";
abc
[470] D=get_addr(X);
139968704
[471] hex_dump(D,10);
0857c0c0: 07000000 98aa5308 0000
0
[473] get_body_addr(X);
139700888
[474] poke(get_body_addr(X),0x41);
0
[475] X;
Abc
```

1. 変数 X に 文字列 abc を代入.
2. 変数 X の実体が存在するアドレスを get_addr で取り出し, 変数 D にいれる.
3. D から 10 バイト分の領域に何があるか hex_dump で調べてみる.
4. アドレス 08 53 aa 98 を 10 進数になおすと, 139700888.
5. poke コマンドで, このアドレスに直接 0x41 (A のアスキーコード) を書き込む.
6. X の値は Abc にかわった.

アドレス get_addr(X) より始まるメモリ領域の最初のバイトは X に格納されているデータの型番号である. この数は, 関数 type(X) の戻す値と一致している. たとえば次の例をみてみよう.

```
[496] X=10;
10
[497] type(X);
1
[498] hex_dump(get_addr(X),4)$
085843b0: 01000001
[499] X=x^2-1;
x^2-1
[500] type(X);
2
[501] hex_dump(get_addr(X),4)$
085840c0: 02000000
```

C 等の一般的な言語での整数は, “32 bit 整数” とよばれ, サイズが 32 bit である. 一番上位の bit を符号としてもちいるので, 扱うことのできる最大の正の整数は $2^{31}-1 = 2147483647$ となる. C でこの数に 1 を加えると負の数となる.

Asir が標準的に用いている整数は , “32 bit 整数” でない. いわゆる bignum である. 数が大きくなっていったら動的に数のデータを格納するメモリの領域を広げて計算をおこなう. したがって, 数の大きさの上限は, 計算機のメモリのサイズに依存するのみである. この様子を, 次のプログラムで眺めてみよう.

```
def naibu() {
  X = 2^16;
  for (I=0; I<4; I++) {
    print("X=",0); print(X);
    A = get_body_addr(X);
    print("address=",0); print(A);
    hex_dump(A,32);
    X = X*X;
  }
}
```

1. X には, 順に $2^{16}$, $2^{32}$, $2^{64}$, $2^{128}$ がはいる.
2. アドレス A より 32 バイトのメモリを見る.

```
[532] naibu()$
X=65536
address=140075328
08596140: 01000000 00000100
          00000000 00000000
08596150: 01000000 00010000
          00000000 00000000

X=4294967296
address=140075040
08596020: 02000000 00000000
          01000000 00000000
08596030: 02000000 40806302
          01000000 00000000
```

1. 00 00 01 00 を逆順にならべると 00 01 00 00 となり, これを 16 進数とみなすと, $16^4 = 2^{16} = 65536$ となる.
2. 00 00 00 00 01 00 00 00 を逆順にならべると 00 00 00 01 00 00 00 00 となり, これを 16 進数とみなすと, $16^8 = 2^{32} = 4294967296$ となる.

以下同様である. address の値も時時刻刻と変化しており, 計算の結果が新しいメモリ領域に格納されている様子もわかるであろう.

```
X=18446744073709551616
address=139038512
08498f30: 03000000 00000000
          00000000 01000000
08498f40: 00000000 00000000
          03000000 00bf9100

X=340282366920938463463374607431768211456
address=140029728
0858af20: 05000000 00000000
          00000000 00000000
0858af30: 00000000 01000000
          00000000 00000000
[533]
```

**問題 11.1 (10)** peek および poke を用いて, あたえられた文字列の中のアルファベット小文字をすべて大文字に変換するプログラムを書きなさい.

**問題 11.2 (20)** 整数がどのようにメモリに格納されているか解析して, 変数に格納された整数を 1 増やすプログラムを peek, poke を用いて書きなさい.

## 11.2　32 bit 整数の内部表現

C 言語などでは int と宣言されている変数には多くの場合 32 bit 整数が格納される. Asir では, 通常利用されている数は, bignum であるが, 関数 ntoint32(N) を用いて bignum N を符号なし 32 bit 整数に変換できる.

C 言語などでは int と宣言されている変数, つまり符号付き 32 bit 整数では, 最上位の bit を数の符号を表すのに用いている. 32 bit では説明を書くのにはあまりに大きな数を使わないといけないので, 符号付き 8 bit 整数の仕組みを説明しよう. 32 bit 整数でも仕組みは同じである.

符号付き 8 bit 整数では, 次のように整数と 8 桁の 2 進数 (2 桁の 16 進数) を対応させる.

| | |
|---|---|
| 00 | 0 |
| 01 | 1 |
| . | . |
| . | . |
| . | . |
| $7F$ | 127 |
| 80 | $-128$ |
| 81 | $-127$ |
| . | . |
| . | . |
| . | . |
| $FE$ | $-2$ |
| $FF$ | $-1$ |

この表をみればわかるように, 負の数では最上位の bit が 1 であり, また FF=11111111 が -1, FE=11111110 が -2 に対応しており, 符号付き 8 bit 整数での最小の負の整数は -128 である.

## 11.3 整数の内部表現

Asir では任意桁数の整数 (いわゆる bignum) を扱うことができるが, CPU が直接扱えるのは 32 bit (2 進数 32 桁) または 64 bit (2 進数 64 桁) というある決まった大きさ (1 ワードと呼ぶことにする) の整数で, それより大きな整数を扱うにはソフトウェアの手助けが必要となる. Asir においては, 大きな整数 (多倍長整数) は, ワードを単位とする配列として表現される. 1 ワードが 32 bit = 4 byte とするとき, 自然数 $a$ は次のように表現される. まず, $a$ の $2^{32}$ 進表示を考える.

$$a = a_0 2^0 + a_1 2^1 + a_2 2^2 + \cdots + a_{N-1} 2^{N-1} \quad (0 \leq a_i \leq 2^{32} - 1, a_{N-1} \neq 0)$$

これに対し, 長さ $N$ のワード配列 $w$, すなわち, 長さが $4N$ byte のメモリ上の連続領域を確保して,

$$w[i] = a_i \quad (0 \leq i \leq N - 1)$$

とする. これで自然数が表現できる. 実際には, Asir においては, 長さ $N + 1$ のワード配列を確保し, 先頭に長さ $N$ が書かれ, 残りに $2^{32}$ 進表示の各桁が書かれる.

1 ワードを構成する複数の byte がどういう順序でメモリに書かれているかは CPU に依存する. 下位 byte が下位アドレスに書かれる場合 *little endian*, 下位 byte が上位アドレスに書かれる場合 *big endian* と呼ばれる. Intel 80386 系 CPU は *little endian* であり, たとえば

$$987654321 = \texttt{0x3a} \cdot (2^8)^3 + \texttt{0xde} \cdot (2^8)^2 + \texttt{0x68} \cdot (2^8)^1 + \texttt{0xb1} \cdot (2^8)^0$$

は, アドレスの下位から上位に向かい `b1 68 de 3a` と並ぶ.

## 11.4 浮動小数点数の内部表現

小数点のある数は計算機のなかで 32 bit または 64 bit のデータとして表現されている. このような数を浮動小数点数とよんでいる. この表現法を Intel 80386 系の CPU を例にして 32 bit 表現の場合に説明しよう. なお, Asir で関数 `deval` が戻すのは 64 bit の浮動小数点数である.

32 bit データは, 1 bit の符号 $s$, 8 bit の指数部 $e$ , 23 bit の仮数部 $t$ に分割される. ここで

$$e = (e_7, e_6, \ldots, e_0),\ t = (t_{22}, t_{21}, \ldots, t_1, t_0)$$

である. $s$, $e_i$, $t_i$ は 0 か 1 の値をとる. 以下, $1.t$ は

$$1 + t_{22} 2^{-1} + t_{21} 2^{-2} + \cdots + t_1 2^{-22} + t_0 2^{-23}$$

を表すものと約束しよう. 80386 系の CPU の場合, $s$ が 0 のとき, 組 $(s, e, t)$ は, 数

$$1.t \times 2^{e-127}$$

を表す. $s$ が 1 のとき, 組 $(s, e, t)$ は, 数

$$-1.t \times 2^{e-127}$$

を表す. また 0 は 組 $(0, 0, 0)$ または $(1, 0, 0)$ で表現する.
例:
1 は

```
00        00       80       3F
00000000  00000000 10000000 00111111    e は 011 1111 1
```

$-1$ は

```
00        00       80       BF
00000000  00000000 10000000 10111111    e は 011 1111 1 = 127
                            ^ 符号 bit
```

8 は

```
00        00       00       41
00000000  00000000 00000000 01000001    e は 100 0001 0 = 130
                   -         ------- e を決める bit
```

$1.000000119209 \simeq 1.0/(2^{23}-1)$ は

```
01        00       80       3F
00000001  00000000 10000000 00111111    e は 011 1111 1 = 127
--------  --------  -------    仮数部 t を決める bit
76543210  fedcba98
```

$e$ が 0 か 255 = 0xff のときは, 特別な意味をもつ. くわしくは IEEE754 規格をみよ.

上級者向け参考: 浮動小数点数がどのようにメモリに格納できるかを調べるには, メモリを" 生" で扱える $C$ や マシン語などの言語を用いると容易である. Basic などの言語ではこのようなことを調べるのは容易ではない. 次の $C$ のプログラムは, 入力された小数点数がメモリにどのように格納されているのかを表示する.

```
#include <stdio.h>
main() {
  float a;
  unsigned char *p;
  int i;
  scanf("%f",&a);  p = (char *)&a;
  printf("%f\n",a);
  for (i=0; i<sizeof(float); i++) {
    printf("%02x ", *p);
    p++;
  }
  printf("\n");
}
```

**問題 11.3** 同じような機能をもつ Asir プログラムを作成せよ.

**問題 11.4** 1 で次のようなプログラムを実行すると, I<2 にもかかわらず, $I = 2$ の時が実行されてしまう例をみた.

```
[0]     for (I=0; I<2; I = I+0.2) {    RETURN
            print(I,0); print(" : ",0);    RETURN
            print(deval(I^(1/2)));    RETURN
        }    RETURN
```

この理由を 0.2 が計算機内部でどう表現されるのか? 0.2.+0.2 の結果はどうなるのか? を調べて説明せよ.

# 第12章　再帰呼び出しとスタック

## 12.1　再帰呼び出し

階乗 $n!$ は次のような性質をみたす.

$$n! = n \cdot (n-1)!, \quad 0! = 1.$$

ある関数のなかから, 自分自身を呼び出すことを再帰呼び出し (recursive call) という. 多くのプログラム言語では, 再帰的な関数呼び出しをみとめており, その機能を使うと上のような性質をもちいてそのままプログラムすることが可能となる.
階乗関数の再帰的実装:

```
def rfactorial(N) {
   if (N < 0) error("rfactorial: argument must be 0 or natural numbers.");
   if (N == 0) return(1);
   else {
      return(N*rfactorial(N-1));
   }
}
```

上の関数定義をみればわかるように関数 `rfactorial` のなかで関数 `rfactorial` を呼んでいる.

**例 12.1** 漸化式 $x_n = 2x_{n-1} + 1,\ x_0 = 0$ できまる数列の $n$ 項めをもとめるプログラムを作ろう. プログラムは以下のようになる. 再帰をつかわないプログラムと比べて, ずいぶんすっきりかけていることに注意しよう.

```
def xn2(N) {
   if (N == 0) return(0);
   XN = 2*xn2(N-1)+1;
   return(XN);
}
```

プログラム自体は単純であるが, 実はこのような場面で再帰をもちいるのはあまり得策ではない. メモリや実行効率の低下があるからである. 8 章で説明した関数呼び出しの仕組みを思いだそう. 関数およびその局所変数は動的に生成, 消滅を繰り返している. たとえば, 上のプログラムで `xn2(4)` をよぶと, `xn2(3)`, `xn2(2)`, `xn2(1)`, `xn2(0)` がつぎつぎとよびだされ, `xn2(0)` の実行中には, 5 つの `xn2` が実行されており, したがって局所変数 `XN` および 引数 `N` も, それぞれ 5 つ生成されている. したがって一般に, `xn2(n)` に対しては, 最大 $2(n+1)$ 個の変数領域が確保されることになる.

次のようなプログラムを書けば, このようなメモリの無駄使いは生じない.

```
def xn2(N) {
   XN = 0;
   for (I=0; I<N; I++) {
     XN = 2*XN+1;
   }
   return(XN);
}
```

処理系によっては, このように非効率的に書かれた再帰呼び出しを自動的に効率的な形式に変更する機能をもったものもある.

**問題 12.1** フィボナッチ数とは次の漸化式できまる数列である.

$$f_n = f_{n-1} + f_{n-2},\ f_1 = f_2 = 1.$$

フィボナッチ数を求める再帰的なプログラム

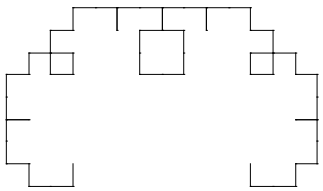
```
def fib(N) {
   if (N == 1) return(1);
   if (N == 2) return(1);
   return(fib(N-1)+fib(N-2));
}
```

を考える. このプログラムは効率の悪い再帰プログラムである. 理由を述べよ. じっさい良くないプログラムであることを, 再帰を用いないプログラムと比較して確かめよ.

再帰がもっとも強力にその威力を発揮するのは, データ構造自身が再帰的な構造をもっているリスト構造の場合や, 構文解析の場合である. Quick sort なども再帰がその威力を発揮する場合である. これらについては後の節でくわしく考察する. その他, フラクタル (自己相似図形) を描くのも再帰をもちいると簡単であることがおおい.

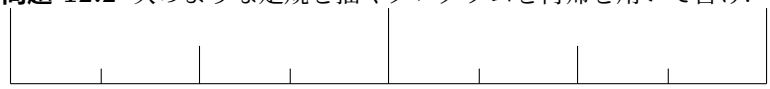
**例 12.2** C 曲線を書くプログラムを書け. C 曲線は与えられた線分の集合に含まれる各線分を

のように折れ線に置き換えることにより生成される図形である. この置き換えプロセスを 1 本の線分よりスタートして, $n$ 回繰り返すと $n$ 次の C 曲線を得ることができる. 次の図は 6 次の C 曲線である.



この図を生成したプログラムを図 12.1 に掲載する. プログラムのポイントは次の事実である:
$(a,b)$ および $(c,d)$ をある正方形の対角線の頂点とするとき, $((a+c+d-b)/2,(b+d+a-c)/2)$, $((a+c+b-d)/2,(b+d+c-a)/2)$ はこの正方形ののこりの頂点である. (内積と対角線の中点からの長さを考えると簡単にわかる.)

**問題 12.2** 次のような定規を描くプログラムを再帰を用いて書け.

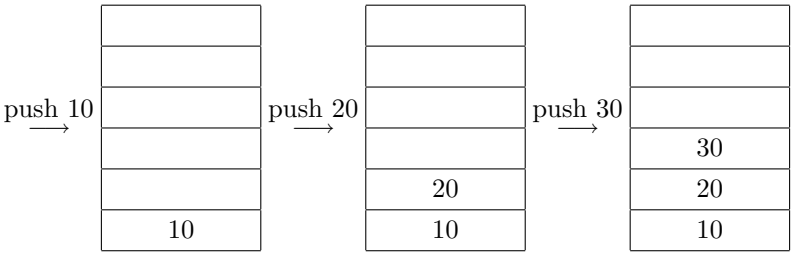


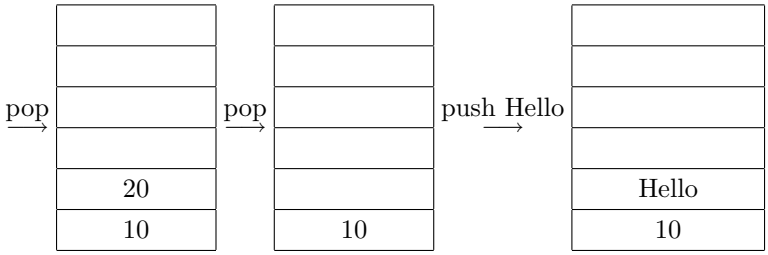
## 12.2 スタック

関数呼び出しとくに再帰的関数呼び出しはスタックをもちいて実現されている.

このことを理解するためにまずスタックとはなにかを説明し, それから再帰がどのようにスタックを用いて実現されているか説明しよう.

スタックは, データを push 操作で格納し, pop 操作でデータを取り出すオブジェクトである (またはデータ構造と理解してもよい). push, pop は先いれ, 先だし機能 (FIFO, First In, First Out) をもつ. たとえば, データ 1, 2, 3 を順番に push すると, pop したときは, 3, 2, 1 の順番でデータを取り出すことが可能である. スタックは次のプログラムのように配列 (ベクトル) を用いると容易に実現可能である.






```
/* $ stack.rr,v 1.2 2001/01/28
   02:22:03 taka Exp $ */
Stack_size = 100$
Stack_pointer = 0$
Stack = newvect(Stack_size)$

def init() {
  extern Stack_pointer;
  Stack_pointer=0;
}

def push(P) {
  extern Stack_size,Stack_pointer,
         Stack;
  if (Stack_pointer>=Stack_size){
    error(" stack overflow. ");
  }
  Stack[Stack_pointer] = P;
  Stack_pointer++;
  return(Stack_pointer);
}

def pop() {
  extern Stack_size,Stack_pointer,
         Stack;
  if (Stack_pointer <= 0) {
    print("Warning: stack
      underflow. Return 0.");
    return(0);
  }
  Stack_pointer--;
  return(Stack[Stack_pointer]);
}
```

実行例:

```
[366] push(10);
1
[367] push(20);
2
[368] push(30);
3
[369] pop();
30
[370] pop();
20
[371] push("Hello");
2
[372] pop();
Hello
[373] pop();
10
[374] pop();
Warning: stack underflow.  Return 0.
```

push, pop を用いると, 次のようにスタック電卓を簡単に作ることが可能となる. スタック電卓は, 式

を後置形式で入力すると計算する電卓である. 後置形式は, 演算子を最後に書く形式であり, 括弧を必要としない. たとえば, 後置形式の

```
2  3  +  5  *  =
```

は, (2+3)*5 を意味する. スタック電卓では 2 と 3 を スタックに push, + がきたら, スタックより 2 個データを pop し, 足した結果を スタックに push, * がきたら, スタックより 2 個データを pop し, かけた結果を スタックに push, = がきたら, データをスタックより pop して, 印刷する.
スタック電卓のプログラムは図 12.2 に掲載する.

関数 casio() は, スタック電卓である. 数字は 1 桁した利用できない. セミコロン ; を行の始めへ入力すると終了する.
例

```
[365] casio();
2 3 + =           入力
Answer=5          答え
;
0
[366] casio();
2 3 + 9 * =       入力
Answer=45         答え
;
0
[367]
```

さてスタックを用いて再帰を実現するには, 関数の実行前に局所変数をスタック上に確保し, 再帰呼び出しの実行が終った時点で, 局所変数をスタックから消去 ( pop ) すればよい. また関数を呼び出す前に戻り番地もスタックに格納しておく必要がある. これが, 関数が生成, 消滅している内部的仕組みである.

```
load("glib")$
def cCurve(P) {
  if (length(P) < 2) return(0);
  A = P[0][0];
  B = P[0][1];
  C = P[1][0];
  D = P[1][1];
  Tmp = [[A,B],[(A+D+C-B)/2,(B+A+D-C)/2],
         [(A+D+C-B)/2,(B+A+D-C)/2],[C,D]];
  return(append2(Tmp,cCurve(cdr(cdr(P)))));
}
def append2(A,B) {
  if (type(B) == 0) return A;
  else return append(A,B);
}
def main(N) {
   Tmp = [[0,0],[1,0]];
   for (I=0; I<N; I++) {
      Tmp = cCurve(Tmp);
      print(Tmp);
   }
   glib_window(-1,-1,2,2);
   for (I=0; I<length(Tmp)-1; I++) {
     glib_line(Tmp[I][0],Tmp[I][1],Tmp[I+1][0],Tmp[I+1][1]);
   }
}
print("Type in, for example,  main(8);")$
end$
```

図 12.1: C 曲線を書くプログラム

```
#define SPACE 0x20
#define ZERO  0x30
#define NINE  0x39
#define PLUS  43    /* + */
#define TIMES 42    /* * */
#define EQUAL 61    /* = */
#define SEMICOLON  59 /* ; */
def casio() {
  init();
  while(1) {
     purge_stdin();
     In = get_line();
     In=strtoascii(In);
     N = length(In);
     if (N == 0) break;
     if (In[0] == SEMICOLON) break;
     for (I=0; I<N; I++) {
       C = In[I];
       if (C <= SPACE) {
         /* skip */
       }else if ((C  >= ZERO) && (C <= NINE)) {
         push(C-ZERO);
       }else if (C == EQUAL) {
         print("Answer=",0); print(pop());
       }else if (C == PLUS) {
         A = pop(); B=pop();
         push(A+B);
       }else if (C == TIMES) {
         A = pop(); B=pop();
         push(A*B);
       }else {
         print("Invalid character ",0);
         print(asciitostr([C]),0);
         print(" in the input: ",0);
         print(asciitostr(In));
       }
}}}  end$
```

図 12.2: スタック電卓 casio.rr

# 第13章　リストの処理

配列は応用が広く, 頻繁に用いられる便利なデータ構造だが, 長さ固定という制限がある. リストもいくつかのデータをまとめたもので次のような特徴がある.

- 先頭に要素を追加できる.
- 先頭の要素を外せる.
- 要素の書き換えはできない.
- 空リストがある.

一見して不自由そうに見えるが, 実はリストは強力で, リストだけでなんでもプログラミングできる. 例えば emacs は LISP と呼ばれるリスト処理言語で記述されていて emacs での 1 文字入力も実はある LISP コマンドに対応している.

上の例で出てきた関数, 表現の説明は以下の通り.

1. リストの作り方 (その 1)

```
[0] A = [1,2,3];                    見ての通り
[1,2,3]                             表示が配列と微妙に違う
```

2. リストの作り方 (その 2)

```
[1] B = cons(0,A);                  先頭に要素を追加
[0,1,2,3]
[2] A;                              A は影響を受けない
[1,2,3]
```

3. リストの作り方 (その 3)

```
[3] C = cdr(A);                     cdr = クッダー; 先頭要素を取り外す
[2,3]
[4] A;                              A は影響を受けない
[1,2,3]
```

4. 空リスト

```
[5] A = [];                         [] は空のリストを表す
[]
[6] cons(1,A);
[1]
```

5. 要素取り出し (その 1)

```
[7] car(B);                          car = カー; 先頭要素を取り出す
0
```

6. 要素取り出し (その 2)

```
[8] B[2];                            配列と同様に書ける
2
```

7. 書き換え不可

```
[9] B[2] = 5;                        書き換えはダメ
putarray : invalid assignment
return to toplevel
```

**例 13.1** 例えば, A を B で割った商と剰余を返す関数は次のように書ける.

プログラム

```
def quo_rem(A,B) {
    Q = idiv(A,B);
    R = A - Q*B;
    return [Q,R];
}
```

実行例

```
[1] QR = quo_rem(123,45);
[2,33]
[2] Q = QR[0];
2
[3] R = QR[1];
33
```

**例 13.2** 集合を配列で表そうとすると, 要素の追加のたびに配列を作り直す必要が出てくる.

プログラム

```
def memberof(Element,Set)
{
  Size = size(Set)[0];
  for ( I = 0; I < Size; I++ )
    if ( Set[I] == Element )
      return 1;
  return 0;
}
```

Element が Set の要素なら 1, そうでなければ 0 を返す

プログラム

配列で表された集合の和集合を返す. 配列内には重複はないとする

```
def union(A,B)
{
  SA = size(A)[0];
  SB = size(B)[0];
  NotinB = 0;
  /* #(A-B) */
  for ( I = 0; I < SA; I++ )
    if ( !memberof(A[I],B) )
      NotinB++;
  /* #(A cup B) = #B+#(A-B) */
  SC = SB + NotinB;
  C = newvect(SC);
  for ( K = 0; K < SB; K++ )
    C[K] = B[K];
  for ( I = 0; I < SA; I++ )
    if ( !memberof(A[I],B) ) {
      C[K] = A[I];
      K++;
    }
  return C;
}
```

プログラム

配列で表された集合の共通集合を返す. 配列内には重複はないとする

```
def intersection(A,B)
{
  SA = size(A)[0];
  SB = size(B)[0];
  AandB = 0;
  /* #(A cap B) */
  for ( I = 0; I < SA; I++ )
    if ( memberof(A[I],B) )
      AandB++;
  C = newvect(AandB);
  for ( I = 0, K = 0; I < SA; I++ )
    if ( memberof(A[I],B) ) {
      C[K] = A[I];
      K++;
    }
  return C;
}
```

このような場合には, データ構造としてリストを用いるのが便利である.

プログラム

Element が Set の要素なら 1, そうでなければ 0 を返す (リスト版)

```
def memberof(Element,Set)
{
  for ( T = Set; T != [];
        T = cdr(T) )
    if ( car(T) == Element )
      return 1;
  return 0;
}
```

プログラム

和集合を返す. 配列内には重複はないとする (リスト版)

```
def union(A,B)
{
  C = B;
  for ( T = A; T != [];
        T = cdr(T) )
    if ( !memberof(car(T),B) )
      C = cons(car(T),C);
  return C;
}
```

**問題 13.1** 配列 A の要素の平均, 分散, 標準偏差をリストで返す関数を書け. 結果は浮動小数で返すこと. 有理数の浮動小数による近似値を返す関数は deval(M), 平方根は M^(1/2).

```
[0] A = 12345/6789;
4115/2263
[1] deval(A);
1.81838
[2] B = A^(1/2);
(4115/2263)^(1/2)
[3] deval(B);
1.34847
```

**問題 13.2** リストを逆順にしたリストを返す関数を書け. もちろん, 組み込み関数 reverse() を呼ぶのは反則.

(先頭を取り外す, という操作と, その要素を他のリストの先頭に付け加える, という操作を繰り返せばできる. 「他のリスト」の初期値として何を設定すればよいか.)

## 13.1 リストに対する基本計算

リストについては第 8 章の最初の節で簡単にふれた. リストというのは, 見かけは, 要素としてなにをいれてもいい配列 (ベクトル) のことである. たとえば, [3,2,"cat"] は, 数字 3 を 1 番目の要素, 数字 2 を 2 番目の要素, 文字列 "cat" を 3 番目の要素として持つリストである. リストの要素はまたリストでもよいので,

[3,[3,2,"dog"],"cat"]

は, 数字 3 を 1 番目の要素, リスト [3,2,"dog"] を 2 番目の要素, 文字列 "cat" を 3 番目の要素として持つリストである. さらには要素のない空リスト [ ] もある.

リスト L の先頭要素を取り出すには L[0] を用いてもよいが, 関数 car(L) を用いることも可能である. また, リスト L から先頭要素を除いたリストは, 関数 cdr(L) で求めることが可能である. たとえば,

```
[429] L=[3,[3,2,"dog"],"cat"];
[3,[3,2,dog],cat]
[430] car(L);
3
[431] cdr(L);
[[3,2,dog],cat]
```

となる. car(カァと読んでいい) と cdr (クッダーと読んでいい) は LISP 由来の関数名である.

リスト L の長さを戻す関数は length(L) である. 関数 append(L,M) は二つのリスト L と M をつないだリストを戻す. たとえば,

```
[432] L=[3,[3,2,"dog"],"cat"];
[3,[3,2,dog],cat]
[433] append(L,M);
[3,[3,2,dog],cat,3,[2,5],6]
```

となる.

リストの使い道は多岐にわたるが, 結果の大きさがあらかじめ予想できないときに結果をかたつけておく “袋” として利用するのは, 一番初歩的な利用法の一つである.

例を一つあげよう. たとえば次のプログラムは, 第 7 章の試し割りによる素因数分解法のプログラムで, N の素因子をリストにして戻す. リスト L がすでに求まった素因子を格納しておく “袋” になっている. 新しい素因子を得たら, 関数 append を用いて, L にその因子を加える.

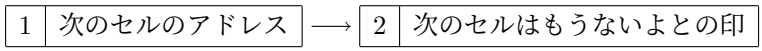
プログラム

```
def prime_factorization(N) {
  K = 2;
  L = [ ];
  while (N>=2) {
    if (N % K == 0) {
      N = idiv(N,K);
      L = append(L,[K]);
    }else{
      K = K+1;
    }
  }
  return(L);
}
```

実行例

```
[430] prime_factorization(98);
[2,7,7]
```

リストをベクトルに, ベクトルをリストに変換する関数はそれぞれ, newvect, vtol である. リストとベクトルの違いは何であろうか? リスト L に対しては L[1] = 3 といった代入ができないのが表面的な違いであるくらいでほとんど同じものに見える. しかし, リストとベクトルではその内部でのデータ表現法が全く異っている. ベクトルはメモリのなかでひとつづきの領域が確保されそのなかにデータはインデックス順に格納されると理解しておいてよい. リストはもっと複雑な構造である. データ領域とアドレス領域なる二つの領域を持っているデータ構造をセルと呼ぶことにする. リストは, セルの集まりである. たとえば [1,2] というリストは, 次のような二つのセルの集まりである.




このような構造の帰結として, リスト L に対して, L[1000] を取り出すのと, ベクトル V に対して, V[1000] を取り出すのを比べると, ベクトルの方が早いことになる. しかし, リストはサイズのどんどん増えて行くデータを格納するには, ベクトルより有利である.

リストの内部構造をきちんと理解するには C 言語の構造体と構造体へのポインタまたは, 機械語の間接アドレッシングの仕組みを理解する必要がある. セルを C 言語の構造体で書くと次のようになる.

```
struct cell {
  void *data;
  struct cell *next_address;
};
```

**問題 13.3** car, cdr はメモリ上でどのように動作しているのか考えてみよ.

上のプログラムでは L=append(L,[K]); を用いて, リストにどんどん要素を付け足していった. 実はこの方法は巨大なリストを扱うときにはよくない. メモリの無駄使いが生じる.

```
L = cons(K,L);
```

と書くとメモリの無駄使いが生じない. cons(K,L) は, car が K, cdr が L となるようなリストを生成するが, L にあらわれるセルの複製は作成しない. 内部的には, K を格納するセルを作成して,

そのセルの次の元として, L を指すようにする. そして, K の先頭アドレスをもどしている. 一方 append(L,[K]) の場合には, 毎回リスト L に現れるセル全ての複製が作成されて, その最後に K を格納するセルがつながれることになる.

## 13.2 リストと再帰呼び出し

リストの要素はまたリストでよいという再帰的構造が存在しているので, リスト処理の関数は, 再帰を用いると気持よく書けることがおおい.

**例 13.3** リストの中に、数値データが何個あるかを数える関数 count_numbers を作れ. たとえば count_numbers([ 1, cat, 3]) は 2 を戻す. ただし, リストの中に入る要素は, 数か多項式か文字列かリストに限るものとする. (ヒント: type を使う. )

データ型をみる関数 type(L) は L がリストの時 4, 0 以外の数字のとき 1, 0 のとき 0 を戻す. よって次のプログラムでよい.
プログラム

```
def count_numbers(L) {
  if (length(L) == 0) return(0);
  C = car(L);
  if (type(C) == 0 || type(C) == 1) {
     return(1 + count_numbers(cdr(L)));
  }else if (type(C) == 4) {
     return(count_numbers(C)+count_numbers(cdr(L)));
  }else{
     return(count_numbers(cdr(L)));
  }
}
```

**問題 13.4** リストの中に、リストが何個あるかを数える関数 count_lists を作れ. たとえば count_lists([ [0,1], "cat", [[ 7, 8], 3] ]) は 2 を戻す.

**問題 13.5** 第 8 章の問題の clone_vector を再帰的に書くことにより, 任意のベクトルの複製を作れるように書き換えよ.

**問題 13.6** リスト L のなかに与えらた要素 A が存在してるかどうか判定する関数 member(A,L) をかけ.

**問題 13.7** Well-formed formula とはたとえば、 [or, p,[and,[[not,p], q]]] なる形の式だとする. ここで、 p, q は真 (1) または偽 (0) だとする. 常に真である式を恒真式という. 与えられた Well-formed formula が恒真式かどうか判定する関数を作れ.

**問題 13.8** リストの要素を一列に並べる関数を書け.

$$[1, 2, [3, [4, 5]], 6, [7, 8]] \rightarrow [1, 2, 3, 4, 5, 6, 7, 8]$$

という操作を行うという意味である. (list_append を使ってよい.)

**問題 13.9** 与えられたリストの要素を並べ変えて得られる全てのリストからなるリストを返す関数を書け. たとえば, `[1,2]` を入力とした場合は, `[[1,2],[2,1]]`, それから `[1,2,3]` を入力したときは `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]` をもどす関数.
ヒント:

1. 与えられたリストを $L$ とし, その長さを $N$ とする.
2. $L$ は $L[0]$ から $L[N-1]$ までの要素をもつ.
3. 結果を保持するリスト $R$ を用意する. 最初は [] にしておく.
4. $I = 0$ から $N-1$ に対して次の操作を行う.
   (a) $L$ から $L[I]$ を抜いたリストを $LI$ とする.
   (例 $L = [1,2,3,4]$ から $L[1]$ を抜くと $LI = [1,3,4]$)
   (b) $LI$ は長さ $N-1$ のリストで, これを引数として自分を呼び出し, $LI$ から生成される順列全てのリスト $RI$ を作る. $RI$ の要素はまたリスト.
   (上の場合, $RI = [[1,3,4],[1,4,3],[3,1,4],[3,4,1],[4,1,3],[4,3,1]]$.)
   (c) $RI$ の各要素の先頭に $L[I]$ を付け加える.
   (上の場合, $[[2,1,3,4],[2,1,4,3],[2,3,1,4],[2,3,4,1],[2,4,1,3],[2,4,3,1]]$ となる.)
   (d) これを, 結果を保持するリスト $R$ に追加する.

ここでの方法ではリストの $I$ 番目を抜く関数が必要となる. これは, 例えば重なった $K$ 枚の紙があったとして, その上から $I$ 番目の紙を抜き出す場合を考えればなにをすればよいか分かると思う. くどいのを承知で説明すると

1. 一番上から 1 枚ずつとって, 順に隣に重ねる操作を $I$ 回行う
2. 一番上をはずす. (隣には重ねない.)
3. 隣の紙を一枚ずつ順に戻す.

とすればよい.

いずれにしても, 再帰で書くことになるが, 注意すべき点は, 再帰の終点をどこにするかである. $L$ の長さが 1 の場合に終点とすれば分かりやすい. この場合 $L = [a]$ なら $[[a]]$ を返すようにすればよい. (結果は順列 (リスト) のリストとなることに注意.) $L = [\,]$ を終点とすることもできるが, この場合 [[ ]] (空リストを要素とするリスト) を返す必要がある. こうしないと, 再帰が進まない.

# 第14章　整列：ソート

Risa/Asir には組み込み関数として qsort がある．qsort の help メッセージをみると, quick sort 法によりソート (データの並べかえ) をやると書いてある．quick sort 法とはどのような方法であろうか?

ソートをするにはいろいろな方法があり、その計算量も詳しく解析されている．またソートのいろいろなアルゴリズムは他の分野のアルゴリズムの設計のよき指針となっているし, ソートを利用するアルゴリズムも多い．たとえば, 多項式の足し算はマージソートにほかならない．この章はソートの仕組みへの簡略な入門である．

## 14.1　バブルソートと クイックソート

14.3 節のプログラムがバブルソートとクイックソートをするプログラムである．このプログラムを解説しよう．

1. データのサイズはそれぞれ testBuble および testQuick の N で指定する．
2. データは配列 (ベクトル) A に乱数をいれて初期化する．
3. quickSort(A,P,Q) は A の $P$ 番めから $Q$ 番めまでをクイックソートする．
4. tstart() で時間計測開始, tstop() で時間計測終了および時間表示である．
5. バブルソートの計算量は $O(n^2)$, クイックソートの平均計算量は, $O(n \log n)$ である．

バブルソートでは, 配列 anArray の隣同士の元を比較して, 大きいものから順にどんどん下図の右に集める．

| anArray[0] | anArray[1] | anArray[2] | ··· | anArray[Size-1] |
|---|---|---|---|---|

関数 bubleSort の変数 J と I による 2 重ループでこれを実現している．

クイックソートではまず, M より小さいデータを, M の左に, M より大きいデータを, M の右にあつめる．これを実行しているのが, 関数 quickSort の while ループである．そのあと, M に左および右にまたクイックソートを再帰的に適用することによりソートを完成させる．

**例題 14.1** [10]　大きさ 7 のデータと大きさ 70, および 700 のデータをバブルソート, クイックソートしてその実行時間を調べなさい．アルゴリズムの違いで計算の速度がかわることを実感してもらいたい．

**入力例 14.1**　まずはデータの数を 7 として, やってみよう．

```
[346] load("sort.rr");
1
[347] load("sort2.rr");
1
```

```
[348] testBuble(7);
0.000644sec(0.00064sec)
0
[349] testQuick(7);
0.000723sec(0.00073sec)
0
```

というぐあいにバブルのほうが早い. このことから, 漸近的な計算量のうえでは, クイックソートの方が早いが, データが少ない時は単純なアルゴリズムのほうがプログラムが単純になってはやいことがわかる. では, つぎにデータの数を 70 としてやってみよう.

```
[357] testBuble(70);
0.0406sec + gc : 0.04641sec(0.09074sec)
0
[358] testQuick(70);
0.008668sec(0.008675sec)
0
```

ということで, クイックソートの方が早くなる.

データ数が 700 になると, クイックソートの方が断然はやい. ($70^2 = 4900$, $70 \log 70 \simeq 297$ だが, $700^2 = 490000$, $700 \log 700 \simeq 4586$ である.)

```
[364] testBuble(700);
4.088sec + gc : 1.476sec(5.571sec)
0
[365] testQuick(700);
0.1606sec + gc : 0.04788sec(0.2147sec)
0
```

**問題 14.1** [10]　`N` の値をいろいろ変えて計算時間を測定し, グラフ用紙にグラフとして書いてみよう.

## 14.2　計算量の解析

各種ソート法の計算量については, たとえばセジビックのアルゴリズムの本が詳しい [1]. 結論だけのべておくと, $n$ 個のデータをバブルソートするための計算量は $O(n^2)$, クイックソートするための平均計算量は $O(n \log n)$ である.

## 14.3　プログラムリスト

バブルソートのプログラム `sort.rr` は次の二つの関数 `bubleSort` と `testBuble` からなる.

```
def bubleSort(AnArray) {
  Size = size(AnArray)[0];
  for (J=Size-1; J>0; J--) {
    for (I=0; I<J; I++) {
      if (AnArray[I] > AnArray[I+1]) {
        Tmp = AnArray[I+1];
        AnArray[I+1] = AnArray[I];
        AnArray[I] = Tmp;
      }
    }
  }
}
```

```
def testBuble(N) {
  A = newvect(N);
  for (I=0; I<N; I++) {
    A[I] = random() % 100;
  }
  /* print(A); */
  tstart();
  bubleSort(A);
  tstop();
  /* print(A); */
}
end$
```

クイックソートのプログラム sort2.rr は次の二つの関数 quickSort と testQuick からなる.

```
def quickSort(A,P,Q) {
   if (Q-P < 1) return;
   Mp = idiv(P+Q,2);
   M = A[Mp];
   B = P; E = Q;
   while (1) {
     while (A[B] < M) B++;
     while (A[E] > M && B <= E) E--;
     if (B >= E) break;
     else {
       Tmp = A[B];
       A[B] = A[E];
       A[E] = Tmp;
       E--;
     }
  }
  if (E < P) E = P;
  quickSort(A,P,E);
  quickSort(A,E+1,Q);
}
```

```
def testQuick(N) {
  A = newvect(N);
  for (I=0; I<N; I++) {
    A[I] = random() % 100;
  }
  /* print(A);*/
  tstart();
  quickSort(A,0,N-1);
  tstop();
  /* print(A); */
}
end$
```

## 14.4　ヒープソート

クイックソートの平均計算量は $O(N \log_2 N)$ だが, 最悪の場合 $O(N^2)$ となる. ここでは最悪でも $O(N \log_2 N)$ でソートできるアルゴリズムを一つ紹介する.

### 14.4.1 ヒープ

次の性質を満たす図を考える (図 14.1). これを 2 分木とよぶ.

1. 各レベル ($i = 0, 1, \ldots$) には, 最終レベルを除いて $2^i$ 個の元 (ノード) が並んでいる.
2. 最終レベルは左からすき間なしに並んでいる.
3. レベル $i$ の左から $k$ 番目 ($k = 1, 2, \ldots$) のノードは, レベル $i+1$ の左から $2k-1$, $2k$ 番目のノードと線で結ばれている. (存在すればの話) 線で結ばれているノードの組において, 上 (レベル番号が小さい) を**親**, 下を**子**と呼ぶ. レベル 0 のノードを**根**, 子がないノードを**葉**と呼ぶ.
4. 各ノードには数字が書かれていて, 親は子より小さくない.

このような性質を満たす図を**ヒープ**と呼ぶ.

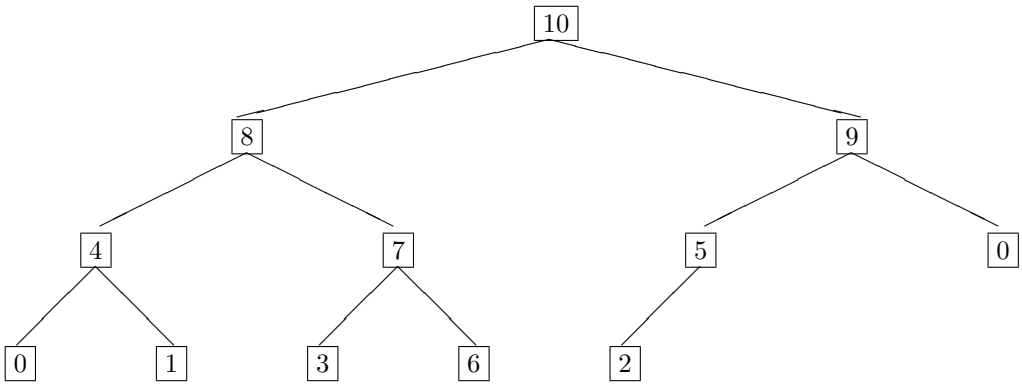



図 14.1: ヒープの例

与えられた集合からヒープを構成できれば, 元の集合を整列するのは容易である.

- 方法 1
  1. レベル 0 のノードが最大なので, これを取り外す.
  2. 2 番目に大きいのはレベル 1 の 2 つのうちどちらか. 大きい方をレベル 0 に昇格.
  3. レベル 1 の空いた場所に, レベル 2 から昇格, ...
  4. これらを繰り返す.
- 方法 2
  1. レベル 0 のノードが最大なので, これを取り外す.
  2. 最終レベルの右端のノードをとりあえずレベル 0 に置く. 2 分木を構成するノードの個数は 1 減っている.
  3. ヒープ条件が満たされるまで, レベル 0 に置いた元を順に落して行く.

方法 2 の利点は

- 「ある場所から落す」というサブルーチンが, ヒープを構成するのにそのまま使える.
- 最終レベルの右端が空くので, そこに取り外したレベル 0 の元を置ける. すると, 最終的にヒープ自体を上位レベルから並べて見ると, 整列されていることになる.

「落す」とは, (親, 子 1, 子 2) という組がヒープ条件を満たすように入れ換えることである. 入れ換えの必要がなくなった時点でストップして次のステップに進めばよい.

### 14.4.2　ヒープの配列による表現

レベル 0 から順に配列に詰めていくことで, ヒープを配列で表現できる. インデックスの対応を分かりやすくするために, 0 番目でなく 1 番目から詰めることにする. 配列を $A$ とすれば,

| | | |
|---|---|---|
| レベル 0 | $A[1]$ | $2^0$ 個 |
| レベル 1 | $A[2], A[3]$ | $2^1$ 個 |
| レベル 2 | $A[4], A[5], A[6], A[7]$ | $2^2$ 個 |
| | ... | |
| レベル $k$ | $A[2^k], A[2^k+1], \ldots, A[2^{k+1}-1]$ | $2^k$ 個 |

と対応する. この表現のもとで, 次が成り立つ. $N$ を要素の個数とする. $\lfloor x \rfloor$ を $x$ を越えない最大の整数とする.

- レベル $k$ までの要素の個数は $1+2+2^2+\ldots+2^k = 2^{k+1}-1$ 個.
- レベル $k$ の $l$ 番目のノードは $A[2^k-1+l]$.
- 子があるノードは $A[1], \ldots, A[\lfloor N/2 \rfloor]$.
- $A[I]$ の子は $A[2I]$, $A[2I+1]$ (もしあれば).
- $A[I]$ の親は $A[\lfloor I/2 \rfloor]$

**問題 14.2** 上の性質を証明せよ.

### 14.4.3　`downheap()`

前節の配列表現を使って, 2 分木の任意の位置から要素を落す関数 `downheap()` を書いてみる.

```
def downheap(A,K,N) {
    /* place A[K] at the correct position in A */
    while ( 2*K <= N ) {
        J = 2*K;  /* A[J] is the first child */
        if ( J == N ) {
            /* A[J] is the unique child */
            if ( A[K] < A[J] ) swap(A,K,J);
            /* A[J] is a leaf */
            break;
        } else {
            /* A[K] has two children A[J] and A[J+1] */
            /* M = max(A[K],A[J],A[J+1]) */
            M = A[K] >= A[J] ? A[K] : A[J];
            if ( A[J+1] > M ) M = A[J+1];

            if ( M == A[K] ) break; /* we have nothing to do */
            else if ( M == A[J] ) {
                swap(A,K,J);
                K = J; /* A[K] is moved to A[J]; */
            } else {
                swap(A,K,J+1);
                K = J+1; /* A[K] is moved to A[J+1]; */
            }
        }
    }
}
```

```
def swap(A,I,J) {
    T = A[I]; A[I] = A[J]; A[J] = T;
}
```

これは, 次のように簡潔に書ける.

```
def downheap(A,K,N) {
    V = A[K];
    while ( 2*K <= N ) {
        J = 2*K;
        if ( J < N && A[J] < A[J+1] ) J++;
        if ( V >= A[J] ) break;
        else {
            A[K] = A[J];
            K = J;
        }
    }
    A[K] = V;
}
```

**問題 14.3** このプログラムの動作を説明せよ.

**問題 14.4** 上から落して正しい位置に置くのが downheap() だが, 葉としてつけ加えて, 親より大きかったら親と交換する, という方法で昇らせることでもヒープが再構成できる. この関数 upheap(A,K) を書け.

### 14.4.4　ヒープソート

前節の downheap() を用いて次のようなプログラムを書くことができる.

```
def heapsort(L) {
    N = length(L);
    A = newvect(N+1);
    for ( I = 1; I <= N; I++, L = cdr(L) ) A[I] = car(L);
    /* heap construction; A[[N/2]+1],...,A[N] are leaves */
    for ( K = idiv(N,2); K >= 1; K-- ) downheap(A,K,N);
    /* retirement and promotion */
    for ( K = N; K >= 2; K-- ) {
        swap(A,1,K);
        downheap(A,1,K-1);
    }
    for ( I = 1, R = []; I <= N; I++ ) R = cons(A[I],R);
    return R;
}
```

このプログラムは, 与えられたリスト $L$ をソートしたリストを返す. ヒープの構成は, 子を持つ最後の要素である $A[\lfloor N \rfloor]$ から順に, その要素の子孫からなる 2 分木に対して downheap() を呼び出すことで行われる. 現在選ばれている要素に対し, 子を根とする木がヒープをなすことは数学的帰納法による.

$A[\lfloor N \rfloor + 1]$ 以降は葉なので, それらを根とする木に対しては自動的にヒープ条件がなりたっていることから帰納法の最初のステップが正当であることがわかる.

出来上がったヒープに対して, 根と, その時点における最後尾の要素を入れ換えて, `downheap()` を呼び出すことで, ヒープ条件を保ちながら要素の個数を一つずつ減らすことができる. さらに, 根はそのヒープの最大要素で, それが順に空いた場所に移されるので, 配列としては, 後ろから大きい順に整列することになる.

**問題 14.5** $L = [11, 9, 5, 15, 7, 12, 4, 1, 13, 3, 14, 10, 2, 6, 8]$ の ヒープソートによるソーティング. `http://www.math.kobe-u.ac.jp/~noro/hsdemo.pdf` にヒープ構成およびソーティング (retirement and promotion) の経過が示されている.

**定理 14.1** ヒープソートの計算量は $O(N \log_2 N)$ である.

**問題 14.6** 定理を証明せよ. (ヒント : $N = 2^n - 1$ で考えてよい. 高さ, すなわち頂点から最下段までのレベルの差が $k$ の `downheap()` 一回にどれだけ比較が必要か考える. あとは, ヒープ構成, 整列それぞれに, どの高さの `downheap()` が何回必要か数えればよい. )

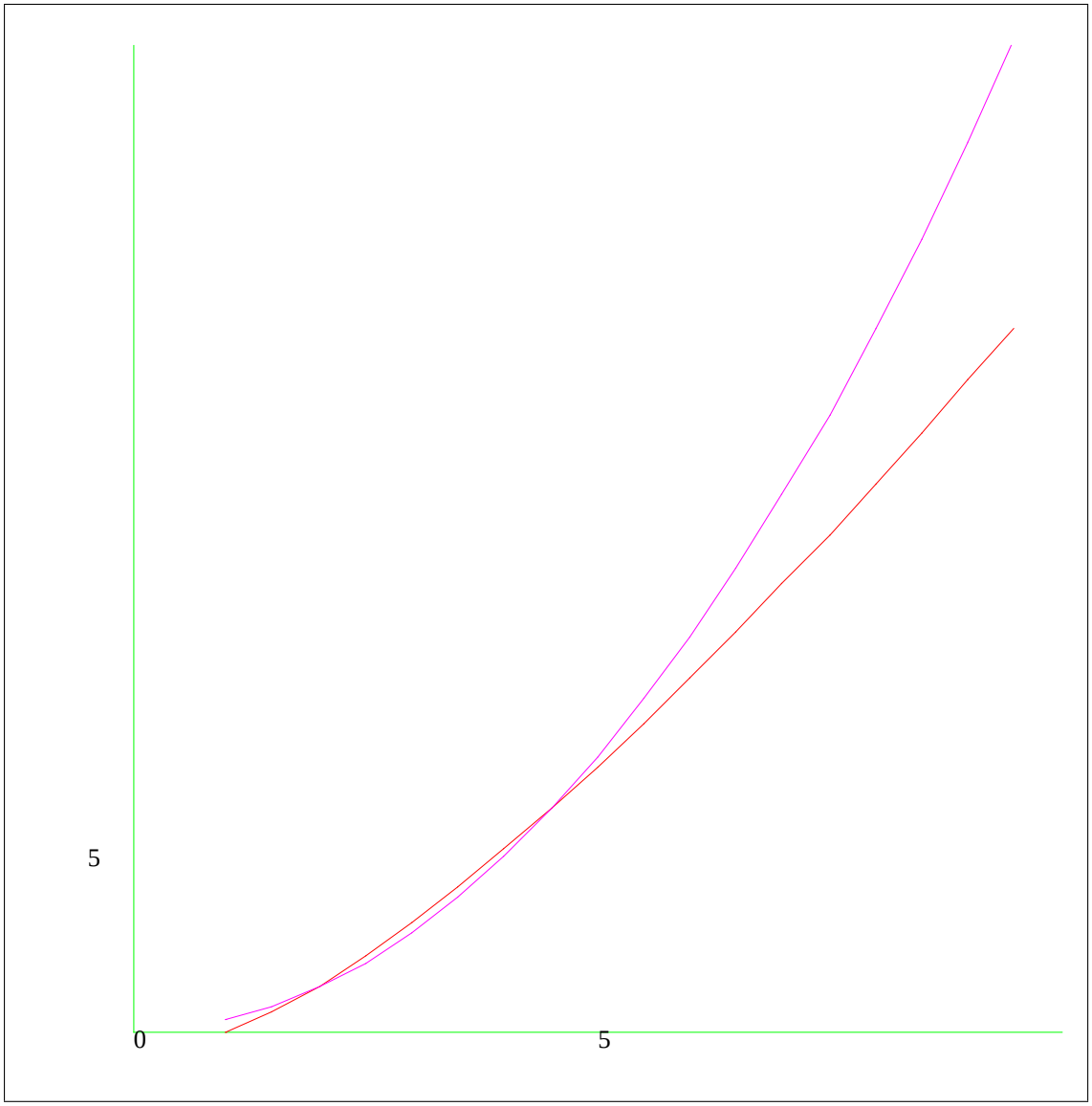
注意: クイックソートは平均 $O(N \log_2 N)$, 最悪 $O(N^2)$ のアルゴリズムで, ヒープソートは最悪でも $O(N \log_2 N)$ だが通常はクイックソートが使われる場合が多い. これは, クイックソートに比べてヒープソートが複雑であるため, ほとんどの入力に対してはクイックソートの方が実際には高速なためである. しかし, 前節, 本節で与えたプログラム例がそれぞれ最良とは限らないので, 双方比較してどちらが高速か分からない. 興味がある人は, 同じ例で比較してみたり, あるいはより効率の高い実装を行ってみるとよい.

## 14.5 章末の問題

1. Selection sort, Insertion sort, Merge sort, Shell sort はどのようなソート法か調べ, これらおよび buble sort, quick sort について計算時間を 500 個から 4000 個程度のデータについて比較せよ. この計測データをもとに shell sort の計算量 $O(f(n))$ の $f(n)$ を推定しなさい.

2. ソートのループの中に, `print` 文を挿入し, 20 個程度のデータについてソートがどのように進んでいるか, 実際のデータについて解説しなさい.

3. 第 6 章の問題 1 の解を高速に求めるプログラムを作成しなさい. (単なる quick sort では不十分. 全体をソートする必要がないことに注意. なお, quick sort の応用だと, 最悪計算量が $O(n^2)$ ($n$ は配列のサイズ) なってしまうが, $O(n)$ アルゴリズムが存在する. 詳しくは [1] 参照.)




Risa/Asir ドリル ギャラリー：$n \log n$ のグラフと $n^2/3$ のグラフ.

# 関連図書

[1] R. Segiwick, アルゴリズム 1,2,3. 近代科学社.

アルゴリズム全般に関する教科書. Pascal, C, C++ 版あり. 1 巻はリスト構造, 木構造およびいろいろなソート法とその計算量解析にくわしい.

# 第15章　1 変数多項式の GCD とその応用

## 15.1　ユークリッドのアルゴリズム

数学科の学生は代数学の講義で, “ユークリッド整域” なる概念を習ったことと思う. 整数環 $\mathbf{Z}$, 一変数多項式環 $\mathbf{k}[x]$ はともにユークリッド整域であり, 次の割算定理が成り立つ: $R$ を 整数環または一変数多項式環とする. このとき $R$ の 0 でない任意の元 $f$, $g$ に対して,

$$f = qg + r, \quad \deg(r) < \deg(g)$$

を満たす $R$ の元 $q$, $r$ が存在する. ここで $R = \mathbf{Z}$ のとき $\deg(f) = |f|$, $R = \mathbf{k}[x]$ のときは $\deg(f) = f$ の次数 と定義する

ユークリッド整域はこの割算定理の成立を仮定した整域であり, ユークリッド整域で議論を展開しておくことにより, 整数での議論も一変数多項式での議論も共通化が可能である. 計算機科学における, Object 指向, 部品化, 抽象データ型等の概念も, このような現代数学の考え方— 抽象化, 公理化— と同じである. 現代数学では, このような思考の節約は多くの分野で有効であったが, それが数学の全てではない. 同じように計算機科学における Object 指向や抽象データ型の概念 ( Java などで実現されている) は, 有効な局面も多くあったが, 万能というわけではないことを注意しておこう.

## 15.2　単項イデアルと 1 変数連立代数方程式系の解法

“ユークリッド整域” では, 整数のときの互除法アルゴリズムがつかえる. 互除法アルゴリズムを用いることにより, 一変数多項式環のイデアルに関する多くの問題を解くことが可能である.

$f, g \in \mathbf{Q}[x]$ に対して, 一変数の連立代数方程式

$$f(x) = g(x) = 0$$

の共通根をもとめることを考えよう. (複素) 共通根の集合を

$$V(f, g) = \{a \in \mathbf{C} \,|\, f(a) = g(a) = 0\}$$

と書くことにする. 私達の考えたい問題は, $V(f, g)$ が空かそれとも何個の元からなっているか? 空でないとして, 根の近似値を求めることである.

この問題を見通しよく考えるには, イデアルの考えをもちいるとよい. $I$ を $f$, $g$ の生成するイデアルとしよう. つまり, $\mathbf{Q}[x]$ の部分集合

$$I = \langle f, g \rangle = \{p(x)f(x) + q(x)g(x) \,|\, p, q \in \mathbf{Q}[x]\}$$

を考える. このとき,

$$V(f, g) = V(I) = \{a \in \mathbf{C} \,|\, h(a) = 0 \text{ for all } \ h \in I\}$$

である.

さて, $I$ は単項生成なので,

$$I = \langle h \rangle$$

となる生成元 $h$ が存在する. $V(I) = V(h)$ であることが容易に分かるので, $h$ が定数なら, $V(I)$ は空集合であり, そうでないときは, 重複度も込みで, $V(I)$ の個数は, $h$ の次数にほかならない. $h$ は $f$, $g$ の GCD にほかならないことが証明できるので, 結局, 互除法アルゴリズムで $h$ を $f$, $g$ より計算して, それから, $h = 0$ を数値的にとけば, $V(f, g)$ を決定できることになる.

以上をプログラムすると以下のようになる. 関数 g_c_d(F,G) は多項式 F と G の最大公約多項式 (GCD) を求める. 関数 division(F,G) は割算定理をみたす, $q$, $r$ を求めている. variety(F,G) で共通根の計算をおこなう.

次の関数達をすべて含めたファイルが gcd.rr である.

```
def in(F) {
  D = deg(F,x);
  C = coef(F,D,x);
  return(C*x^D);
}
```

```
def division(F,G) {
   Q = 0; R = F;
   while ((R != 0) && (deg(R,x) >= deg(G,x))) {
      D = red(in(R)/in(G));
      Q = Q+D;
      R = R-D*G;
   }
   return([Q,R]);
}
```

```
def g_c_d(F,G) {
   if (deg(F,x) > deg(G,x)) {
     S = F; T = G;
   }else {
     S = G; T = F;
   }
   while (T != 0) {
     R = division(S,T)[1];
     S = T;
     T = R;
   }
   return(S);
}
```

```
def variety1(F,G) {
   R = g_c_d(F,G);
   if (deg(R,x) == 0) {
     print("No solution.(variety is empty.)");
     return([]);
   }else{
     Ans = pari(roots,R);
     print("The number of solutions is ",0); print(size(Ans)[0]);
     print("The variety consists of  : ",0); print(Ans);
     return(Ans);
   }
}
end$
```

上のプログラムで利用されている組み込み関数について解説を加えておこう.

1. `deg(F,x)`： 多項式 `F` の変数 `x` についての次数をもどす. たとえば, `deg(x^2+x*y+1,x)` は 2 を戻す.
2. `coef(F,D,x)` : 多項式 `F` の変数 `x` の `D` 次の係数を戻す. すなわち, 多項式 `F` を変数 `x` の 1 変数多項式とみたとき `x^D` の係数を戻す. たとえば `coef(x^2+x*y+2*x+1,1,x)` は `y+2` を戻す.

よりくわしくは, `help` コマンドでマニュアルを参照してほしい.
例. $x^4-1=0$ と $x^6-1=0$ の共通根の集合, $V(x^4-1, x^6-1)$ の計算をしてみよう.

```
[346] load("gcd.rr");
1
[352] variety1(x+1,x-1);
No solution.(variety is empty.)
[]
[353] variety1(x^4-1,x^6-1);
The number of solutions is 2
The variety consists of  : [ -1.0000000000000000000 1.0000000000000000000 ]
[ -1.0000000000000000000 1.0000000000000000000 ]
[354]
```

あとの節でみるように, ユークリッドの互除法は数学において基本的のみならず, RSA 暗号系の基礎としても利用されており, 現代社会の基盤技術としても重要である. 蛇足ながら, こんな八方美人な数学の話はそうめったにないのも注意しておこう.

**問題 15.1** 3 つの多項式の共通零点を求めるプログラムを書きなさい.

**問題 15.2** 多項式環における一次不定方程式

$$p(x)f(x)+q(x)g(x)=d(x)$$

の解を一つ求めるアルゴリズムを考え, そのプログラムを書きなさい. ここで, $f$, $g$, $d$ が与えられた一変数多項式で, $p$, $q$ が未知である.

**問題 15.3** 来週数学のテスト?! プログラミングなんかしてらんない! ちょっとまった. 数学の教科書をみながら, いろんなプログラミングを考えてみるのはどうでしょう. この節でみたように, たとえばユークリッド環とそのイデアルについてプログラミングをすれば, 対象の理解がぐんとすすみます. 教科書を読んでわからなかったこともわかるようになるかも.

補足: ここでは, いくつかの一変数多項式が与えられたとき, それらが生成するイデアルの生成元が互除法で求められることを見た. そこで求めた生成元は, イデアルの中で 0 を除く最低次数のものであり, ある多項式がそのイデアルに属するかどうかは, 求めた生成元による割算の結果で判定できる. 多変数の場合, 一般にイデアルは単項生成にはならないが, 単項式の中にある種の全順序を入れることで, 剰余が一意的に計算できるような生成系 (グレブナ基底) を考えることができる. グレブナ基底を求めるアルゴリズムとして Buchberger アルゴリズムがあるが, それは互除法の拡張と思ってよい. グレブナ基底は多変数多項式の共通零点を求めるだけでなく, 理論的にも重要な役割を演じる. 詳しくは, 16 章および [1] または [2] を参照.

Risa/Asir でグレブナ基底を計算するコマンドは, `gr` か `hgr` である. グレブナ基底の計算は, 互除法の拡張であるので, `gr` を用いても GCD を計算できる. `x` の多項式 `F` と `G` の GCD は, 集合 {`F`, `G`} のグレブナ基底であるので, コマンド `gr([F,G],[x],0);` でも計算できる.

## 15.3　計算効率

前節の関数 `g_c_d` で, $f = (2x^3 + 4x^2 + 3)(3x^3 + 4x^2 + 5)^{10}$, $g = (2x^3 + 4x^2 + 3)(4x^3 + 5x^2 + 6)^{10}$ の GCD を計算してみよう.

```
[151] F=(2*x^3+4*x^2+3)*(3*x^3+4*x^2+5)^10$
[152] G=(2*x^3+4*x^2+3)*(4*x^3+5*x^2+6)^30$
[153] H=g_c_d(F,G)$
6.511sec + gc : 0.06728sec(6.647sec)
```

使用する計算機にもよるが, 数秒程度で巨大な係数を持つ多項式が得られる. 実はこの多項式は $2x^3 + 4x^2 + 3$ の定数倍である. これを組み込み関数 `ptozp(F)` で確かめてみよう. `ptozp(F)` は, `F` に適当な有理数をかけて, 係数を GCD が 1 であるような整数にした多項式を返す関数である.

```
[154] ptozp(H);
2*x^3+4*x^2+3
```

この例からわかるように, 前節の `g_c_d` では, 互除法の途中および結果の多項式に分母分子が巨大な分数が現れてしまう. 人間と同様, 計算機も分数の計算は苦手である. そこで, 分数の計算が現れないように工夫してみよう. まず, 剰余を定数倍しても, GCD は定数倍の影響を受けるだけということに注意して, 次のような関数を考える.

```
def remainder(F,G) {
   Q = 0; R = F;
   HCG = coef(G,deg(G,x));
   while ((R != 0) && (deg(R,x) >= deg(G,x)))
      R = HCG*R-coef(R,deg(R,x))*x^(deg(R,x)-deg(G,x))*G;
   return R;
}
```

この関数は, 適当な自然数 $k$ に対し

$$\mathrm{lc}(g)^k f = qg + r, \quad \deg(r) < \deg(g)$$

($\mathrm{lc}(g)$ は $g$ の最高次の係数) なる $r \in \mathbf{Z}[x]$ を求めていることになる. この関数で, 前節の `division` を置き換えてみよう.

```
def g_c_d_1(F,G) {
   if (deg(F,x) > deg(G,x)) {
     S = F; T = G;
   }else {
     S = G; T = F;
   }
   while (T != 0) {
     R = pseudo_remainder(S,T);
     S = T;
     T = R;
   }
   return(S);
}
```

```
[207] g_c_d_1(F,G);
Needed to allocate blacklisted block at 0x988d000
Needed to allocate blacklisted block at 0x9899000
```

どうしたことか, 妙なメッセージは出るものの結果は出そうもない. 実は, `pseudo_remainder` でかけた $\mathrm{lc}(g)^k$ のせいで, 途中の多項式の係数が大きくなりすぎているのである. そこで, `pseudo_remainder` の結果を `ptozp` で簡単化してみよう.

```
def g_c_d_2(F,G) {
   if (deg(F,x) > deg(G,x)) {
     S = F; T = G;
   }else {
     S = G; T = F;
   }
   while (T != 0) {
     R = pseudo_remainder(S,T);
     R = ptozp(R);
     S = T;
     T = R;
   }
   return(S);
}
```

```
[237] g_c_d_2(F,G);
2*x^3+4*x^2+3
0.057sec(0.06886sec)
```

今度はずいぶん速く計算できた．ptozp では, 実際に係数の整数 GCD を計算することで簡単化を行っているが, より詳しく調べると, GCD を計算しなくても, GCD のかなりの部分はあらかじめ知ることができる．この話題にはこれ以上立ち入らない．[3] Section 4.6.1 または [2] 5.4 節 を参照して欲しい．

ここで見たように, 互除法のような単純なアルゴリズムでも, 実現方法によってはずいぶん効率に差が出る場合がある．特に, 分数が現れないようなアルゴリズムを考えることは重要である．

**問題 15.4** g_c_d も ptozp を用いることで高速化できる．その改良版 g_c_d と g_c_d_2 をさまざまな例で比較してみて, 分数が現れる演算が効率低下を招くことを確認せよ．

# 関連図書

[1] D.Cox, J.Little, D.O'Shea, *Ideals, Varieties, and Algorithms — An Introduction to Commutative Algebraic Geometry and Commutative Algebra*, 1991, Springer-Verlag.
日本語訳: D. コックス, J. リトル, D. オシー: グレブナ基底と代数多様体入門 (上/下). 落合他訳, シュプリンガー フェアラーク 東京, 2000. ISBN 4-431-70823-5, 4-431-70824-3.
世界的に広く読まれているグレブナ基底の入門書. Buchberger アルゴリズム自体は, 2 章までよめば理解できる. Risa/Asir ドリルの 15 章 (本章) および 16 章 (次の章) はコックス達の本をもとにした, グレブナ基底の入門講義等の補足プリントがもとになっている. したがってコックス達の本とともに本章と次の章を読むと理解が深まるであろう. 本章で証明や説明を省略した数学的事実や概念については, コックス達の本の 1 章を参照されたい. 大学理系の教養課程の数学の知識で十分理解可能である.

[2] 野呂: 計算代数入門, Rokko Lectures in Mathematics, 9, 2000. ISBN 4-907719-09-4.
`http://www.math.kobe-u.ac.jp/Asir/ca.pdf` から, PDF ファイルを取得できる.
`http://www.openxm.org` より openxm のソースコードをダウンロードすると, ディレクトリ `OpenXM/doc/compalg` にこの本の TeX ソースがある.

[3] D.E. Knuth: *The Art of Computer Programming*, Vol2. Seminumerical Algorithms, 3rd ed. Addison-Wesley (1998). ISBN 0-201-89684-2.
日本語訳: "準数値算法", サイエンス社.

# 第16章　割算アルゴリズムとグレブナ基底

## 16.1　Initial term の取り出し

この節はイデアルの話題の続きである. 前の節で議論した, GCD を計算するアルゴリズムを用いると, 1 変数の多項式環の場合, 多項式 $f$ が イデアル $I$ に入っているかどうかを, 計算機を用いて確かめることが可能である. イデアル $I$ が, 多項式 $p$ と $q$ で生成されているとしよう. このとき $h$ を $p$ と $q$ の GCD とすると, $f$ がイデアル $I$ に入っているための必要十分条件は, $\texttt{division}(f,h)[1]$ が 0 を戻すことである. つまり, $f$ が $h$ で割り切れればよい. これは $I=\langle h\rangle$ であることから明らかであろう. まとめると,

1. 入力 $p$, $q$ に対して, 前節の互除法のアルゴリズムによりイデアルの生成元 $h$ を求める.
2. $f\in I$ かどうか判定したい多項式 $f$ の $\texttt{division}(f,h)$ を計算して, その第 2 成分 (割算した余り) が 0 なら, $f\in I$ と答え, 0 でないなら $f\notin I$ と答える.

この判定アルゴリズムをイデアルメンバシップアルゴリズム (ideal membership algorithm) とよぶこともある.

2 変数以上のイデアルは, 単項生成とは限らない. たとえば,

$$I=\langle x^2,xy,y^2\rangle=\mathbf{Q}[x,y]x^2+\mathbf{Q}[x,y]xy+\mathbf{Q}[x,y]y^2$$

を考えると $I=\langle h\rangle$ となるような $h$ は存在しない. なぜなら, そのような $h$ があれば, $x^2$, $xy$, $y^2$ を同時に割り切らないといけないが, そのような $h$ は定数のみである. $h$ が定数だと $I$ は $\mathbf{Q}[x,y]$ に等しくなってしまうが, $\langle x^2,xy,y^2\rangle$ はたとえば 1 を含まない (証明せよ). よって $I$ は単項生成でない.

このアルゴリズムは多変数多項式に一般化できる. この一般化されたアルゴリズムの中心部分が Buchberger アルゴリズムである. Buchberger アルゴリズムは, イデアルに入っているかどうかの判定だけでなく, 代数方程式系を解いたり, 環論や代数幾何や多面体や微分方程式論などにでてくるさまざまな数学的対象物を構成するための基礎となっている重要なアルゴリズムである.

この節では Buchberger アルゴリズム自体の解説は優れた入門書がたくさんあるので (たとえば [2], [3] など), それらを参照してもらうことにして, Buchberger アルゴリズムの中で重要な役割をはたす割算アルゴリズム (または reduction アルゴリズム) について計算代数システムの内部構造の立場からくわしく考察する. それから Buchberger アルゴリズムとグレブナ基底を簡潔に解説して, イデアルメンバシップアルゴリズムを説明する. 最後にグレブナ基底の別の応用である連立方程式の求解について簡単にふれる.

イデアルメンバシップアルゴリズムを多変数へ拡張するには 1 変数の場合と同様にまず initial term を計算する関数を用意しないといけない.

注意すべきは, 2 変数以上の場合, さまざまな順序を考えることが可能となることである. 以下 3 変数多項式環 $\mathbf{Q}[x,y,z]$ で議論する. たとえば $z\succ_{lex}y\succ_{lex}x$ なる辞書式順序 (lexicographic order) は次のように定義する.

$$Px^ay^bz^c\succ_{lex}P'x^{a'}y^{b'}z^{c'}$$

$$\Leftrightarrow \quad (c-c', b-b', a-a') \text{ の最初の } 0 \text{ でない成分が正}$$

つまり, まず $z$ の巾で大小を比較して, $z$ の巾が同じなら次に $y$ の巾で比較し, $y$ の巾も同じなら, $x$ の巾で比較せよという順序である. なおここで $P$, $P'$ は係数であり, 係数は順序の比較に影響しない.

別の順序を考えることも可能である. たとえば, $w = (w_1, w_2, w_3) \in \mathbf{R}^3$ を $w_i \geq 0$ であるようなベクトルとして,

$$\begin{aligned} & Px^a y^b z^c \succ_w P' x^{a'} y^{b'} z^{c'} \\ \Leftrightarrow \quad & d = (a-a')w_1 + (b-b')w_2 + (c-c')w_3 > 0 \\ & \text{または } (d = 0 \text{ かつ } Px^a y^b z^c \succ_{lex} P' x^{a'} y^{b'} z^{c'}) \end{aligned}$$

と順序 $\succ_w$ を定義する. これを重みベクトル $w$ を辞書式順序で細分して得られた順序という.

あたえられた順序 $\succ$ ($\succ$ は $\succ_{lex}$ か $\succ_w$) と多項式 $f$ に対して, 多項式 $f$ の中の一番大きいモノミアル (単項式) をその多項式の initial term といい $\mathrm{in}_\succ(f)$ と書く.

次の関数 `in(F)` および `in2(F)` はそれぞれ, $z > y > x$ および $x > y > z$ なる辞書式順序 (lexicographic order) で initial term をとりだす関数である.

```
def in(F) {
   D = deg(F,z);
   F = coef(F,D,z);
   T = z^D;
   D = deg(F,y);
   F = coef(F,D,y);
   T = T*y^D;
   D = deg(F,x);
   F = coef(F,D,x);
   T = T*x^D;
   return [F*T,F,T];
}
```

```
def in2(F) {
   D = deg(F,x);
   F = coef(F,D,x);
   T = x^D;
   D = deg(F,y);
   F = coef(F,D,y);
   T = T*y^D;
   D = deg(F,z);
   F = coef(F,D,z);
   T = T*z^D;
   return [F*T,F,T];
}
```

結果はリストで戻しており, リストの先頭要素が多項式 `F` の initial term, リストの 2 番目の要素が initial term の係数, リストの 3 番目の要素が initial term の係数を除いたモノミアル部である.

## 16.2　多項式の内部表現と initial term の取り出しの計算効率

前の節の二つの関数 `in(F)` および `in2(F)` には違いがないと思うかもしれないが, 次の関数で実行時間をはかるとおおきな違いがある. between `in(F)` and `in2(F)`.

```
def test1() {
  F = (x+y+z)^4;
  for (I=0; I<1000; I++) G=in(F);
  return(G);
}
def test2() {
  F = (x+y+z)^4;
  for (I=0; I<1000; I++) G=in2(F);
  return(G);
}
```

```
[764] cputime(1);
0
0sec(0.000193sec)
[765] test2();
[z^4,1,z^4]
0.06sec + gc : 0.08sec(0.1885sec)
[766] test1();
[x^4,1,x^4]
0.2sec + gc : 0.08sec(0.401sec)
```

`test1` と `test2` では, 3 倍ちかくの実行時間の差がある. これはどういった理由からであろうか?

理由を理解するには計算機のメモリに多項式がどのように格納されているかを考えてみる必要がある.

Asir ではメモリにたとえば多項式 $5x^2$ が格納されるときには, つぎのようなセルとして格納されている.

| 主変数名 | x |
|---|---|
| 主変数にかんする次数 | 2 |
| 係数 | 5 |
| 次のモノミアルのアドレス | なし |

このようなセルの和でモノミアルの和を表現する. たとえば, $5x^2+x$ であれば, 次のようなセルの集まりとして格納されている.

| 主変数名 | x |
|---|---|
| 主変数にかんする次数 | 2 |
| 係数 | 5 |
| 次のモノミアルのアドレス | AAAA |

AAAA:

| 主変数名 | x |
|---|---|
| 主変数にかんする次数 | 1 |
| 係数 | 1 |
| 次のモノミアルのアドレス | なし |

$5x^3+5xz^2+xz$ を asir に入力すると,

```
[755] 5*x^3+5*x*z^2+x*z;
5*x^3+(5*z^2+z)*x
```

と $x$ の多項式として表示されるであろう. メモリ内部には, 上で説明したしたように, $x$ に関するモノミアルを表現するセルのリストとして表現される. 上の説明との違いは, $x$ の係数が, $z$ の多項式でありそれが $z$ に関するモノミアルを表現するセルのリストとして表現されているという再帰的構造を持つことである. 図示すると次のようになる.

| 主変数名 | x |
|---|---|
| 主変数にかんする次数 | 2 |
| 係数 | 5 |
| 次のモノミアルのアドレス | AAAA |

AAAA:

| 主変数名 | x |
|---|---|
| 主変数にかんする次数 | 1 |
| 係数のアドレス | BBBB |
| 次のモノミアルのアドレス | なし |

BBBB:

| 主変数名 | z |
|---|---|
| 主変数にかんする次数 | 2 |
| 係数 | 5 |
| 次のモノミアルのアドレス | CCCC |

CCCC:

| 主変数名 | z |
|---|---|
| 主変数にかんする次数 | 1 |
| 係数 | 1 |
| 次のモノミアルのアドレス | なし |

この表現をみれば, $x$ についての最高次の項を取り出す操作は 1 ステップで終るのにたいし, $z$ についての最高次の項をとり出す操作は, $x$ の各次数の係数の $z$ についての最高次の項を取り出してから, その中でさらに一番大きいものをとりださねばならないことが分かるであろう. こちらの方がはるかに複雑であるので, 時間がかかるのである.

C 言語を知ってる読者は 図 16.1 のような構造体を用いて, このような多項式の足し算および `deg` 関数 `coef` 関数の実装を試みると理解が深まるであろう.

```
union object_body {
  struct polynomial_cell f;
  int c;
};
struct object {
  int tag;
  union object_body obj;
};
struct polynomial_cell {
  char *v;  /* variable name */
  int   e;  /* degree */
  struct object *coef;  /* coefficients */
  struct polynomial_cell *next;
};
```

図 16.1: 多項式を表現する構造体

Asir では, このような表現による多項式を再帰表現多項式とよぶ. この型のデータに対して type 関数の戻す値は 2 である.

さて, あたえられた順序に対して, このような多項式の表現法では, initial term を取り出す効率が順序によりかわってしまうし, 効率の点からは最適な表現とはいえない. initial term を取り出すには, 多項式は多変数のモノミアルのあたえらた順序に関するリストとして表現するのがよいであろう. この表現方法を, asir では, 分散表現多項式とよんでいる. たとえば, 多項式 $5x^3+5xz^2+xz$ を $x>y>z$ なる lexicographic order での分散表現多項式としてメモリに格納するときにはつぎのようなリストとして表現する.

| $x,y,z$ についての次数 | [3,0,0] |
|---|---|
| 係数 | 5 |
| 次のモノミアルのアドレス | DDDD |

DDDD :

| $z,y,x$ についての次数 | [1,0,2] |
|---|---|
| 係数 | 5 |
| 次のモノミアルのアドレス | EEEE |

EEEE :

| $z,y,x$ についての次数 | [1,0,1] |
|---|---|
| 係数 | 1 |
| 次のモノミアルのアドレス | なし |

このように表現しておくことにより, initial term の取り出しはリストの先頭項を取り出すだけの操作ですむことになる.

次の関数では, initial term の取り出し 1000 回のテストを, 分散表現多項式に対しておこなっている. `dp_ptod(F)` は `F` を分散表現多項式に変換する関数である. 詳しくはマニュアルを参照.

```
def test1_dp() {
  F = (x+y+z)^4;
  dp_ord(2);
  F = dp_ptod(F,[x,y,z]);
  /* print(F); */
  for (I=0; I<1000; I++) G=dp_hm(F);
  return(G);
}
```

とりだしにかかる時間は以下のとおりである.

```
[778] test1_dp();
0.01sec(0.01113sec)
```

## 16.3 割算アルゴリズム

1 変数多項式に対する割算アルゴリズムは, initial term に注目することにより多変数に拡張できる. すなわち, 多項式 $f$ のある項 $t$ が, 多項式 $g$ の initial term で割り切れるとき, 適当な単項式 $m$ を選んで $f-mg$ に $t$ が現れないようにできる. この操作を reduction と呼ぶ. reduction を繰り返して, それ以上 reduction できない多項式を得たとき, それを割算による剰余とする.

3 変数 $x,y,z$ の場合のプログラム.

```
def multi_degree(F) {
  F = in(F);
  T = F[2];
  return([deg(T,x),deg(T,y),
          deg(T,z),F[1]]);
}
```

$x > y > z$ なる辞書式順序での initial term を取り出す
各変数の次数および, initial term の係数をリストにして返す

```
def is_reducible(F,G) {
  DF = multi_degree(F);
  DG = multi_degree(G);
  if (DF[0] >= DG[0]
   && DF[1] >= DG[1]
   && DF[2] >= DG[2])
    return red(DF[3]/DG[3])
           *x^(DF[0]-DG[0])
           *y^(DF[1]-DG[1])
           *z^(DF[2]-DG[2]);
  else return 0;
}
```

F が G で reducible でないとき (reduction できないとき) 0 を戻す. F が G で reducible のとき (reduction できるとき) M G が F の initial term に等しいような, M を戻す. 別の言い方をすると, in(F) = in(M) in(G) となるモノミアル M を戻す.

```
def division(F,G) {
  Q = 0; R = F;
  D = is_reducible(R,G);
  while (type(D) != 0) {
    Q = Q+D;
    R = R-D*G;
    D = is_reducible(R,G);
  }
  return [Q,R];
}
```

G の initial term が F の initial term を割り切る限り, F から G の単項式倍を引いて F initial term を消去する.
この関数の出力は, initial term が G の initial term で割り切れないことは保証される.

いくつかの元を含む集合 G による剰余計算も可能である. どの項を reduction するかに任意性があるため, 一般には結果は 1 通りとは限らないことに注意しておく.

プログラム

```
def reduction(F,G)
{
  Rem = 0;
  while ( F ) {
    for ( U = 0, L = G; L != [];
          L = cdr(L) ) {
      Red = car(L);
      Mono = is_reducible(F,Red);
      if ( Mono != 0 ) {
        U = F-Mono*Red;
        if ( !U )
          return Rem;
        break;
      }
    }
    if ( U )
      F = U;
    else {
      H = in(F);
      Rem += H[0];
      F -= H[0];
    }
  }
  return Rem;
}
```

G は多項式のリストである. この関数では, F の先頭項が, G のどの要素の initial term によっても割り切れない場合に, F から先頭項をとりはずし, Rem に追加される. この関数の出力は, どの項も, G のどの要素の initial term によっても割り切れないような多項式である.

実行例

```
[216] reduction(x^2+y^2+z^2,
      [x-y*z,y-z*x,z-x*y]);
2*x^2+y^2
[217] reduction(x^2+y^2+z^2,
      [z-x*y,y-z*x,x-y*z]);
(y^2+1)*x^2+y^2
```

この例では, G の要素の並び方により, 結果が異なっている.

## 16.4 グレブナ基底

$\prec$ を 前節で考えた順序 $\prec_{lex}$ か $\prec_w$ とする. $f_1, \ldots, f_p$ を $n$ 変数多項式環 $S_n = \mathbf{Q}[x_1, \ldots, x_n]$ に属する多項式としよう. $S_n$ の部分集合

$$I = \langle f_1, \ldots, f_p \rangle := S_n f_1 + \cdots + S_n f_p$$

を考える. $I$ はイデアルである. 多項式の有限集合

$$G = \{g_1, \ldots, g_m\}$$

が次の二つの条件を満たすときイデアル $I$ の順序 $\prec$ に関するグレブナ基底であるという.

1. $I = \langle g_1, \ldots, g_m \rangle$ ( $g_1, \ldots, g_m$ が $I$ を生成. )
2. $\mathrm{in}_\prec(I) := \langle \mathrm{in}_\prec(f) \,|\, f \in I \rangle$ が $\langle \mathrm{in}_\prec(g_1), \ldots, \mathrm{in}_\prec(g_m) \rangle$ に等しい.

例として, 1 変数多項式で 順序 $1 < x < x^2 < \cdots$ の場合を考える. $f_1, \ldots, f_s$ の GCD を $h$ とすると, $G = \{h\}$ はグレブナ基底である. グレブナ基底は余分な元をふくんでいてよくて, たとえば, $G = \{f_1, \ldots, f_p, h\}$ も $I$ のグレブナ基底となる.
問題 (代数学既習者向け): ヒルベルトの基底定理を用いてグレブナ基底の存在を証明せよ.

グレブナ基底の構成をおこなうアルゴリズムが Buchberger アルゴリズムである. まず, S 多項式の計算について簡単に説明する. 二つの多項式 $f$, $g$ に対し, $f$, $g$ の initial term をそれぞれ $c_f t_f$, $c_g t_g$ とするとき, $\{f, g\}$ の S 多項式は

$$\mathrm{Spoly}(f, g) = \frac{\mathrm{LCM}(t_f, t_g)}{c_f t_f} f - \frac{\mathrm{LCM}(t_f, t_g)}{c_g t_g} g$$

で定義される. ここで $c_f$ は数, $t_f$ は係数 1 のモノミアル, LCM の意味は下のプログラムから理解頂きたい.

```
def spolynomial(F,G)
{
  DF = multi_degree(F);
  DG = multi_degree(G);
  Mx = DF[0]>DG[0]?DF[0]:DG[0];
  My = DF[1]>DG[1]?DF[1]:DG[1];
  Mz = DF[2]>DG[2]?DF[2]:DG[2];
  return x^(Mx-DF[0])*y^(My-DF[1])*z^(Mz-DF[2])*F/DF[3]
        -x^(Mx-DG[0])*y^(My-DG[1])*z^(Mz-DG[2])*G/DG[3];
}
```

このプログラムで `A?B:C` なる表現があるが, これは, `A` が 0 なら `C` を戻し, `A` が 0 でないならば, `B` を戻す.

**定理 16.1** $I = \langle G \rangle$ とするとき, $G$ が $I$ のグレブナ基底であることと, $G$ から作られる $S$ 多項式の `reduction` による剰余がすべて *0* になることとは同値である.

この定理の証明は, 例えば [2, 2 章 6 節 定理 6] にある. これよりただちに次の Buchberger アルゴリズムを得る.

1. $G = F$, $D = \{G$ から作られるペア全体 $\}$ とする. (初期化)
2. $D$ が空なら $G$ がグレブナ基底.
3. $D$ から 1 つペア $s$ を取り除き, $s$ の S 多項式を $G$ による剰余ををを $r$ とする.
4. もし $r$ が 0 でなければ, $r$ と $G$ から作られるペアを $D$ に追加する. さらに, $r$ を $G$ に追加する. (この時点で $s$ の S 多項式の $G$ による剰余は 0 となる. )
5. 2. に戻る.

**問題 16.1** このアルゴリズムは有限時間内に停止することを証明せよ. (ヒント: $G$ の各要素の initial term の生成するイデアルを上のアルゴリズムの 2 で考えよ. 停止しないとすると, イデアルの真の無限増大列が作れる.)

グレブナ基底の応用はいろいろなものがあるが, たとえばグレブナ基底により, ある多項式がイデアルに属するかどうかが割算により判定できる.

**定理 16.2 (イデアルメンバシップ)** $G$ をイデアル $I$ のグレブナ基底とするとき, $f \in I$ であることと, $f$ を $G$ で reduction した剰余が 0 になることは同値である.

イデアルの元を, そのイデアルの基底で reduction した剰余もやはり同じイデアルの元になる. これとグレブナ基底の定義により定理が成り立つことがわかる (この定理の詳しい証明については [2, 2 章 6 節 系 2] を参照).

さらに, グレブナ基底の重要な性質として, どのように割算を行なっても結果が一意的である, ということがある. これも, 上の定理によりすぐにわかるであろう.

3 変数の場合の Buchberger アルゴリズムを実行するプログラムを書いてみる.

プログラム

```
def buchberger(F)
{
  N = length(F);
  Pairs = [];
  for ( I = N-1; I >= 0; I-- )
    for ( J = N-1; J > I; J-- )
      Pairs = cons([I,J],Pairs);
  G = F;
  while ( Pairs != [] ) {
    P = car(Pairs);
    Pairs = cdr(Pairs);
    Sp = spolynomial(G[P[0]],G[P[1]]);
    Rem = reduction(Sp,G);
    if ( Rem ) {
      G = append(G,[Rem]);
      for ( I = 0; I < N; I++ )
        Pairs = cons([I,N],Pairs);
      N++;
    }
  }
  return G;
}
```

Buchberger アルゴリズムは, イデアルの基底 G に属する二つの多項式からつくられる S 多項式に対し, reduction 関数により剰余を計算して, G に追加していく.

実行例

```
[218] F=[x-y*z,y-z*x, z-x*y];
[219] G=buchberger(F);
[x-z*y,-z*x+y,-y*x+z,-x^2+y^2,
 -y*x^3+y*x,-x^5+x^3,-x^4+x^2,
 x^3-x,-y*x^2+y]
[220] reduction(x^2+y^2+z^2,G);
3*x^2
[221] reduction(x^2+y^2+z^2,
                reverse(G));
3*x^2
```

この実行例は, 前節で例に用いたイデアルのグレブナ基底を計算してみたものである. 順序を変えて reduction しても, 今度は同じ結果になっていることに注目してほしい.

上のプログラムは Buchberger アルゴリズムのもっともシンプルな形である. 残念ながら, このままでは, ごく簡単な例しか計算できない. 次のような点から種々の改良が行なわれており, それらを実装してはじめて実用的に使えるものが得られる.

1. Pairs から 1 つペアを選ぶ方法を指定する.
2. Pairs から, あらかじめ不必要なペアを取り除く.

3. 有理数係数の場合には, 分数が現れないような計算の工夫.

これらについては [3] でくわしく解説されている.

Asir に組み込まれている函数 gr は与えらた多項式と順序に対して, グレブナ基底を戻す. もちろん, 上に挙げたようなさまざまな改良が実装されている. (OpenXM 版でない Asir では, `load("gr");` でまずグレブナ基底用のライブラリをロードしておく必要がある.)

gr(イデアルの生成元, 変数, 順序の指定)

例: `gr([x^2+y^2-1, x*y-1],[y,x], 2);`

順序は 0 が graded reverse lexicographic order, 1 が graded lexicographic order, 2 が lexicographic order である. 行列を用いた一般の順序の指定も可能だが, それはマニュアルを見よ. 上の例では, $x^2+y^2-1$ と $xy-1$ で生成されるイデアルの $y > x$ なる lexicographic order に関するグレブナ基底を戻す. 答えは,

$$\{\underline{-x^4} + x^2 - 1, \underline{y} + x^3 - x\}$$

である.

**例題 16.1** (大事)

1. $R = \mathbf{Q}[x,y]/\langle x^2+y^2-1, xy-1\rangle$ は, $\mathbf{Q}$ 上のベクトル空間であることを示し, 基底および次元を求めよ. 順序としては $y > x$ なる lexicographic order を用いよ.

2. $\mathbf{Z}/3\mathbf{Z}$ の乗法表のような形式で, $R$ の乗法表を作れ. 自分で作成した割算函数を用いてみよ.

まず, ブレブナ基底の initial term が $-x^4$, $y$ なので, これらで割れないモノミアルで係数 1 のものは,

$$1, x, x^2, x^3$$

である. これらが $R$ の $\mathbf{Q}$ ベクトル空間としての基底になる. したがって, $R$ の $\mathbf{Q}$ ベクトル空間としての次元は 4 である. つぎにここでは, Asir に組み込みの割算函数 p_true_nf を用いて乗法表を作成してみる.

```
[713] G = gr([x^2+y^2-1,x*y-1],[y,x],2)$
[714] G;
[-x^4+x^2-1,y+x^3-x]
[715] p_true_nf(x^4,G,[y,x],2);
[-x^2+1,-1]
[716] p_true_nf(x^5,G,[y,x],2);
[-x^3+x,-1]
[717] p_true_nf(x^6,G,[y,x],2);
[-1,1]
```

以上の出力より, $R$ での次の乗法表を得る.

| | 1 | $x$ | $x^2$ | $x^3$ |
|---|---|---|---|---|
| 1 | 1 | $x$ | $x^2$ | $x^3$ |
| $x$ | | $x^2$ | $x^3$ | $x^2-1$ |
| $x^2$ | | | $x^2-1$ | $x^3-x$ |
| $x^3$ | | | | $-1$ |

掛け算は可換なので, 下半分は省略してある.

**例 16.1** 上で書いた buchberger と, Asir の gr の速度比較をしてみよう.

```
[284] F = [-3*x^3+4*y^2+(-2*z-3)*y+3*z^2,(-8*y-4)*x+(2*z+3)*y,
           -2*x^2-3*x-2*y^2+2*z*y-z^2]$
[285] V = [x,y,z]$
[286] cputime(1)$
2.3e-05sec(1.895e-05sec)
[287] buchberger(F)$
10.38sec + gc : 0.6406sec(11.41sec)
[288] gr(F,V,2)$
0.05449sec(0.05687sec)
```

この例では gr が圧倒的に高速である. しかし, 入力をほんの少し変更すると次のようなことが起こる.

```
[289] F = [-3*x^3+4*y^2+(-2*z-3)*y+3*z^2,(-8*y-4)*x^2+(2*z+3)*y,
           -2*x^2-3*x-2*y^2+2*z*y-z^2]$
8.7e-05sec(7.892e-05sec)
[290] buchberger(F)$
49.02sec + gc : 3.573sec(53.4sec)
[291]  gr(F,V,2)$
163.1sec + gc : 4.694sec(168.2sec)
[292]  hgr(F,V,2)$
0.02296sec(0.02374sec)
```

これでわかるように, gr に実装されている改良も, あらゆる入力に対して有効なわけではなく, かえって悪影響を及ぼす場合もある. hgr というのは, gr にある前処理, 後処理を付け加えたもので, おおむね安定だが, やはり遅くなる場合もある. 計算効率の点では, グレブナ基底計算はまだ発展途上であり, 改良や新しいアルゴリズムも発表され続けている. 200310 以降の Asir には nd_ から始まる新しいグレブナ基底計算関数がテスト実装されている;
nd_gr(多項式リスト，変数リスト，標数，順序). 有限体上の計算の場合 gr より 数倍から数十倍高速である.

```
[1039] F = [-3*x^3+4*y^2+(-2*z-3)*y+3*z^2,(-8*y-4)*x^2+(2*z+3)*y,
            -2*x^2-3*x-2*y^2+2*z*y-z^2]$
[1040] nd_gr(F,[x,y,z],13,2);
ndv_alloc=4896
[z^14+4*z^13+2*z^12+7*z^11+ ....  ]
```

なお標数 0, 辞書式順序での計算は係数膨張を招きやすいので, 直接計算をする場合には nd_gr_trace(多項式リスト, 変数リスト,1,1,2) (斉次化つきトレースアルゴリズム) を使うほうが安全である.」

## 16.5　グレブナ基底と多変数連立代数方程式系の解法

多変数の連立方程式系

$$f_1(X) = \cdots = f_m(X) = 0 \tag{16.1}$$

($X = (x_1, \ldots, x_n)$, $f_i(X) \in \mathbf{Q}[X]$) を解く場合にも, イデアル

$$I = \langle f_1, \ldots, f_m \rangle = \{g_1(X)f_1(X) + \cdots + g_m(X)f_m(X) \mid g_i(X) \in \mathbf{Q}[X]\}$$

を考えることが有効である. 1 変数の場合とちがい, $I$ は一般には単項イデアルではないが, グレブナ基底を計算することにより, 解を求めやすい形に変形することができる.

$\mathbf{Q}[X]/I$ の $\mathbf{Q}$ 上のベクトル空間としての次元 $m$ が有限である場合, $I$ を 0 次元イデアル とよぶ. 根の多重度を適切に定義してやると $m$ は連立方程式系 (16.1) の複素解の個数に一致することが知られている ([2, 5 章 2 節 命題 3, 3 節 命題 8] に多重度のない場合の証明がある).

次の定理は 0 次元イデアルの定義とメンバシップアルゴリズムアルゴリズムを与えている定理 16.1 より容易に証明できる ([2, 5 章 3 節 定理 6] も参照).

**定理 16.3** 次は同値である.

1. $I$ は 0 次元イデアルである.
2. $I$ の, ある項順序に関する $I$ のグレブナ基底が, 各変数 $x_i$ に対して, initial term が $x_i$ のべきであるような元を含む.
3. $I$ の, 任意の項順序に関する $I$ のグレブナ基底が, 各変数 $x_i$ に対して, initial term が $x_i$ のべきであるような元を含む.

この定理より, 解が有限個かどうかの判定が (任意の項順序の) グレブナ基底により判定できる.

**例 16.2** $I = \langle x - yz, y - zx, z - xy \rangle$ の $x > y > z$ なる辞書式順序に関するグレブナ基底は $\{\underline{z^3} - z, \underline{-z^2y} + y, \underline{-y^2} + z^2, \underline{x} - zy\}$ である (冗長な要素は省いてある). よって定理より $I$ は 0 次元イデアルである.

**問題 16.2** 与えられたイデアルの 0 次元性を判定する関数を書け.

また, この定理を, 辞書式順序の場合に適用すると, 次を得る.

**定理 16.4** $I$ を 0 次元イデアルとし, $G$ を $x_1 > x_2 > \cdots > x_n$ なる辞書式順序に関する $I$ のグレブナ基底とすると, $G$ は

$$\begin{aligned}
g_1(x_1, x_2, \ldots, x_n) &= x_1^{d_1} + g_{1,d_1-1}(x_2, \ldots, x_n)x_1^{d_1-1} + \cdots + g_{1,0}(x_2, \ldots, x_n) \\
g_2(x_2, \ldots, x_n) &= x_2^{d_2} + g_{2,d_2-1}(x_3, \ldots, x_n)x_2^{d_2-1} + \cdots + g_{2,0}(x_3, \ldots, x_n) \\
&\cdots \\
g_{n-1}(x_{n-1}, x_n) &= x_{n-1}^{d_{n-1}} + g_{n-1,d_{n-1}-1}(x_n)x_{n-1}^{d_{n-1}-1} + \cdots + g_{n-1,0}(x_n) \\
g_n(x_n) &= x_n^{d_n} + g_{n,d_n-1}x_n^{d_n-1} + \cdots + g_{n,0}
\end{aligned}$$

という形の元を含む.

この定理により, まず 1 変数多項式 $g_n(x_n)$ の根を求め, それを $g_n(x_{n-1}, x_n)$ に代入すれば, $x_{n-1}$ に関する 1 変数多項式を得る. その根を求め $g_{n-2}(x_{n-2}, x_{n-1}, x_n)$ に代入して, と, 次々に 1 変数多項式の根を求めることで, もとの方程式系の解の候補を求めることができる. (本当の解かどうかは, 他のすべて多項式の零点になっているかどうかをチェックしなければならない. 根を近似で求めてあると, このチェックは容易ではない. )

**例 16.3** 例 16.2 のイデアル $I$ の零点を求めよう. まず, $z^3 - z = 0$ より $z = -1, 0, 1$.

1. $z = -1$

   $-y^2 + z^2 = 0$ より $y = -1, 1$. このとき $(x, y, z) = (-1, 1, -1), (1, -1, -1)$.

2. $z = 0$

   $-y^2 + z^2 = 0$ より $y = 0$. このとき $(x, y, z) = (0, 0, 0)$.

3. $z = 1$

   $-y^2 + z^2 = 0$ より $y = -1, 1$. このとき $(x, y, z) = (1, 1, 1), (-1, -1, 1)$.

**例題 16.2** $n$ 変数の多項式を $n$ 本ランダム生成し, それらで生成されるイデアルの辞書式順序グレブナ基底を求めて, どういう形をしているか観察せよ.

この例題の実験を行なう場合, $n$ をあまり大きくすると計算できない. せいぜい 4, 5 程度にする. また, 次数も 2, 3 次程度にしておくこと. 辞書式順序グレブナ基底は, いきなり gr などで計算しても大抵ムリなので, Asir マニュアルをよく読んで, 上手な計算方法を学ぶこと. 実際にやってみればわかるように, 辞書式順序グレブナ基底は大変特徴的な形をしている場合が多い:

$$\begin{aligned} g_1(x_1, x_n) &= x_1 - h_1(x_n) \\ g_2(x_2, x_n) &= x_2 - h_2(x_n) \\ &\cdots \\ g_{n-1}(x_{n-1}, x_n) &= x_{n-1} - h_{n-1}(x_n) \\ g_n(x_n) &= h_n(x_n) \end{aligned}$$

これを shape base と呼ぶ. この形の場合には, $g_n(x_n) = 0$ の根を求めれば, 他の変数の値は代入により求まる.

**例題 16.3** 与えられたイデアルが shape base を持つかどうか判定し, shape base を持つ場合に零点の近似値を計算する関数を書け.

1 変数多項式の根は, pari(roots,Poly) で計算できる. たとえば, pari(roots,x^3-1) で $x^3 - 1 = 0$ の (複素) 近似根を計算できる. ただし,

```
***   the PARI stack overflows !
 current stack size: 65536 (0.062 Mbytes)
 [hint] you can increase GP stack with allocatemem()
```

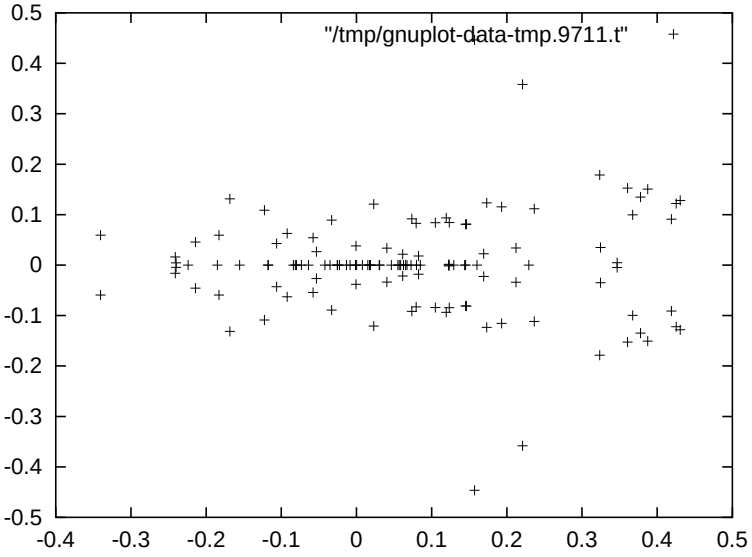
というようなエラーが出た場合には,

```
[295] pari(allocatemem,10^6)$
```

などを実行して, pari の使用できるメモリを増やすこと.

```
[826] load("katsura");
1
[831] katsura(7);
[u0+2*u7+2*u6+2*u5+2*u4+2*u3+2*u2+2*u1-1,
2*u6*u0+2*u1*u7-u6+2*u1*u5+2*u2*u4+u3^2,
2*u5*u0+2*u2*u7+2*u1*u6-u5+2*u1*u4+2*u2*u3,
2*u4*u0+2*u3*u7+2*u2*u6+2*u1*u5-u4+2*u1*u3+u2^2,
2*u3*u0+2*u4*u7+2*u3*u6+2*u2*u5+2*u1*u4-u3+2*u1*u2,
2*u2*u0+2*u5*u7+2*u4*u6+2*u3*u5+2*u2*u4+2*u1*u3-u2+u1^2,
2*u1*u0+2*u6*u7+2*u5*u6+2*u4*u5+2*u3*u4+2*u2*u3+2*u1*u2-u1,
u0^2-u0+2*u7^2+2*u6^2+2*u5^2+2*u4^2+2*u3^2+2*u2^2+2*u1^2]
```

Risa/Asir ドリル ギャラリー：$n = 7$ での katsura の方程式系. u0 から u7 が未知変数.




Risa/Asir ドリル ギャラリー：$n = 7$ での katsura の方程式系の解の第一成分 u0

# 関連図書

[1] 大阿久俊則: D-加群と計算代数, 朝倉書店.
微分方程式とグレブナ基底の入門書. グレブナ基底に関する証明付きの簡潔な解説もある.

[2] D.Cox, J.Little, D.O'Shea, *Ideals, Varieties, and Algorithms — An Introduction to Commutative Algebraic Geometry and Commutative Algebra*, 1991, Springer-Verlag.
日本語訳: D. コックス, J. リトル, D. オシー: グレブナ基底と代数多様体入門 (上/下). 落合他訳, シュプリンガー フェアラーク 東京, 2000. ISBN 4-431-70823-5, 4-431-70824-3.
世界的に広く読まれている, グレブナ基底の入門書. Buchberger アルゴリズム自体は, 2 章までよめば理解できる.

[3] 野呂: 計算代数入門, Rokko Lectures in Mathematics, 9, 2000. ISBN 4-907719-09-4.
`http://www.math.kobe-u.ac.jp/Asir/ca.pdf` から, PDF ファイルを取得できる.
`http://www.openxm.org` より openxm のソースコードをダウンロードすると, ディレクトリ `OpenXM/doc/compalg` にこの本の TeX ソースがある. グレブナ基底を用いた代数方程式系の解法や, Buchberger アルゴリズムの高速化法について詳しい解説がある.

[4] 野呂, 横山: グレブナー基底の計算 基礎篇 計算代数入門, 東京大学出版会, 2003. ISBN 4-13-061404-5.
グレブナー基底の基礎から代数方程式系の解法, イデアルの分解まで詳しく解説してある.

[5] 齊藤, 竹島, 平野: グレブナー基底の計算 実践篇 Risa/Asir で解く, 東京大学出版会, 2003. ISBN 4-13-061405-3.
一変数, 多変数の代数方程式 (系) のさまざまな解法を, Risa/Asir 上で実際に計算を行うためのプログラムを示しながら解説している.

# 第17章　RSA 暗号系

## 17.1　数学からの準備

RSA 暗号系は, 次の定理を基礎としている.

**定理 17.1** $G$ を位数 (要素の個数) が $n$ の群とするとき $G$ の任意の元 $a$ に対して $a^n = e$ である. ここで $e$ は単位元である.

群の定義については, 適当な数学の本を参照されたい.

この定理は可換とは限らない一般の群で成立するが, ここでは可換な場合の証明のみを紹介する. この証明の理解には群の定義を知ってるだけで十分である.

定理 17.1 の証明: 群 $G$ の $n$ 個の相異なる要素を $g_1, \ldots, g_n$ としよう. このとき, $\{ag_1, \ldots, ag_n\}$ を考えるとこれらもまた, $G$ の $n$ 個の相異なる元の集合となる. なぜなら, たとえば $ag_i = ag_j$ となると, $a$ の逆元を両辺にかけることにより, $g_i = g_j$ になり, 仮定に反するからである.

$\{g_1, \ldots, g_n\}$ と $\{ag_1, \ldots, ag_n\}$ は集合として等しいのであるから,

$$g_1 \cdots g_n = (ag_1) \cdots (ag_n) = a^n (g_1 \cdots g_n)$$

がなりたつ. 両辺に $g_1 \cdots g_n$ の逆元を掛けてやると, $e = a^n$ をえる. 証明おわり.

$p$ を素数としよう. とくにこの定理を, $\mathbf{Z}/p\mathbf{Z}$ の乗法群

$$G = \{1, 2, \ldots, p-1\}$$

に適用すると, 次の定理を得る.

**定理 17.2** $p$ を素数とするとき, $p$ で割れない任意の整数 $x$ について,

$$x^{p-1} = 1 \bmod p$$

となる.

もうすこしくわしくこの定理の説明をしよう. $a \bmod p$ で, $a$ を $p$ でわった余りをあらわすものとする. このとき

$$(a \bmod p)(b \bmod p) \bmod p = ab \bmod p$$

が成立する. 左辺は, $a$ を $p$ でわった余りと $b$ を $p$ でわった余りを掛けたあと, $p$ でわった余りをとることと, $ab$ を $p$ でわった余りをとることは同じである. という意味である. この事実および $p$ が素数のとき, 集合 $G = \{1, 2, \ldots, p-1\}$ の元 $a \in G$, $b \in G$ に対して, $ab \bmod p \in G$ でかけ算を定義することにより, $G$ は位数 $p-1$ の可換な群となることを用いると, 定理 17.2 の証明ができる. もちろん 1 がこの群の単位元である. $G$ が (可換な) 群であるを示すには, 逆元の存在が非自明である. 次の問題の 1 を示す必要がある.

**問題 17.1**　1. $p$ を素数とする. $a$ を 1 以上, $p-1$ 以下の数とするとき,

$$ab = 1 \bmod p$$

となる 1 以上, $p-1$ 以下の数 $b$ が存在する.

2. $a$, $p$ が互いに素な数なら, $ab = \bmod p$ となる数 $b$ が存在する.

3. $b$ を構成するアルゴリズムを考えよ. その計算量を考察せよ.

ヒント: $a$ と $p$ にユークリッドの互除法を適用せよ. Asir では, 関数 `inv` を $a$, $p$ より $b$ を求めるのに利用できる.

```
[346] for (I=1; I<5; I++) print(inv(I,5));
1
3
2
4
```

上の結果をみればわかるように, たしかに $1 \times 1 \bmod 5 = 1$, $2 \times 3 \bmod 5 = 1$, $3 \times 2 \bmod 5 = 1$, $4 \times 4 \bmod 5 = 1$ である.

## 17.2　RSA 暗号系の原理

$p$, $q$ を相異なる素数とし,

$$n = pq, \quad n' = (p-1)(q-1)$$

とおく. $e$ を

$$\gcd(e, n') = 1$$

となる適当な数とする.

$$de = 1 \bmod n'$$

となる数 $d$ をとる. このような $d$ が存在してかつ 互除法アルゴリズムで構成できることは, 問題 17.1 で考察した.

**定理 17.3** $m$ を $n$ 未満の数とする. このとき $c = m^e \bmod n$ とすると,

$$c^d = m \bmod n$$

が成り立つ.

証明: 定理 17.2 を $x = m^{q-1} \bmod p$ に対して適用すると,

$$(m^{q-1})^{p-1} = m^{n'} = 1 \bmod p$$

である. 同様の考察を素数 $q$ と $m^{p-1}$ に対しておこなうと,

$$(m^{p-1})^{q-1} = m^{n'} = 1 \bmod q$$

がわかる. $m^{n'}-1$ は $p$ でも $q$ でも割り切れかつ $p$ と $q$ は相異なる素数であるので $m^{n'}-1$ は $pq = n$ で割り切れる. よって, $m^{n'} = 1 \bmod n$ が成り立つ.

さて, 証明すべき式は, $(m^e)^d = m \bmod n$ であるが, 仮定よりある整数 $f$ が存在して $ed = 1 + fn'$ が成り立つことおよび $m^{n'} = 1 \bmod n$ を用いると, $(m^e)^d = m^{ed} = m^{1+fn'} = m(m^{n'})^f$ を $n$ で割った余りが $m$ であることがわかる. 証明おわり.

上のような条件をみたす数の組の例としては, たとえば

$$p = 47, q = 79, n = 3713, n' = 3588, e = 37, d = 97$$

がある. 最後の数, $d$ は `inv(37,3588);` で計算すればよい. したがって, 二つの素数 $p$, $q$ を用意すれば, 簡単に上のような条件をみたす数の組を作れる.

RSA 暗号系では, $p, q, d$ を秘密にし, $e, n$ を公開する. $(e, n)$ を公開鍵, $d$ を秘密鍵と呼ぶ. $m$ ($m$ は $n$ 未満の数) の暗号化は,

$$m^e \bmod n$$

でおこなう. この暗号化されたメッセージの復号化 (もとのメッセージにもどすこと) は, 秘密鍵 $d$ を利用して,

$$m^d \bmod n$$

でおこなう. この計算で正しくメッセージ $m$ が復号できることは, 定理 17.3 で証明した.

さて, これのどこが暗号なんだろうと思った人もいるかもしれない. $e, n$ が公開されているのなら, $n$ を素因数分解して, $p$, $q$ を求め, $\texttt{inv}(e, (p-1)*(q-1))$ をもとめれば, 秘密鍵 $d$ がわかってしまうではないか! ここで, 素因数分解は最大公約数 (GCD) の計算に比べて, コストのかかる計算だということを思い出してほしい. $p$, $q$ を十分大きい素数にとると, $pq$ の素因数分解分解の計算は非常に困難になる. したがって $p$, $q$ の秘密が保たれるのである.

参考: 量子計算機はこの素因数分解を高速にやってしまうということを Shor が示した. これが現在量子計算機がさかんに研究されている, ひとつのきっかけである.

## 17.3 プログラム

下のプログラムの `encrypt(M)` は文字列 $S$ を RSA 暗号化する. `decrypt(C)` は `encrypt` された結果を元の文字列に戻す. 例を示そう.

```
[356] encrypt("OpenXM");
Block_size = 2
The input message = OpenXM
20336
25966
22605
0

[4113338,3276482,4062967,0]
[357] decrypt(@@);
Block_size = 2
The input message to decrypt
= [4113338,3276482,4062967,0]
20336
25966
22605
0

[OpenXM,[79,112,101,110,88,77]]
```

文字列 “OpenXM” を `encrypt` で暗号化する. 結果は [4113338,3276482,4062967,0] である. これを入力として, `decrypt` を呼び出すと, 文字列 “OpenXM” を復元できる.

`encrypt` はあたえらた文字列をまず アスキーコードの列に変換し, それをブロックに分割してから, 各ブロック $m$ の $m^e \bmod n$ を計算して暗号化する. `20336, 25966, 22605` は各ブロックの $m$ の値である. なお下のプログラムの `PP` が $p$, `QQ` が $q$, `EE` が $e$, `DD` が $d$(秘密鍵) にそれぞれ対応する.

この実行例では, $p = 1231$, $q = 4567$, $e = 65537$, $d = 3988493$ を利用している.

以下の変数への値の設定プログラムと関数を集めたファイルが rsa.rr である.

```
PP=1231$
QQ=4567$
EE=65537$
DD=3988493$
/*
    PP = 1231, QQ=4567, N=PP*QQ,  N'=(PP-1)*(QQ-1)
    EE = 65537, (gcd(EE, N') = 1),
    DD = 3988493, ( DD*EE = 1 mod N').
 (These values are taken from the exposition on RSA at
  http://www8.big.or.jp/%7E000/CyberSyndrome/rsa/index.html)
 (EE,N) is the public key.
  DD is the private key.  PP, QQ, N' should be confidential
*/
```

```
def naive_encode(S,P,N) {
  /* returns S^P mod N */
  R = 1;
  for (I=0; I<P; I++) {
    R = (R*S) % N;
  }
  return(R);
}
```

```
def encode(X,A,N) {
  R = 1; P = X;
  while (A != 0) {
    if (A % 2) {
      R = R*P % N;
    }
    P = P*P % N;
    A = idiv(A,2);
  }
  return(R);
}
```

```
def encrypt(M) {
   extern EE,PP,QQ;
   E =  EE; N= PP*QQ;
   Block_size = deval(log(N))/deval(log(256));
   Block_size = pari(floor,Block_size);

   print("Block_size = ",0); print(Block_size);
   print("The input message = ",0); print(M);
   M = strtoascii(M);
   L = length(M);
   /* Padding by 0 */
   M = append(M,
        vtol(newvect((idiv(L,Block_size)+1)*Block_size-L)));
   L = length(M);

   C = [ ];  S=0;
   for (I=1; I<=L; I++) {
     S = S*256+M[I-1];
     if (I % Block_size == 0) {
       print(S);
       S = encode(S,E,N);
       C = append(C,[S]);
       S = 0;
     }
   }
   print(" ");
   return(C);
}
```

```
def decrypt(M) {
   extern DD, PP, QQ;
   D = DD; N = PP*QQ;
   Block_size = deval(log(N))/deval(log(256));
   Block_size = pari(floor,Block_size);

   print("Block_size = ",0); print(Block_size);
   print("The input message to decrypt = ",0); print(M);
   L = length(M);

   C = [ ];
   for (I=0; I<L; I++) {
     S = encode(M[I],D,N);
     print(S);
     C1 = [ ];
     for (J=0; J<Block_size; J++) {
       S0 = S % 256;
       S = idiv(S,256);
       if (S0 != 0) {
         C1 = append([S0],C1);
       }
     }
     C = append(C,C1);
   }
   print(" ");
   return([asciitostr(C),C]);
}
end$
```

encode(X,A,N) は, $\mathtt{X}^{\mathtt{A}} \bmod \mathtt{N}$ を計算する関数である. native_encode(X,A,N) は定義どおりにこの計算をする関数である. この関数をためしてみればわかるように, 工夫してこの計算をしないと大変な時間がかかる. A を 2 進展開して計算しているのが, encode(X,A,N) である. 実行時間を比べてみてほしい.

A を 2 進展開し

$$\sum a_i 2^i, \quad (a_i = 0 \text{ or } 1)$$

なる形にあらわすと,

$$\mathtt{X}^{\mathtt{A}} = \prod_{i\,:\,a_i \neq 0} \mathtt{A}^{2^i}$$

とかける. encode(X,A,N) では, A, $\mathtt{A}^2$, $\mathtt{A}^4$, ... を N でわった余りを順番に計算して変数 P にいれている. あとは, 2 進展開を利用して

$$\mathtt{X}^{\mathtt{A}} \bmod \mathtt{N} = \prod_{i\,:\,a_i \neq 0} \mathtt{A}^{2^i} \bmod \mathtt{N}$$

を計算している.

encrypt では, あたえらた文字列をまずアスキーコードに変換して, 変数 M にいれている. Block_size を $b$ とするとき, まず,

$$\mathtt{M}[0]256^{b-1} + \mathtt{M}[1]256^{b-2} + \cdots + \mathtt{M}[\mathtt{b}-1]256^0$$

を変数 S に代入し, この S に対して, $\mathtt{S}^{\mathtt{E}} \bmod \mathtt{N}$ を計算する. この操作を各ブロック毎に繰り返す.

decrypt は encrypt とほぼ同様の操作なので説明を省略する.

さて次の問題として, RSA 暗号化システムのための公開鍵 $(n, e)$ および秘密鍵 $d$ を生成する問題がある. 次のプログラム rsa-keygen.rr は, これらの数を生成し, 変数 EE, DD などに設定する.

```
def rsa_keygen(Seed) {
  extern PP,QQ,EE,DD;
  random(Seed);
  do {
    P = pari(nextprime,Seed);
    Seed = Seed+P;
    Q = pari(nextprime,Seed);
    PP = P;
    QQ = Q;
    Phi = (P-1)*(Q-1);
    E = 65537;
    Seed = Seed+(random()*Q % Seed);
  } while (igcd(E,Phi) != 1);
  EE = E;
  DD =inv(EE,Phi);
  print("Your public key (E,N) is ",0); print([EE,PP*QQ]);
  print("Your private key D is ",0); print(DD);
  return([PP,QQ,EE,DD]);
}
end$
```

次の例は, $2^{128} = 340282366920938463463374607431768211456$ 程度の大きさの素数を 2 個生成して, RSA の公開鍵, 秘密鍵 を作る例である. なお, この程度の大きさの素数の積は最新の理論とシステムを用いると容易に因数分解可能である.

```
[355] load("rsa.rr")$
[356] load("rsa-keygen.rr")$
[359] rsa_keygen(2^128);
Your public key (E,N) is [65537,
231584178474632390847141970017375815766769948276287236111932473531249232711409]
Your private key D is
19961886913057446009652405554498340187104891091301936388575383184168509927206 1

[340282366920938463463374607431768211507,
68056473384187692692674921486353642298 7,
```

```
65537,
19961886913057446009652405554498340187104891091301936388575383184168509927206 1]
[360] encrypt("Risa/Asir");
Block_size = 32
The input message = Risa/Asir
372759688465508844469111436918076915836368359054402083770354411365009352294 40

[146634940900113296504342777649966848592634201106623057430078652022991264082696]
[361] decrypt(@@);
Block_size = 32
The input message to decrypt =
[146634940900113296504342777649966848592634201106623057430078652022991264082696]
372759688465508844469111436918076915836368359054402083770354411365009352294 40

[Risa/Asir,[82,105,115,97,47,65,115,105,114]]
```

高速に安全な公開鍵 $(n, e)$ および秘密鍵 $d$ を生成する問題は, RSA 暗号ファミリを利用するうえでの一つの中心的問題である. たとえば, $p-1$, $q-1$ が小さい素数の積に分解する場合は, 比較的高速な $n$ の素因数分解法が知られている. つまりこのような $(p, q)$ から生成した鍵はこの素因数分解法の攻撃に対して脆弱である. 上の関数 `rsa_keygen` はこのような攻撃に対する脆弱性がないかのチェックをしていない. その他, さまざまな攻撃法に対する, 脆弱性がないかのチェックが必要となる.

Risa/Asir は, 楕円曲線の定義する可換群をもちいる暗号系である, 楕円暗号系に関して, 安全なこれらのパラメータを生成するシステムの基礎部分として実際に利用されている. Risa/Asir に組み込まれている, 大標数有限体の計算機能および高速な 1 変数多項式の計算機能はこれらに必要な機能として開発された.

**問題 17.2** 上のプログラムでは, 文章をブロックに分けてから, RSA 暗号化している. 1 byte づつ暗号化すると比較的容易に暗号が解読できる. その方法を考察せよ.

**問題 17.3** 公開鍵 $(e, n) = (66649, 24691358025875308641989 47)$ を用いて, 関数 `encrypt` で, ある文字列を変換したら,
`[534331413430079382527551,48621867143313553552184 0]`
を得た. どのような文字列だったか?

# 第18章　構文解析

## 18.1　再帰降下法

第 12 章 の 12.2 節において, 後置記法 (逆ポーランド記法) で書いた数式を計算するプログラムを解説した. このプログラムはスタックの利用により後置記法の式の計算を実現している. この節では, 中置記法の数式を後置記法へ変換するプログラムを開発する. 中置記法は演算子を真中におく, おなじみの式の記述方法である. たとえば

$$2 + 3 * (45 + 2)$$

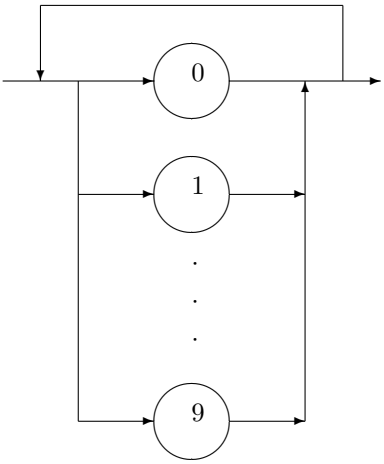
は中置記法の式であり, これを後置記法に変換すると, たとえば　`2 3 45 2 + * +`　となる.

変換するプログラムを書く前にどのような “文法” の中置記法の式を入力として認めるのかきちんと定義する必要がある. 計算機の世界で意味する文法というのは, 英文法の意味する文法とは違い, 曖昧さが入る余地はゆるしていないし, 人間が人工的に定義する.

文法は BNF (Bakus-Nauer) 形式や構文図を用いて定義する. われわれの, “式” の構文を構文図で定義してみよう.

number は次の構文図で定義する.

number :

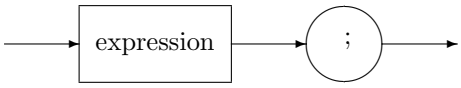



左端からスタートして右端への矢印へたどりつければ number である. たとえば, `123` は number であるが, `1.23` は number でない. なぜなら `1` は構文図がうけつけてくれるが, 次にくる `.` は構文図がうけつけてくれない.

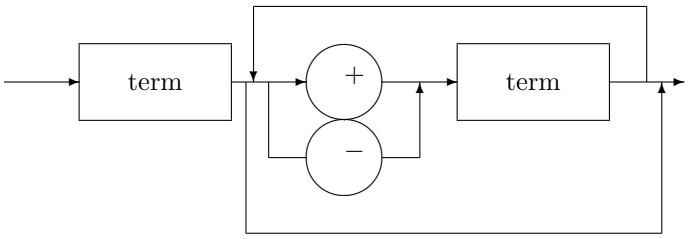
variable は, `A` から `Z`, `a` から `z` のうちの一文字である. 構文図は省略する.

sentense は, 次の構文図で定義する.

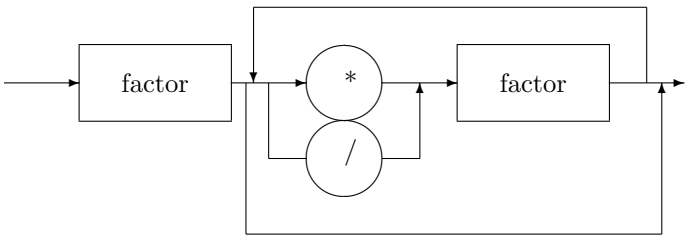
sentense :



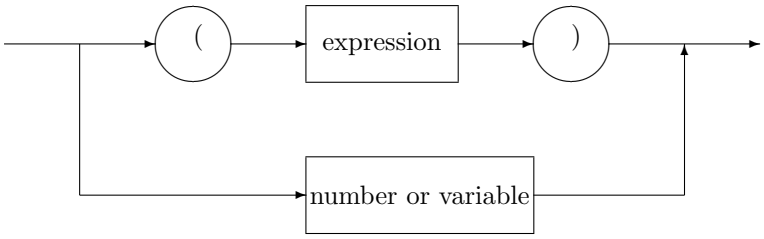

expression は, 次の構文図で定義する.
expression :




term は, 次の構文図で定義する.
term :




factor は次の構文図で定義する.




構文図にでてくる ▭ でかこまれた記号を非終端記号 とよぶ. これらは, 別の構文図で定義されている. ○で囲まれた記号を終端記号という.

## 18.2　プログラム minicomp.rr

前の節で定義した文法を満たす式を後置記法に直すプログラムは次のようになる.

`#define` 文で定義している, `CodeOf` で始まるシンボルは, アスキーコードをプログラム中に直接書くと読みにくいので使用している. たとえば `CodeOfA` は `A` のアスキーコード `0x41` に対応

している.

大域変数の解説をしよう.

| | |
|---|---|
| Ch | 最後に読み込んだ文字. |
| Sy | シンボルのタイプ. VARIABLE が変数, NUMBER が数字, その他が特殊記号. |
| Val | シンボルに対する値. NUMBER の時は実際の数値. |
| Ptr | 入力文字列 Str の何文字目を読むか? |
| Result | 後置記法に変換した結果の文字列. |

関数名と非終端記号はつぎのように対応している.

| | |
|---|---|
| expr() | 最上位レベルの式 (expression) |
| exprterm() | 項 (term) |
| exprfactor() | 因子 (factor) |

```
#define SPACE 0x20
#define CodeOfA 0x41
#define CodeOfZ 0x5a
#define CodeOfa 0x61
#define CodeOfz 0x7a
#define CodeOf0 0x30
#define CodeOf9 0x39
#define CodeOfPlus  0x2b
#define CodeOfMinus 0x2d
#define CodeOfMult  0x2a
#define CodeOfDiv   0x2f
#define CodeOfLeftBracket  0x28
#define CodeOfRightBracket 0x29

#define VARIABLE 1
#define NUMBER   2

Ch = 0x20$ Sy = 0$
Value =0$ Ptr = -1$
Result=""$
```

次の関数 parse() に変換したい文字列を引数として与えると, 後置記法に直す. たとえば

```
parse("2+3*(45+2);");
```

と入力すればよい. ; を忘れると parse() が修了しないので注意.

```
def parse(S) {
  extern Ch,Sy,Value,Ptr,Str, Result;
  Str = strtoascii(S);
  Ch = 0x20; Sy = 0; Value = 0; Ptr = -1;
  Result = "";
  in_symbol();
  expression();
  Result += " = ";
  return Result;
}
```

次は補助関数群である.

```
def is_space(C) {
  if (C <= SPACE) return 1;
  else return 0;
}

def is_alpha(C) {
  if ((C >= CodeOfA) && (C <= CodeOfZ)) return 1;
  if ((C >= CodeOfa) && (C <= CodeOfz)) return 1;
  return 0;
}

def is_digit(C) {
  if ((C >= CodeOf0) && (C <= CodeOf9)) return 1;
  return 0;
}

def getch() {
  extern Ptr, Str;
  if (Ptr >= length(Str)-1) return -1;
  else {
    Ptr++;
    return Str[Ptr];
  }
}
```

構文解析と変換は Str より文字を順番に読むことにより, おこなわれる. このとき文字列のなかの不必要な空白を読み飛ばしたり, 数字文字列を数字に変換したり, 数字と演算子を切り分けたりする必要がある. このような作業をトークンへの分解 とよぶ. この作業を担当しているのが, in_symbol() である.

```
def in_symbol() {
  extern Ch;
  while (is_space(Ch)) {
    Ch = getch();
  }
  if (is_alpha(Ch)) {
    Sy = VARIABLE;
    Value = Ch;
    Ch = getch();
  } else if (is_digit(Ch)) {
    Sy = NUMBER;
    Value = 0;
    do {
      Value = Value*10+(Ch-CodeOf0);
      Ch = getch();
    }while (is_digit(Ch));
  } else {
    Sy = Ch;
    Ch = getch();
  }
}
```

構文図に対応する関数達. 再帰的に呼ばれている.

```
def expression() {
  expr();
}
```

```
def expr() {
  extern Result;
  exprterm();
  while ((Sy == CodeOfPlus) ||
         (Sy == CodeOfMinus)) {
    Ope = (Sy == CodeOfPlus? CodeOfPlus :
                             CodeOfMinus);
    in_symbol();
    exprterm();
    Result += asciitostr([Ope])+" ";
  }
}
```

```
def exprterm() {
  extern Result;
  exprfactor();
  while ((Sy == CodeOfMult) ||
         (Sy == CodeOfDiv)) {
    Ope = (Sy == CodeOfMult? CodeOfMult :
                             CodeOfDiv);
    in_symbol();
    exprfactor();
    Result += asciitostr([Ope])+" ";
  }
}
```

```
def exprfactor() {
  extern Result;
  if (Sy == NUMBER) {
    Result += rtostr(Value)+" ";
    in_symbol();
  }else if (Sy == CodeOfLeftBracket) {
    in_symbol();
    expr();
    if (Sy != CodeOfRightBracket) {
       error("Mismatched parenthesis");
    }
    in_symbol();
  }else{
    error("Invalid factor");
  }
}
```

これらの関数をまとめて, `minicomp.rr` なる名前にまとめて入力, ロード, 実行すると関数 `parse()` で中置記法の後置記法への変換を実現できる. これと 12 章の関数 `casio()` を組み合わせれば, 簡単な数式の処理システムが書ける.

## 18.3　LR パーサ

前の節で説明した構文解析の方法を再帰降下法とよぶ. この方法は終端記号, 非終端記号に対応した関数を用意したら, あとは構文図に従いこれらの関数を呼び出していけばよい.

この方法より効率のよい構文解析の方法が LR パーサである. この話題は上級の話題なので再帰降下で十分という方は読み飛ばされたい.

さて, この節ではこの構文解析法の原理を簡略に説明する. 詳しくは, Aho, Ullman の本 コンパイラを御覧いただきたい [1]. LR パーサではつぎのように構文解析する.

1. 文法 (構文図) から, オートマトンを生成する.
2. 入力をそのオートマトンで解析する.

SLR (simple LR) 構文解析法の説明を例を用いて説明する. 次の文法（生成規則）に対する SLR パーサを作ろう. この文法は [1] の 例 6.1 でとりあげられている文法である.

$$
\begin{aligned}
&(1)\ E \rightarrow E + F \\
&(2)\ E \rightarrow F \\
&(3)\ F \rightarrow F * T \\
&(4)\ F \rightarrow T \\
&(5)\ T \rightarrow (E) \\
&(6)\ T \rightarrow id\ \ (identifier)
\end{aligned}
$$

これは

$$
\begin{aligned}
E &\rightarrow E + F \mid F \\
F &\rightarrow F * T \mid T \\
T &\rightarrow (E) \mid id
\end{aligned}
$$

と書いてもよい.

E は expression, F は factor, T は term の略である. id (identifier) は数である. したがって, $2 + 3 * 5$ とか $(2 + 3) * 5$ とかはこの文法をみたしている. なお $+2$ は満たしていないことを注意しておく.

SLR 法では文法より次のような構文解析表をつくってそれを用いたオートマトンで構文解析をする.

| 状態 | id | + | * | ( | ) | ; | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | s5 | | | s4 | | | s1 | s2 | s3 |
| 1 | | s6 | | | | OK | | | |
| 2 | | r2 | s7 | | r2 | r2 | | | |
| 3 | | r4 | f4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | s8 | s2 | s3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | s9 | s3 |
| 7 | s5 | | | s4 | | | | | s10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | s7 | | r1 | r1 | | | |
| 10 | | r1 | s7 | | r1 | r1 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

この構文解析表をもちいてどの様に構文解析をやるか説明しよう. エラーの処理と OK(受理) の処理は省略してある.

```
driver:=
00: push(0); (* 状態 0 をスタックへ積んで置く.*)
01: c = ipop();
02: state = peek();
03: action = actionTable[state,c];
04: If action == 移動 (s) then
05:       push(c);
06:       push(nextStateTable[state,c]);
07: else (* 還元する. *)
08:       rule = nextStateTable(state,c);
09:       rule 分 pop() する;
10:       ipush(c);
11:       ipush( rule の左辺 );
12: endif
14: Goto[1];
```

各関数の働きはいかのとおり.

1. `stack` – スタック.
2. `push(c)` – スタック (stack) に `c` を積む.
3. `pop()` – スタックからデータをえる.
4. `peek()` – スタックのデータを得る, がスタックからとりだしはしない.
5. `inputStack` – 入力データが入っているスタック.
6. `ipush(c)` – `c` を入力スタック (inputStack) へ積む.
7. `nextStateTable` – 構文解析表. 移動のときは 1 から 11 を戻す. 還元のときは文法規則の番号を戻す.
8. `actionTable` – 構文解析表. 移動 (s), 還元 (r), 受理 (OK), 駄目のどれを戻す.

`03:` を終了した時点での各変数の様子を次の入力データ `2+3*5;` に対して見てみよう.

| c | state | action | stack, inputStack |
|---|---|---|---|
| 2 | $I_0$ | s | { $I_0$ } , { +,3,*,5,; } |
| + | $I_5$ | r | { $I_0, id, I_5$ }, { 3,*,5,; } |
| F | $I_0$ | s | { $I_0$ }, { +,3,*,4,; } |

次にどの様にして文法より構文解析表を作るのか説明しよう. 状態の集合, nextStateTable および Following[非終端記号] の三つのデータを計算すれば構文解析表ができる.

まず, 状態の集合の計算法を示そう. 生成規則の右辺に · を含む生成規則を項 (item) とよぶ. たとえば生成規則

$$E \to E + F$$

からは

$$E \to \cdot E + F,\ E \to E \cdot + F$$

$$E \to E + \cdot F,\ E \to E + F \cdot$$

の四つの項ができる. 直感的には生成規則のどこまでをすでに読み込んだのかを項は示している.

$I$ を項とする. `Closure(I)` とは $I$ を含む項の集合であり, 次の性質を満たすものとする.

$$A \to \alpha \cdot B\beta \in \texttt{Closere(I)}$$

かつ $B \to \gamma$ なる生成規則があれば,

$$B \to \cdot\gamma \in \texttt{Closure(I)}$$

である.

たとえば $E' \to \cdot E$ の Closure を計算すると

$$\begin{aligned} E' \to \cdot E & \\ E &\to \cdot E + F \\ E &\to \cdot F \\ F &\to \cdot F * T \\ F &\to \cdot T \\ T &\to \cdot(E) \\ T &\to \cdot id \end{aligned}$$

となる. これが上の構文解析表の状態 0 ($I_0$) に対応する. 次に id が読まれたとすると状態は

$$T \to id \cdot$$

となるはずである. · の後ろになにもないのでこの状態の Closure はこのままである. この状態の集合は 5 ($I_5$) に対応する. この状態では `Follow(F)` に属する終端記号が次に待っていれば $F \to id$ の還元をおこなう. 以下, 省略.

状態 $I_k$ が項

$$A \to \alpha \cdot \beta\gamma$$

を含むとする. `nextStateTable[` $I_k$`,` $\beta$`]` は

$$\texttt{Closure}(A \to \alpha\beta \cdot \gamma)$$

である.

例として, `nextStateTable[` $I_0$`, ( ]` を計算してみよう.

$$T \to \cdot(E)\ \in I_0$$

だから

$$T \to (\cdot E)$$

の Closure が求めるものである. Closure は

$$\begin{aligned} T &\to (\cdot E) \\ E &\to \cdot E + F \\ E &\to \cdot F \\ F &\to \cdot F * T \\ F &\to \cdot T \\ T &\to \cdot(E) \\ T &\to \cdot id \end{aligned}$$

である. この状態を $I_4$ となずける.

状態 $I_k$ が

$$A \to \alpha\cdot$$

を含むとする. 次の入力記号が `Follow(A)` ならルール

$$A \to \alpha$$

で還元 (r) する.

状態 $I_k$ が

$$E' \to E\cdot$$

を含むとする. 次の入力記号が ; (終わりマーク) ならば受理 (OK).

**問題 18.1**　次の生成規則に対する構文解析表を作れ.

$$\begin{aligned} E &\to E + F \mid F \\ F &\to (E) \mid id \end{aligned}$$

# 関連図書

[1] A.V.Aho and J.D.Ullman, *Principles of Compiler Design*, 1977, Addison-Wesley Publishing.
日本語訳は
A.V.Aho and J.D.Ullman, コンパイラ, 情報処理シリーズ 7, 培風館

# 第19章　OpenXMと分散計算

Risa/Asir は外部モジュールを比較的容易に組み込むことが可能である. Asir と外部モジュールの通信は OpenXM-RFC 100 プロトコルにしたがいおこなう. OpenXM については “野呂, 下山, 竹島 : Asir User's Manual” (cf. 26.3 節) の第 7 章を参照してほしい. 外部モジュール (OpenXM server) として Asir 自体を呼ぶことも可能であり, 分散並列アルゴリズムを容易に試すことが可能である.

## 19.1　OpenXM Asir server

関数 ox_launch を引数なしで実行すると, OpenXM Asir server (ox_asir) が起動する. 同時に, メッセージ表示用ウィンドウが現れる.

```
[0] Id0 = ox_launch();
0
```

戻り値 0 が, ox_asir を識別する番号で, その後の通信で必要となる. ox_asir は複数立ち上げることも可能である.

```
[1] Id1 = ox_launch();
1
[2] Id2 = ox_launch();
2
```

ox_asir を終了させるには, 識別番号を引数にして ox_shutdown() を実行すればよい.

```
[3] ox_shutdown(Id1);
0
```

対応するメッセージウィンドウが消えるはずである. あるいは, Asir 自体を終了させれば, 全ての ox_asir は自動的に終了する. ox_asir の機能自体は通常の Asir と全く同様であるが,

- データ, 命令を, 通信相手 (この場合は ox_asir を起動した Asir) から読む.
- 計算結果はスタックと呼ばれる ox_asir の内部配列に一旦格納し, 通信相手からの要求があったときに通信路を通して返す.

という点が異なる. ox_asir に対する関数実行は, ox_rpc() で行う.

```
[10] ox_rpc(Id0,"fctr",x^10-y^10);
0
[11]
```

即ち, Asir における xxx(Arg0,Arg1,...) を ox_rpc(Id,"xxx",Arg0,Arg1,...) とすればよい. ファイルのロードも ox_rpc(Id,"load","filename") でできる. 即ち, キーボードから入力したい

内容をファイルに書いて, それをロードさせれば, 通常の Asir と何らかわらないことになる. ox_rpc() は, ox_asir での実行終了を待たないですぐに復帰する. 結果の受け取りは ox_pop_local() で行う.

```
[12] ox_pop_local(Id0);
[[1,1],[x^4+y^2*x^3+y^4*x^2+y^6*x+y^8,1],[x^4-y^2*x^3+y^4*x^2-y^6*x+y^8,1],
[x-y^2,1],[x+y^2,1]]
[13]
```

この時点で, ox_asir での計算が終っていない場合には, 計算が終るまで ox_pop_local() は復帰しない. このような仕組みは, 一見不便そうに見えるかもしれないが, 複数の server を使って分散計算を行う場合に, 一度に複数の server で計算をスタートさせたり, 同時に自分も他の計算を始めたい場合などに必須である. その結果がなければ先の計算ができなくなった時点で server に結果をもらいにいくようにすれば, 待たされても何も困ることはないのである. サーバが計算結果を送信可能状態かどうかは関数 ox_select により調べる.
さて, 最低限以上のことを知っていれば, 分散計算を試す準備は OK である. 以下では, 2 つのアルゴリズムを OpenXM で並列計算してみよう.

## 19.2　Quick Sort

第 14 章でみたように, quick sort は, アルゴリズムのおしまいの部分で, 分割された二つの配列を引数として自分自身を呼び出すという再帰的アルゴリズムとなっている. この部分で, 自分で計算する代わりに, server を起動して計算を任せる, という (とても安直な) 分散計算を試してみよう.

注意 : ちょっと考えれば, 「親」は, 配列を二つに分けることしかしておらず, 肝腎の整列は「子」に全部任せて待つだけということが分かる. 通信はそれなりにコストがかかるし, server の立ち上げはとてもコストがかかるので, こんなことをしても効率向上は望めないが, こんなことが簡単に実行できるという例として紹介している.

```
#define LevelMax 2
Proc1 = -1$
Proc2 = -1$
def quickSort(A,P,Q,Level) {
   extern Proc1, Proc2;
   if (Q-P < 1) return A;
   print("Level=",0); print(Level);
   Mp = idiv(P+Q,2);
   M = A[Mp];
   B = P; E = Q;
   while (1) {
     while (A[B] < M) B++;
     while (A[E] > M && B <= E) E--;
     if (B >= E) break;
     else {
       Tmp = A[B];
```

```
        A[B] = A[E];
        A[E] = Tmp;
        E--;
      }
  }
  if (E < P) E = P;
/* -------------------------- */
  if (Level < LevelMax) {
    if (Proc1 == -1) {
      Proc1 = ox_launch();
    }
    if (Proc2 == -1) {
      Proc2 = ox_launch();
    }
    ox_rpc(Proc1,"quickSort",A,P,E,Level+1);
    ox_rpc(Proc2,"quickSort",A,E+1,Q,Level+1);
    if (E-P < Q-E) {
       A1 = ox_pop_local(Proc1);
       A2 = ox_pop_local(Proc2);
    }else{
       A2 = ox_pop_local(Proc2);
       A1 = ox_pop_local(Proc1);
    }
    for (I=P; I<=E; I++) {
       A[I] = A1[I];
    }
    for (I=E+1; I<=Q; I++) {
       A[I] = A2[I];
    }
    return(A);
/* -------------------------- */
  }else{
    quickSort(A,P,E,Level+1);
    quickSort(A,E+1,Q,Level+1);
    return(A);
  }
}
end$
```

このプログラムは第 14 章のものとほとんど同じで, 異なるのは `/* -------------------------- */` ではさまれた部分のみである. このプログラムでは, `ox_launch()` により動的に `ox_asir` を起動しているので, 起動時にこの関数が定義されるように, `.asirrc` に書いておく必要がある. たとえば, この関数が含まれるファイルを`/home/noro/asir/d_qsort` とすれば, `.asirrc` に

```
load("/home/noro/asir/d_qsort")$
```

を追加しておけばよい. この関数では, 際限なく server が生成されるのを防ぐために, 再帰の段数が LevelMax という定数を越えたら自プロセス内で再帰するようにしてある. また, 一旦 server が生成されたらその識別番号を覚えておき, 無駄に server を生成しないようにもしてある. (いくら効率無視でもこのような工夫は必要. 特に段数の制限は重要. さもないと計算機自体がストップしてしまうこともあり得る. )

## 19.3　Cantor-Zassenhaus アルゴリズム

有限体上の一変数多項式の因数分解法はさまざまな方法が知られているが, ここでは, 次数が等しい相異なる既約多項式の積 $f$ を因数分解する方法である Cantor-Zassenhaus アルゴリズムを紹介する. 扱う入力が限られているように見えるが,

1. 一般の多項式は, GCD により, 重複因子を含まない多項式 (無平方多項式) の積に分解できる.
2. 無平方多項式は, GCD により, 次数が等しい相異なる既約多項式の積に分解できる. 各因子に含まれる既約多項式の次数も知ることができる.

により, 実用的にも意味がある. ここでは, 係数体を, 標数 $p$ が奇数である $q$ 元体 $(q = p^n)$ とする. このとき, 次が成り立つ.

**定理 19.1** $f$ が $d$ 次既約多項式 $f_1, f_2$ の積とする. $GF(q)$ 上の高々 $2d-1$ 次式 $g$ をランダムに選ぶとき $\mathrm{GCD}(g^{(q^d-1)/2}-1, f)$ が $f_1$ または $f_2$ となる確率は $1/2 - 1/(2q)^d$.

即ち, $f$ に含まれる二つの $d$ 次既約因子が, ランダムに選んだ $2d-1$ 次多項式一つにより, 確率 $1/2$ 弱で分離できることが分かる. $f$ に含まれる因子の数が 2 より大きいでも, ある (小さくない) 確率で $f$ が二つの自明でない因子に分離できる.

これを用いて, quick sort と同様に, 入力多項式を既約多項式に分解する再帰的なアルゴリズム (Cantor-Zassenhaus アルゴリズム) が得られる. このアルゴリズムも quick sort と同様に並列化できる.

```
#define LevelMax 5

extern Proc1$
Proc1 = -1$

/* random poynomial generator */

def monic_randpoly_ff(N,V)
{
  for ( I = 0, N1 = N-1, S = 0; I < N1; I++ )
    S += random_ff()*V^I;
  return V^N1+S;
}

/* a wrapper of c_z() called on a server */
```

```
def ox_c_z(F,E,M,Level)
{
  setmod_ff(M);
  F = simp_ff(F);
  L = c_z(F,E,Level);
  return map(lmptop,L);
}

/*
  Input : a square free polynomial F s.t.
          all the irreducible factors of F has the degree E.
  Output: a list containing all the irreducible factors of F
*/

def c_z(F,E,Level)
{
  V = var(F);
  N = deg(F,V);
  if ( N == E )
    return [F];
  M = field_order_ff();
  K = idiv(N,E);
  L = [F];
  while ( 1 ) {
    W = monic_randpoly_ff(2*E,V);
    T = generic_pwrmod_ff(W,F,idiv(M^E-1,2));
    W = T-1;
    if ( !W )
      continue;
    G = ugcd(F,W);
    if ( deg(G,V) && deg(G,V) < N ) {
      if ( Level >= LevelMax ) {
        L1 = c_z(G,E,Level+1);
        L2 = c_z(sdiv(F,G),E,Level+1);
      } else {
        if ( Proc1 < 0 )
          Proc1 = ox_launch();
        ox_cmo_rpc(Proc1,"ox_c_z",lmptop(G),E,setmod_ff(),Level+1);
        L2 = c_z(sdiv(F,G),E,Level+1);
        L1 = ox_pop_cmo(Proc1);
        L1 = map(simp_ff,L1);
      }
      return append(L1,L2);
    }
```

```
  }
}
end$
```

このプログラムでは, server の有効活用のため, 二つ生成される因子の分解のうち, 一方のみを新しい server に送り, 残りは自分が分解する.

**問題 19.1** PC-UNIX (FreeBSD, Linux など) では, 複数の CPU を持つマシン上で, 複数のプロセスが実際に異なる CPU 上で動作するように設定することができる. このような場合に, 上のプログラムでどの程度速度が向上するか試してみよ. 入力としては, 異なる一変数多項式の積を用いればよい.

**問題 19.2** 上のプログラムは, 元のプロセスが動作しているマシンの上に server を立ち上げてしまう. 複数台のマシンが使える人は, “野呂, 下山, 竹島 : Asir User’s Manual” (cf. 26.3 節) の第 7 章をよく読んで, 上のプログラムを複数のマシン上の server を使えるように改良せよ. さらに, 計算時間と server 数の比 (台数効果) を調べよ.

余談 (by T): 複数台のマシンを実際に接続して計算しようという場合, 如何に安全に他のマシンでサーバを起動するかという問題がある. 伝統的に組織全体のファイアウオールを嫌い, cracking の嵐が直接研究室に襲ってくる大学環境の場合, これは常に切実になやまされる問題である. 分散計算実験機全体をお手製ファイアウオールにいれてしまったり, PAM を用いるのが正当的だろうが, もっとおてがるな方法に, `rsh` を `ssh` に symbolic リンクしてしまうという方法がある. `ssh-agent` の利用で, 安全にパスフレーズなしでリモートサーバを起動できるし, `rsh` のお手軽さはそのままである. もちろん余計なサービスは全て停止しておくのが常識である.

# 第20章　Asir 用のライブラリの書き方

## 20.1　ライブラリの書き方

あるプログラムをグループ内で共有したり, user contributed package として Asir あるいは OpenXM distribution の一部として配布したりする場合, そのようなプログラムはしばしばライブラリと呼ばれる. 本節では, プログラムをライブラリとして書く場合に注意すべき点について述べる.

OpenXM プロジェクトでは, Asir Contrib という, Asir のライブラリを整備するプロジェクトを進めている. この Asir Contrib におけるルールも解説する.

### 20.1.1　関数名の衝突の回避

Asir においては, 組み込み関数と同名の関数を定義することはできない.

```
[101] def fctr(X) { print("afo");}
def : builtin function fctr() cannot be redefined.
```

定義しようとする関数が, 組み込み関数としてもう存在しているかどうかは, 関数 `flist()` の出力を見ればわかるが, とりあえずダミーで定義してみるのが最も簡単だろう. (`flist()` の出力をみるのは大変である.) とは言え, 実際に困るのは, 既にあるライブラリ中のユーザ関数名との衝突である. あるセッションで, 既に定義されているユーザ関数との衝突は, `ctrl("verbose",1)` により見ることができる.

```
[183] def afo(X) { print("afo");}
[184] def afo(X) { print("bfo");}
Warning : afo() redefined.
[185]
```

しかし, この方法は読み込んであるファイルにしか対応できないので, 結局のところ, 関数名に固有の prefix をつけるのが現実的であろう. 例えば, 関数名は, `noro_` のように, 書いた本人を識別できるような文字列で始まるようにする. `poly_` や `matrix_` のように数学的に意味がある prefix を使うのは自然なように見えるが, これらは「当たる」危険性が高いので避けたほうがよい. 以上のような方法の他にモジュール機能を用いて自分の書いたライブラリをカプセル化してしまう方法がある. これについては, 次の節で解説する.

これに関連して, Asir Contrib プロジェクトにおいて 種々のシステムを混用する際の関数名の混乱を避けるために, 著者 (T) が OpenXM 版のソースにおいて `OpenXM/src/asir-contrib/packages/src/names.rr` なるテンプレートを準備している. そこでは, 共通関数名仕様に基いて, それを実現する Asir プログラムが書かれている. OpenXM 版の Risa/Asir (OpenXM/Risa/Asir ) では asir contrib ライブラリを起動時に自動的に読み込む. `names.rr` に定義されている関数が, OpenXM/Risa/Asir の asir

contrib のトップレベルの関数である. たとえば以下のようなきまりで, 関数名がついている.

| | |
|---|---|
| base_ で始まる関数 | 置き換えなど基本的な種々の演算 |
| matrix_ で始まる関数 | 行列に関する種々の演算 |
| print_ で始まる関数 | TeX への変換などの演算 |

このファイルで定義されている関数名は, 数学的に自然なものが使われている. Asir contrib の各関数の本体は taka_print_tex_form などとプログラムを書いた人の識別子 (この場合は taka) からはじまる名前がついてる. Asir contrib のトップレベルの関数からはこのような名前の関数が呼ばれている. たとえば print_tex_form は, taka_print_tex_form を現在は呼んでいる. 他によりよい実装がでたら, そちらを呼ぶようにすればいいだけでトップレベルの関数名 print_tex_form はかわらない.

### 20.1.2　変数名の衝突

Asir プログラムでは, 関数内で宣言なしにあらわれた変数は全て局所変数なので, どのような変数名を使っても問題はないが, 大域変数 (extern 宣言された変数) は要注意である. トップレベルでは比較的短い名前 (1 文字など) を使うことが多い. そのため, ファイル中で短い名前の変数を extern 宣言すると, 計算途中その変数に思わぬ値が代入されてしまう場合がある. ファイル中で初期データなどを入力する場合, 関数の外で行われた変数への代入は, 全て大域変数への代入となる. 即ち, 「トップレベル」というのが必ずしも手で入力しているという意味ではないことに注意.

Asir Contrib では, たとえば Taka_print_flag のように, パッケージを書いた人の識別子からはじまる, 大域変数名をもちいるのを推奨している.

## 20.2　モジュール機能

ライブラリで利用されている関数, 変数をカプセル化する仕組みがモジュール (module) である. 大規模なライブラリを作成する場合は, いままでの節に述べたような名前の問題を回避するために, module を用いるとよい.

まず module を用いたプログラムの例をあげよう. これは 12 章で詳しく説明したスタックを実現しているプログラムである.

```
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. 大域変数の初期化は宣言に続いてたとえば,

```
static Sp $
Sp = 0$
```

のようにおこなう. なお上の例にはあらわれないが, モジュールの外の大域変数は extern で宣言する.

次に来るのが localf を用いたモジュール内の関数の宣言である. ここでは, push と pop を宣言している. この宣言は必須である.

(大域) 関数 demo では, モジュール stack の中の関数 push と pop を

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

なる形式で呼び出している. つまり

モジュール名. 関数名 (引数 1, 引数 2, ... )

の形式でモジュール内の関数をモジュールの外から呼び出せる. さて,

```
if (Sp >= Ssize) {print("Warning: Stack overflow\nDiscard the top"); pop();}
```

ではモジュールの内の関数 pop を呼び出している. これをみればわかるように, モジュールの内部の関数を呼ぶには普通の形式で呼び出せばよい. 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.

なお asir では局所変数の宣言は不要であった. しかし上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.

```
module stack;
localf push $
localf pop $
endmodule;    /* ここまでがプロトタイプ宣言 */

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

module stack;
  stack の定義実体
endmodule;
```

宣言がないと, たとえば demo の実行時に push が undefined function エラーとなり停止する.

## 20.3　マニュアルの書き方 その 1 : texinfo で直接書く

“野呂, 下山, 竹島 : Asir User's Manual” は texinfo というスタイルで書かれている (cf. 26.3 節). これは LaTeX のようなマクロパッケージの一つで, GNU tool のマニュアルの記述用に伝統的に用いられている. 一つのソースファイルから, 次の 3 つのマニュアルが生成できるので便利である.

- TEX で整形されたマニュアル

  TEX で typeset する.

- GNU info 用ハイパーテキスト形式マニュアル

  makeinfo で変換する.

- HTML マニュアル

  texi2html で変換する.

本節では, Risa/Asir 関連のマニュアルを書く際の標準的なスタイルについて説明する. もちろん, そのスタイルに従う必要はないが, 従えばスタイルを考える必要がないぶん楽ができるのでおすすめである.

まず, 例を示す. この例は OpenXM 版 Asir のソース配布 (付録 26 章) の OpenXM/src/asir-doc に man-sample.texi としておいてあるのでこれを適当に改変することで, 穴埋め的にマニュアルが書ける.

```
@comment --- おまじない ---
\input jtexinfo
@iftex
@catcode`@#=6
@def@fref#1{@xrefX[#1,,@code{#1},,,]}
@def@b#1{{@bf@gt #1}}
@catcode`@#=@other
@end iftex
@overfullrule=0pt
@c -*-texinfo-*-
@comment %**start of header
@comment --- おまじない終り ---

@comment --- GNU info ファイルの名前 ---
@setfilename xyzman

@comment --- タイトル ---
@settitle XYZ

@comment %**end of header
@comment %@setchapternewpage odd

@comment --- おまじない ---
@ifinfo
@macro fref{name}
@ref{\name\,,@code{\name\}}
@end macro
@end ifinfo
```

```
@iftex
@comment @finalout
@end iftex

@titlepage
@comment --- おまじない終り ---

@comment --- タイトル，バージョン，著者名，著作権表示 ---
@title XYZ
@subtitle XYZ User's Manual
@subtitle Edition 1.0
@subtitle March 2001

@author  by Risa/Asir comitters
@page
@vskip 0pt plus 1filll
Copyright @copyright{} Risa/Asir committers
2001. All rights reserved.
@end titlepage

@comment --- おまじない ---
@synindex vr fn
@comment --- おまじない終り ---

@comment --- @node は GNU info, HTML 用 ---
@comment --- @node  の引数は node-name,  next,  previous,  up ---
@node Top,, (dir), (dir)

@comment --- @menu は GNU info, HTML 用 ---
@comment --- chapter 名を正確に並べる ---
@menu
* XYZ::
* Index::
@end menu

@comment --- chapter の開始 ---
@comment --- 親 chapter 名を正確に ---
@node XYZ,,, Top
@chapter XYZ

@comment --- section 名を正確に並べる ---
@menu
* XYZ について::
* XYZ に関する関数::
```

```
@end menu

@comment --- section の開始 ---
@node XYZ について,,, XYZ
@section XYZ について

@comment --- 書体指定について ---
@comment --- @code{} はタイプライタ体表示 ---
@comment --- @var{} は斜字体表示 ---
@comment --- @b{} はボールド表示 ---
@comment --- @samp{} はファイル名などの表示 ---

@b{Asir} は, xyz に関する計算をサポートしている.
xyz は @samp{xxx/xyz} で定義されている.
xyz の abc は次の通りである.

@comment --- @enumerate〜@end enumerate は番号付き箇条書き ---
@enumerate
@item abc は efg である.
@item abc の hij が klm することがある.
@end enumerate

xyz を abc した場合, 結果およびその意味を示す.

@comment --- @table〜@end table は表形式箇条書き ---
@comment --- @table の後に, 最初の column に対する書体指定をする ---
@table @code
@item 0

失敗

@item 0 以外の整数
成功
@end table

@node XYZ に関する関数,,, XYZ
@section XYZ に関する関数

@comment --- 関数名を正確に ---
@comment --- 複数の関数をまとめて説明できる ---
@menu
* xyz_abc::
* xyz_pqr xyz_stu::
@end menu
```

```
@comment --- 個々の関数の説明の開始 ---
@comment --- section 名を正確に ---
@node xyz_abc,,, XYZ に関する関数
@subsection @code{xyz_abc}
@comment --- 索引用キーワード
@findex xyz_abc

@comment --- 関数の簡単な説明 ---
@table @t
@item xyz_abc(@var{number})
:: @var{number} の xyz を abc する.
@end table

@comment --- 引数の簡単な説明 ---
@table @var
@item return
整数
@item number
整数
@end table

@comment --- ここで関数の詳しい説明 ---
@comment --- @itemize〜@end itemize は箇条書き ---
@comment --- @bullet は黒点付き ---
@itemize @bullet
@item
@var{number} の xyz を abc する.
@item
@var{number} は整数でなければならない.
@end itemize

@comment --- @example〜@end example は実行例の表示 ---
@example
[3] xyz_abc(123);
456
[4] xyz_abc(1.2);
xyz_abc : input must be an integer
@end example

@comment --- 参照 (リンク) を書く ---
@table @t
@item 参照
@fref{xyz_pqr xyz_stu}
```

```
@end table

@comment --- 複数の関数をまとめて説明する例 ---
@node xyz_pqr xyz_stu,,, XYZ に関する関数
@subsection @code{xyz_pqr}, @code{xyz_stu}
@findex xyz_pqr
@findex xyz_stu

@comment --- 関数の簡単な説明 ---
@comment --- @itemx は複数に対して説明を一つつける場合に使う ---
@table @t
@item xyz_pqr(@var{arg1},@var{arg2}[,@var{flag}])
@itemx xyz_stu(@var{arg1},@var{arg2})
:: xyz に関する操作.
@end table

@table @var
@item return
整数
@item arg1, arg2
整数
@item flag
0 または 1
@end table

@itemize @bullet
@item
@code{xyz_pqr()} は, @var{arg1}, @var{arg2} を pqr する.
@item
@var{flag} が 0 でないとき, モジュラ計算を行う.
@item
@code{xyz_stu()} は stu アルゴリズムを用いる.
@end itemize

@example
[219] xyz_pqr(1,2);
3
[220] xyz_pqr(1,2,1);
3
0
[221] xyz_stu(1,2);
3
@end example
```

```
@table @t
@item 参照
@fref{xyz_abc}
@end table

@comment --- おまじない ---
@node Index,,, Top
@unnumbered Index
@printindex fn
@printindex cp
@iftex
@vfill @eject
@end iftex
@summarycontents
@contents
@bye
@comment --- おまじない終り ---
```

書くべき場所に `@comment --- コメント ---` を入れてあるので, それに従えば書けると思うが, 一応要点をまとめておく.

### 20.3.1　構成

全体の構成は, 通常の LaTeX 文書等と同様で, ヘッダ部, タイトルその他および本文から成る. 本文に関しては, あるパッケージに対して 1 chapter を充てるとして, 概要, パッケージの構成, 機能説明などをいくつかの section で説明したあと, 個々の関数の説明をする section を設け, そこで各関数を subsection として説明する, というスタイルをとる.

### 20.3.2　書体指定

書体あるいは装飾指定の方法は以下の通り.

`@code{abc}`

: `abc` (タイプライタ体)

`@var{abc}`

: *abc* (斜字体)

`@samp{abc}`

: ‘`abc`’ (シングルクォートつきタイプライタ体)

`@b{abc}`

: **abc** (ボールド)

### 20.3.3 箇条書き

箇条書きは `@`*keyword* で始まり `@end` *keyword* で終る. 各項目は `@item` で始まる. 各キーワードの説明は次の通り.

`itemize`

`@itemize` の後ろに `@bullet` を通常伴って, 黒点つきの箇条書き.

`enumerate`

数字つきの箇条書き.

`table`

`@table` の後ろに書体指定を伴って, 表形式の箇条書き. 書体指定は第一 column に対する指定.

### 20.3.4 例の表示

LaTeX の verbatim 環境に対応するのが `@example` 〜 `@end example` である.

### 20.3.5 GNU info, HTML に関する指定

TeX で整形した場合には表に現れないが, GNU info や HTML 化した場合に必要となる指定として `@node` および `@menu` がある.

`@node` *Current*`,`*Previous*`,`*Next*`,`*Up*

chapter, section, subsection 間の関係を記述する. 最低限 *Current* の指定は必要であるがその他はオプションである. 関係が記述してあれば info あるいは HTML において対応するリンクが生成される.

`@menu` 〜 `@end menu`

`*`*name*`::` (*name* は chapter, section, subsection 等の名前) を列挙しておくと, info, HTML において menu ページが生成される.

### 20.3.6 TeX による整形

まず, texinfo マクロパッケージが必要だが, これは `OpenXM/src/asir-doc/jtexinfo.tex` および `OpenXM/src/asir-doc/texinfo-js.tex` をコピーして使う. また, index 生成のために `OpenXM/src/asir-doc/jtexindex/C/texindex` を使う. 今, カレントディレクトリにファイル `xyzman.texi` があり, OpenXM ソースツリーが `somewhere` にあるとする.

```
% cp somewhere/OpenXM/src/asir-doc/{jtexinfo.tex,texinfo-js.tex} .
% (cd somewhere/OpenXM/src/asir-doc; make)
<messages>
% ptex xyzman.texi
<messages>
% somewhere/OpenXM/src/asir-doc/jtexindex/C/texindex xyzman.??
% ptex xyzman.texi
```

これで xyzman.dvi ができる.

### 20.3.7　GNU info ファイルの生成

Version 1.68 以降の makeinfo が必要である. また, ファイルを EUC に変換するため, 例えば nkf が必要である.

```
% nkf -e xyzman.texi > xyzman_euc.texi
% makeinfo xyzman_euc.texi
```

これで info ファイルができる. ファイル名は @setfilename で指定した名前となる.

### 20.3.8　HTML ファイルの生成

Version 1.52 以降の texi2html が必要である. また, ファイルを EUC に変換するため, 例えば nkf が必要である.

```
% nkf -e xyzman.texi | sed -e "s/@fref/@ref/g" > xyzman_euc.texi
% texi2html -menu -split_node xyzman_euc.texi
```

これで xyzman_euc_toc.html および xyzman_euc_$n$.html ($n$ は通し番号) が生成される. xyzman_euc_toc.html が目次ファイルで, このファイルをブラウザで見ればよい.

## 20.4　マニュアルの書き方 その 2 : コメントから自動生成する

前節で述べた方法とは別に, プログラム中に埋め込まれた構造化されたコメントをもとに, texinfo ファイルを自動生成する方法もある.

これは Asir Contrib プロジェクト (OpenXM の一部) で開発しているツール gentexi.c を用いる方法である. このツールは Risa/Asir のプログラムに埋め込まれたコメントから, texinfo のマニュアルのソースを生成する. gentexi.c は OpenXM 配布ファイル群のディレクトリ OpenXM/src/asir-contrib/packages/doc にある.

次の例は asir contrib の関数 base_replace のマニュアルの記述例である.

```
/*&usage begin: base_replace(S,Rule)
 It rewrites {S} by using the rule {Rule}
 example: base_replace(x^2+y^2,[[x,a+1],[y,b]]);
 example_description: x is replaced by a+1 and y is replaced by b in
 x^2+y^2.
end: */
```

gentexi でとりだしたい部分は /*&usage で始める. : で終る単語が gentexi 用のキーワードである. 始めの begin: と 最後の end: は必須のキーワードである. 変数は { と } で囲む. 上の例以外のキーワードとしては, ref: や description: がある.

# 第21章　Risa/Asir に C で書いたプログラムを加えるには?

Risa/Asir は大部分が C 言語で記述されており, ソースコードも公開されているので, C 言語の心得がある人なら新機能を組み込み関数として追加したり, 改良, バグフィックスを行ったりすることも比較的簡単である. とはいえ, たとえば多項式を扱うような関数を追加する場合には, 多項式がどのような内部表現を持つか, メモリをどう確保するか, どのように初期化するかなど知らなければならないことが多い. ここでは,

- ファイルをさまざまなモードで open する
- 書き出し可能モードで open したファイルに 1 バイト書く.

という例で説明する.

## 21.1　asir2000 の source tree

asir2000 直下には以下のようなディレクトリがある.

include
: 内部形式を表現する構造体, 諸定数, 関数プロトタイプ, マクロなどを定義するヘッダファイルおよび, make 時の構成を定義する Risa.tmpl ファイルがある.

engine, asm, fft
: 内部形式で表現されたオブジェクトに対する諸演算が記述されている.

gc
: メモリ管理部.

parse
: キーボードおよびファイルから入力された文字列を, Asir の文法に従って解析し, 中間言語に変換する parser と, 中間言語を翻訳して, 対応する組み込み機能あるいはユーザ定義関数を呼び出し, 内部形式に変換する interpreter (evaluator) から成る.

builtin
: fctr, gcd などの組み込み関数インタフェースが定義されている.

io
: 内部形式を可読な文字列に変換して表示したり, バイナリ形式でのファイル入出力, プロセス間通信の管理および実際の送受信を行う.

`plot`

関数描画などを行う.

`lib`

Asir 言語で記述されたライブラリ.

## 21.2 builtin ディレクトリ

buitin ディレクトリには, カテゴリ別に仕分けされた組み込み関数を含む多くのファイルがある. 主なものは以下の通り.

| | |
|---|---|
| `algnum.c` | 代数的数を扱う. |
| `array.c` | 行列, ベクトルなどの配列を扱う. |
| `ctrl.c` | 実行制御のためのフラグを扱う. |
| `dp.c` | グレブナ基底計算で用いられる分散表現多項式を扱う. |
| `ec.c` | 楕円曲線に関する演算を扱う. |
| `fctr.c` | 多項式因数分解, GCD. |
| `file.c` | ファイル操作. |
| `miscf.c` | メモリ操作, 環境変数操作など特殊な機能. |
| `poly.c` | 多項式の形式変換, 高速乗算など. |
| `strobj.c` | 文字列操作. |

この他に, ユーザが組み込み関数を追加するために, `user.c` が用意してある. 機能を組み込み関数として追加する場合には, これらのファイルから適当なものを選び, そこに関数を追加すればよい.

## 21.3 例 — 一バイト書き出し関数の追加

ここでは, ファイルに 1 バイトを書き出す関数を追加するという作業を通して, 組み込み関数追加方法を説明する.

### 21.3.1 ファイルの選択

明らかに `builtin/file.c` に追加するのが望ましい.

### 21.3.2 関数名, 仕様を決める

既に, ファイルから一文字読み込む関数 `get_byte` があるので, `put_byte` とする. `get_byte` は引数がファイル記述子 (整数) のみであるが, `put_byte` では書き出すバイトとして 2 番目の引数に指定することとする.

### 21.3.3 open_file の改造

ファイルを open し, そのファイル記述子を返す組み込み関数 `open_file` の本体は次のようになっている.

```
void Popen_file(), Pclose_file(), Pget_line(), Pget_byte();
void Ppurge_stdin();
```

```
struct ftab file_tab[] = {
    {"purge_stdin",Ppurge_stdin,0},
        /* 関数本体は Popen_file, 引数は 1 */
    {"open_file",Popen_file,1},
    {"close_file",Pclose_file,1},
    {"get_byte",Pget_byte,1},
...
```

```
void Popen_file(arg,rp)
NODE arg;
Q *rp;
{
    char *name;
    FILE *fp;
    char errbuf[BUFSIZ];
    int i;

    /* 引数 (ファイル名) のチェック */
    asir_assert(ARG0(arg),O_STR,"open_file");
    /* ファイルポインタ配列の空きを探す */
    for ( i = 0; i < BUFSIZ && file_ptrs[i]; i++ );
    if ( i == BUFSIZ )
        error("open_file : too many open files");
    name = BDY((STRING)ARG0(arg));
    /* 読み込みモードで open */
    fp = fopen(name,"r");
    if ( !fp ) {
        sprintf(errbuf,"open_file : \"%s\" not found",name);
        error(errbuf);
    }
    /* ファイルポインタを格納 */
    file_ptrs[i] = fp;
    /* index をファイル記述子として返す */
    STOQ(i,*rp);
}
```

この状態では, ファイルは常に読み込みモードで open されるが, 書き出しモードが指定された場合に, 対応するモードでファイルが open されるように変更する.

```
struct ftab file_tab[] = {
    {"purge_stdin",Ppurge_stdin,0},
     /* 関数本体は Popen_file, 引数は最大で 2 */
    {"open_file",Popen_file,-2},
...
```

```
void Popen_file(arg,rp)
...
    name = BDY((STRING)ARG0(arg));
    /* 2 番目の引数が指定されたらそのモードで open */
    /* そうでなければ読み込みモードで open     */
    if ( argc(arg) == 2 ) {
        asir_assert(ARG1(arg),O_STR,"open_file");
        fp = fopen(name,BDY((STRING)ARG1(arg)));
    } else
        fp = fopen(name,"r");
    if ( !fp ) {
        sprintf(errbuf,"open_file : \"%s\" not found",name);
        error(errbuf);
    }
...
```

関数の引数は NODE という, リストを表す構造体により渡される. NODE は

```
typedef struct oNODE {
    pointer body;
    struct oNODE *next;
} *NODE;
```

と宣言されている. 第 $k$ 引数 ($k \geq 0$ ) を取り出す場合, メンバ next を $k$ 回たどる必要があるが, 組み込み関数の引数取り出し用に ARG0(arg), ···, ARG10(arg) が用意されている.

file_tab[] における引数の個数指定 $n$, 正の場合, $n$ 以外は不正, 負の場合, 0 以上 $n$ 以下の任意の個数を意味する. 上の変更の場合, 無引数の場合をチェックしていないのでやや不備である.

### 21.3.4 put_byte の追加

関数名は, 慣習上組み込み関数名の先頭に “P” を付ける. 即ち今の場合 Pput_byte である. まず, file_tab[] にエントリを追加する. このテーブルの上にプロトタイプ void Pput_byte(); を宣言する. さらに, テーブルの適当な位置に, 先に決定した仕様で登録する.

```
void Pput_byte();
...
struct ftab file_tab[] = {
    {"purge_stdin",Ppurge_stdin,0},
    /* 関数本体は Popen_file, 引数は最大で 2 */
    {"open_file",Popen_file,1},
    {"close_file",Pclose_file,1},
    {"get_byte",Pget_byte,1},
    /* 追加したエントリ. 引数は 2 */
    {"put_byte",Pput_byte,2},
...
```

次に, 関数本体を file.c の適当な位置に書く.

```
void Pput_byte(arg,rp)
NODE arg;
Q *rp;
{
    int i,c;
    FILE *fp;

    /* 引数のチェック */
    asir_assert(ARG0(arg),O_N,"put_byte");
    asir_assert(ARG1(arg),O_N,"put_byte");
    /* ファイル記述子の取り出し */
    i = QTOS((Q)ARG0(arg));
    /* 書き出すバイトの取り出し */
    c = QTOS((Q)ARG1(arg));
    if ( fp = file_ptrs[i] ) {
        /* ファイルへ書き出す */
        putc(c,fp);
        /* この関数の出力を作る */
        STOQ(c,*rp);
    } else
        error("put_byte : invalid argument");
}
```

マクロ QTOS は, 内部形式で有理数として表現されているものの, 実際には 32 bit 符号付き整数であることが分かっているオブジェクトを 32 bit 符号付き整数に変換する. また, STOQ はその逆の働きをする. 関数は必ず何らかのオブジェクトを内部形式で返さなければならない. それを書き込む場所は, この関数でいえば rp というポインタとして渡されているので, そこにオブジェクトのポインタを書く. この関数の場合, 書いたバイトをそのまま返すことにしている.

## 21.4　error 処理

組み込み関数において, 引数が不正な場合などのエラー処理は, 通常 `error(message)` という関数を呼び出すことで行う場合が多い. ここで, `message` はエラーの理由などを示す文字列へのポインタである. この関数は, 然るべき後始末を行って, `message` を表示したあと, 以下を実行する.

- 問題の組み込み関数が, トップレベル, 即ちプロンプトに対して直接入力されて実行された場合には, そのままトップレベルに戻る.
- 関数中で実行された場合には, デバッガを呼び出す.

また, 関数引数の簡易チェックである `asir_assert()` はマクロであり, やはりエラー時には `error` を呼び出す.

## 21.5　プログラムの解読

他の人の書いたプログラムを読むのは勉強になる. しかしプログラムを眺めていても, なかなか理解できない. そのようなときは, プログラムの中にデータのプリント文を挿入して, プログラムの動きを調べると参考になることが多い. Asir には `printexpr(変数リスト, 任意のオブジェクト)` という C の関数があるのでこれを適宜 Asir のソースコードに挿入してコンパイル, 実行するとプログラムの動きを調べる助けとなる.

# 第22章　Mathematica と Asir

本書では Risa/Asir を用いて基礎的なプログラミングおよび数学プログラミングを練習した. 一旦, Risa/Asir のプログラミングを修得すれば, Mathematica や Maple などの他の数式処理系を利用するのも簡単である.

この節では, Risa/Asir と Mathematica を対比させながら, Mathematica のプログラムの書き方を解説しよう.

以下, まだ書いてない.

# 第23章　付録: エラー

## 23.1　Windows のエラー

**疑問 23.1**　Windows の反応がとってもおそくなったり、グラフをかいていたら突然止まってしまったり、メモリーがないから書けないよ、といってきたりする. どうしてか？

**答え 23.1**　RAM (メモリ) の不足でこのようなことがおこる. とりあえず, 止められるなら計算を中断して, マシンを再起動して, 再計算してみる. それでも同様な状態になるなら, 実際にメモリが不足しているということなので, (お金に余裕のある人は) RAM を買い足せばよい.

ただし, 計算を数時間続けてメモリが足りない事態に至るときは, アルゴリズムの本質的改良がないと解けない場合がおおい. 計算量の議論を思いだそう.

## 23.2　Asir のエラー と Q & A

**疑問 23.2**　; RETURN を入力しても Asir のプロンプトが戻って来ない. 返事がない. なぜか？

**答え 23.2**　" 記号や ’ 記号は組にして使用する. たとえば `print("Hello);` としていて, Hello のあとの " を忘れている可能性がある. そのときは " ; RETURN でプロンプトが戻るはずである.

{ と } も組にして使用する. これが閉じていないとやはりプロンプトがもどってこないことがある. そのときは } RETURN でプロンプトが戻るはずである. または, CTRL+C を入力して, プロンプトにもどしてもよい.

**疑問 23.3**　次のプログラムを実行したらいつまでも終りません. ; RETURN を入力してもプロンプトがでません.

```
S=0$
for (I=0; I<100; I+1) {
  S = S+I;
}
print(S)$
```

**答え 23.3**　I の値が更新されない無限ループにおちいっていてプログラムの実行中です. CTRL+C でプログラムを中断して下さい.

`for (I=0; I<100; I+1)` は誤りで, `for (I=0; I<100; I=I+1)` か `for (I=0; I<100; I++)` に訂正して下さい.

さて、Asir の言語処理系からのエラーメッセージをみよう.

**疑問 23.4**

```
[346] fctr(x^2-@i*y^2);
fctrp : invalid argument
return to toplevel
```

**答え 23.4**　取り扱えない引数があたえられている (invalid argument). 有理数を係数とする多項式しか因数分解できない. $x^2 - iy^2$ は複素数を係数としている.

**疑問 23.5**　次のプログラムの動作が変です.

```
def foo(M) {
  G1 = x+y;
  G2 = x-y;
  G = GM;
  return(G^3);
}
```

**答え 23.5**　このプログラムはもしかして,

```
def foo(M) {
  if (M == 1) {G = x+y;}
  if (M == 2) {G = x-y;}
  return(G^3);
}
```

というつもりで書いたのでないでしょうか? GM は 引数 M の値に応じて, G1, G2 と解釈されるわけではなくて, GM という新しい局所変数と解釈され, 初期値は 0 となります.

**疑問 23.6**　asirgui の画面で, マウスを click してそこにある入力を修正しようとしてもできません.

**答え 23.6**　asirgui は Windows 標準の ユーザインタフェースにしたがっていない. むしろ, unix のターミナルに似せてつくってある. たとえば, Windows 版 asirgui 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.

# 第24章　付録: パス名

## 24.1　ファイル名の付け方

作成した文書, プログラム等はファイル (file) としてハードディスク, CD やフロッピィディスク等に格納されている. ファイルには名前をつけて格納するのであるが, 名前として利用できる文字には制限があったり, またファイルの名前づけについては慣習がある.

ファイルの名前として利用される文字は OS の種類によりいろいろ違いがあるが, アルファベット A から z まで, 数字, それから ピリオド `.`　はよく利用されるほとんどの OS でファイルの名前として用いることが可能である. 実習でも, 日本語や特殊記号, 空白の入ったファイル名などは利用しないのが無難であろう.

ピリオドのあとにファイルの種別を表す文字をつけるのが習慣である. たとえば, C 言語のソースは, `hoge.c` とか `test.c` なる名前であり, この本では Asir 言語のソースは, `hoge.rr` とか `test.rr` なる拡張子 `.rr` をつけて統一した. ちなみに `r` は Risa の `r` である.

CP/M80, MSDOS が [8 文字以内] ピリオド [3 文字以内] というファイルの名前 ( 8.3 システムと呼ぶ) しか許さなかったという歴史的事情により, ファイルの名前がこのような形式になっている場合も多い. たとえば Windows では, テキストファイルは `hoge.txt` とか `letter.txt` なる名前がついていることが多いが, この `txt` 部が 3 文字であることに注意しよう. また Windows では HTML ファイルは `hoge.htm` とか `index.htm` なる名前のことが多いが, この `htm` も 8.3 システムの名残である. 現在の Windows では, ロングネームによるファイル名と 8.3 形式によるファイル名の二つのファイル名をもつ. 8.3 形式の方は自動的につけられる場合がおおい. ロングネームによるファイル名は, 8.3 形式によるファイル名の制限はない. コマンドプロンプトの `dir` コマンドでこの両方の名前をみることが可能である.

## 24.2　ディスクのなかにどんなファイルがあるのか調べたい

Windows ファイルエクスプローラを用いてしらべればよい. または MSDOS プロンプト または コマンドプロンプトを起動して, `dir` コマンドをもちいる. ディレクトリ間の移動には `cd` コマンドを用いる.

Unix シェルより `ls` コマンドをもちいる. ディレクトリ間の移動には `cd` コマンドを用いる. `ls -laR` で, カレントディレクトリのしたにある全てのファイルを表示できる (l は long, a は all, R は recursive).

## 24.3　階層ディレクトリ

たとえば 1 万個のファイルをあなたが所有していたとする. Unix の `ls` コマンドでファイルの一覧を表示したら, 1 万個ものファイルの名前が表示されたらとしたらとても目的のファイルを発見できない. したがって, ファイルは意味あるまとまり毎にまとめておくべきであろう. これを実現する

仕組みが階層ディレクトリである. Windows では普通フォルダと呼ぶ. フォルダという用語法およびフォルダをグラフィックで表示して階層ディレクトリを直観的に扱えるようにしたのは Macintosh が最初に普及させた技術であることはとくに名誉のために強調しておこう.

ファイルをまとめておく特別なファイルをディレクトリ (directory) と呼ぶ. 以下, フォルダという用語法はつかわない.

ディレクトリの考えを例を用いて説明しよう. 現在ファイル hoge1.txt, hoge2.txt, foo.rr, esc.rr を所有している. あたらしくディレクトリ text および prog を作成してこれらのファイルをそちらへ移動して分類したい. text には文書類, prog にはプログラムをまとめて格納するものとする.

Unix

```
bash$ mkdir text
bash$ mkdir prog
bash$ cp hoge1.txt text
bash$ cp hoge2.txt text
bash$ cp foo.rr prog
bash$ cp esc.rr prog
bash$ rm hoge1.txt hoge2.txt
          foo.rr esc.rr
```

MSDOS プロンプト またはコマンドプロンプト

```
C> mkdir text
C> mkdir prog
C> copy hoge1.txt text
C> copy hoge2.txt text
C> copy foo.rr prog
C> copy esc.rr prog
C> del hoge1.txt hoge2.txt
         foo.rr esc.rr
```

mkdir は空のディレクトリを作成するコマンドである. ディレクトリにファイルをコピーすると, そのファイルはそのディレクトリ内に格納される. 今の例の場合ファイルの構造は次のようになる.

```
text -------  hoge1.txt
      |______ hoge2.txt

prog -------  foo.rr
      |______ esc.rr
```

ディレクトリの中にディレクトリを作成してもいい. たとえば

```
bash$ mkdir prog/asirprog
bash$ cp prog/esc.rr prog/asirprog
```

```
C> mkdir  prog¥asirprog
C> copy prog¥esc.rr  prog¥
asirprog
```

であたらしくディレクトリ asirprog が prog の中に作成される. この場合ファイル全体を図示すると次のようになる.

```
text -------  hoge1.txt
      |______ hoge2.txt

prog -------  foo.rr
      |______ esc.rr
```

```
|______ asirprog ----- esc.rr
```

したがって, ファイル全体は木のような構造になる. あらかじめ用意してある一番最初のディレクトリをルートディレクトリと呼ぶ.

ファイル名をルートディレクトリから表したものをフルパス名 (full path name) とよぶ. `text`, `prog` はルートディレクトリ直下にあると仮定しよう. `esc.rr` のフルパス名は unix では

```
/prog/esc.rr
```

となる. `esc.rr` のフルパス名は Windows (MSDOS) では

```
¥prog¥esc.rr
```

となる. 最初の `/` や ¥は, ルートディレクトリを意味する. なお, Asir で `load` コマンドを用いるときは Unix でも Windows (MSDOS) でも `/` を用いる.

ファイル名を指定するのに毎回フルパス名を利用するのはつかれる. そこで便宜上, 現在自分がいるディレクトリを仮定し, これを カレントディレクトリ (current directory) と呼ぶ. カレントディレクトリを変更しディレクトリ内を移動するコマンドが `cd` (change directory) である.

```
cd ディレクトリ名
```

で引数として与えたディレクトリへ,

```
cd ..
```

で一つ上のディレクトリに移動できる. Unix では `pwd` (print working directory) でカレントディレクトリを表示できる. MSDOS では `cd` でカレントディレクトリを表示できる. パス名の最初に`/` や¥をかかないと, カレントディレクトリからの名前とみなされる. たとえば, カレントディレクトリが `prog` だとしよう. このとき `esc.rr` も`/prog/esc.rr` も同じ意味となる.

空のディレクトリは削除できる. 削除のためのコマンドは

```
rmdir ディレクトリ名
```

である.

## 24.4 ドライブ名

Windows (MSDOS) の場合さらにドライブ名およびカレントドライブの概念がある. 一般的な構成の PC では, `A` ドライブがフロッピーディクスドライブ, `C` ドライブが内蔵のハードディスクドライブである. ドライブ名を指定するには, 最初に

```
ドライブ名:
```

と書く. ドライブまで指定してファイル名を指定するにはたとえば,

```
C:¥prog¥esc.rr
```

と書く. ドライブ名を省略するとカレントドライブがもちいられる. MSDOS プロンプトを起動すると,

```
C:￥Windows>
```

と表示されるが, これは カレントドライブが C でカレントディレクトリが￥Windows であることを意味する. たとえばカレントドライブを A に変更するには

```
A:
```

と入力するればよい.

## 24.5　自分はどこにファイルをセーブしたの?

MicroEmacs や Emacs の一番下の行を, ファイルをセーブするときに注意深くみるとセーブしたファイルのフルパス名がわかる.

たとえばディスクトップにファイル hoge.rr をセーブしてしまうと, そのフルパス名は

```
C:￥Windows￥デスクトップ￥hoge.rr
```

みたいになってるはずである.

MicroEmacs では, ドライブ名が自動的に付加されるときがある. 自動的に付加された名前にさらにドライブ名を付加すると, そのフルパス名は

```
A:￥A:hoge.rr
```

みたいになってるはずで, 余計に付加した A: はファイル名の一部となっているはずなので, この場合は A: を書いてはいけない.

**問題 24.1 (05)** ディレクトリを用いてあなたのハードディスクおよびフロッピーディスクを整理しなさい.

## 24.6　Q and A

### 24.6.1　新しく買ってきたフロッピーディスクを A ドライブにいれましたが, ランプがついたままでセーブできません.

新しいフロッピーディスクは, format が必要な場合がある. この場合は format をまだしていないと予想できる. Windows の場合, デスクトップのマイコンピュータ (My computer) フォルダの 3.5 インチディスクアイコンの場所で, マウスの右ボタンをクリックするとフォーマットのメニューをだせる. または, MSDOS プロンプトよりコマンド

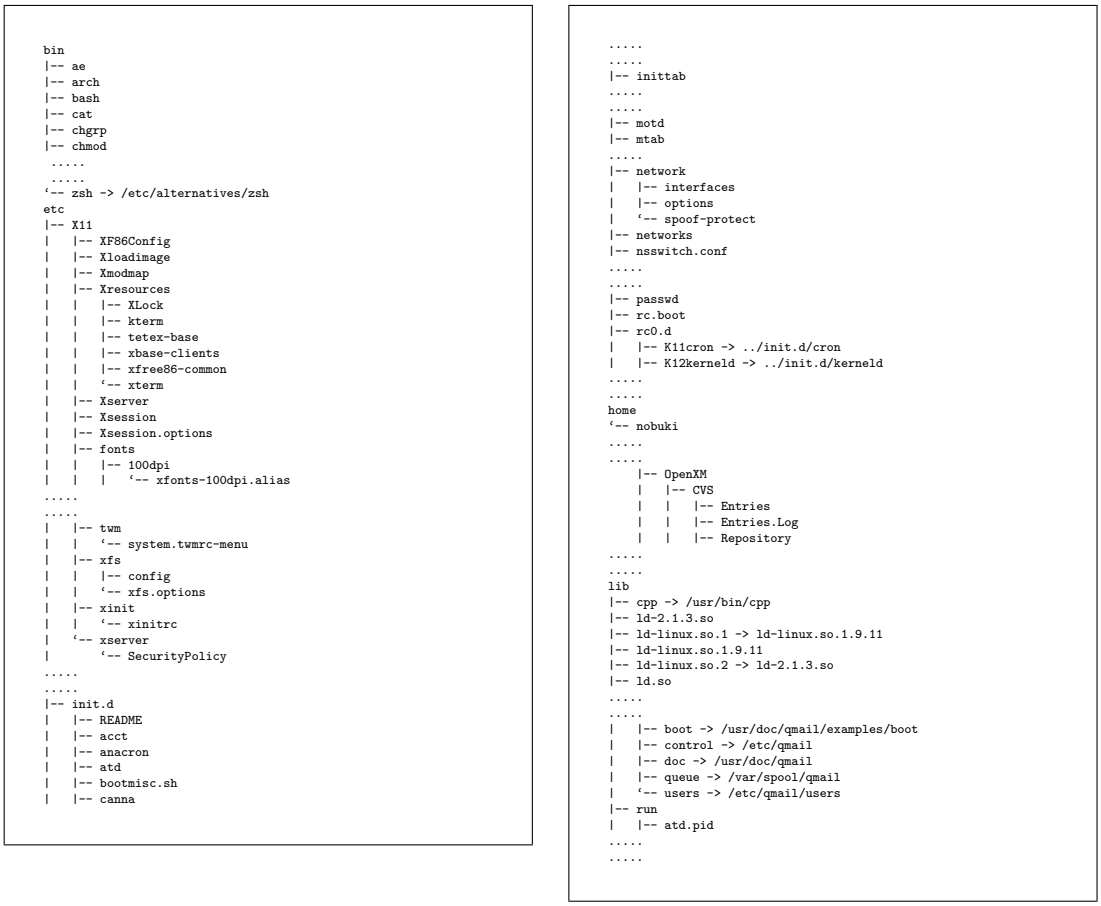
```
format a:
```

でフォーマットをおこなうのも可能である.

使用されてことのあるディスクを format するといままでのデータはすべて消え去る.

フロッピーディスクは, トラック (track) とセクター (sector) に分けられており, ディスクに, ここからは track 1, sector 1 ですよ, このマークのあとに track 1, sector 1 に書き込むデータを書いて下さい, とか, ここからは track 1, sector 2 ですよ, このマークのあとに track 1, sector 1 に書き込むデータを書いて下さい, とかいう情報をまず書き込むのが format という操作である. 一般に市販されているフロッピーは 1.2M または 1.4M 用と書いてあるはずだが, format コマンドは 1440 KB

(1.4M) の容量をフロッピーディスクが持つようにトラック, セクターの切りわけをする. 日本では NEC の PC9801 が長くつわれていた, PC9801 では 1.2M のディスクを用いていた, という歴史的理由により 1.2M の容量をフロッピーディスクが持つように format できる機種も存在する.




Risa/Asir ドリル ギャラリー : Debian GNU Linux の / からのファイル tree の一部. 最初から順番にフルパスで書くと/bin/ae, /bin/arch, /bin/bash (シェル bash はここにある), /bin/cat, /bin/chgrp, /bin/chmod, ..., /etc/X11/XF86Config (X window システムの設定ファイル), /etc/Xloadimage, ...

# 第25章　付録: 実習用 CD の利用法

## 25.1　CD の構成

付録 CD に収録されているのは Risa/Asir および OpenXM 版 Risa/Asir (OpenXM/Risa/Asir) である.

### 25.1.1　asirgui の起動

1. デスクトップの マイコンピュータ を ダブルクリックして開き, そのなかの CDROM をダブルクリックして開く.
2. CDROM の中のフォルダ `asir-book` をダブルクリックして開く.
3. フォルダ `asir-book` のなかのフォルダ `asir` を開きさらに その中の `bin` フォルダ (binary, 実行可能ファイルをおく置場) をダブルクリックして開く.
4. フォルダ `bin` に `asirgui.exe` (または `asirgui`) があるので, `asirgui` のアイコン

をダブルクリックしてスタートする.

参考: 一般に Windows の実行可能なファイルは `.exe` か `.com` か`.bat` なる拡張子を持つ.

### 25.1.2　Meadow のインストール (Windows)

Windows 用の emacs である Meadow がフォルダ `asir-book` の中の, フォルダ `Meadow` にある. これをダブルクリックすると, Meadow がインストールされる.

### 25.1.3　その他のファイル

1. `asir-book￥index.html` をダブルクリックすると, HTML 版の `Asir` のマニュアルを見れる.
2. `Asir` のソースコードは `openxm-head.tar.gz` に含まれている. これは unix tar 形式ファイルを, gzip で圧縮したファイルである. Windows で展開し, ファイルの中身を見るには, `tar` と `gunzip` が必要である. フリーソフト, シェアウエアの配布サイトである `http://www.vector.co.jp` 等から `tar`, `gunzip` を取得することが可能である.
3. `asir-book￥Prog` ディレクトリには, この本で説明しているプログラムが収録してある.

## 25.2 Q and A

### 25.2.1 ハードディスクにコピーして利用したい

ファイルエクスプローラで CD を開き, その中の asir-book フォルダをハードディスクの好きな場所へドラッグ & ドロップすればハードディスクにフォルダごとコピーできる.

現在, 日本語をフォルダ名として含むフォルダへコピーした場合正しく動作しない現象が報告されています. したがって, デスクトップおよびそれの上へ作成したフォルダへコピーしてから使用することはできません. ファイルエクスプローラで, ハードディスク (普通 C ドライブ) を開き, 直接 asir-book フォルダをハードディスクへコピーして下さい.

### 25.2.2 asirgui を Windows 95, 98, ME で利用したい

asirgui は Windows 2000 で開発されていますが, いちおう Windows 95, 98, ME でも動作します. だたし, Windows 95, 98, ME では不可解な出来事やパーフォーマンスの低下がある場合があります.

### 25.2.3 asirgui を起動しようとすると DLL WS2_32.DLL がありません とエラーがでる. (古い Windows 95)

古い版の Windows 95 では, asirgui の実行に必要な DLL (Dynamic Link Library) を C:￥Windows￥system にコピーしておく必要があります. WS2_32.DLL は Winsock2 (Windows 用の socket ライブラリ) の DLL です.

1. W95ws2setup.exe を実行.
2. rvb5_1109a0.lzh をアーカイバ lha32 で展開し DLL ファイルをコピー.

```
mkdir mytmp
cd mytmp
lha32   x  rvb5_11-9a.lzh
copy  *.dll c:￥Windows￥system
```

### 25.2.4 Asirgui を起動しようとすると, Engine.exe は 欠落エクスポート OLEaut32.DLL にリンクされている なるエラーメッセージがでる. (古い Windows 95)

古い版の Windows 95 では, asirgui の実行に必要な DLL (Dynamic Link Library) を C:￥Windows￥system にコピーしておく必要があります. Winsock2 (Windows 用の socket ライブラリ) の DLL はすでに更新されているようなので, 上の場合と違い, winsock の更新は必要ありません.

1. rvb5_1109a0.lzh をアーカイバ lha32 で展開し DLL ファイルをコピー.

```
mkdir mytmp
cd mytmp
lha32   x  rvb5_11-9a.lzh
copy  *.dll c:￥Windows￥system
```

### 25.2.5 … プログラムの製造元に連絡して下さい. ENGINE のページ違反です. モジュール ENGINE.EXE アドレス 0137:2004037a9 ….

メモリーをすべてつかいきったときに, このエラーがでる場合があります.

Risa/Asir ドリル ギャラリー : マザーボード (Athlon dual)

# 第26章　付録: ソースコードの入手

## 26.1　著作権情報

Risa/Asir はいわゆるオープンソースのシステムであり, ソースコードは, 次のライセンスのもと富士通研究所から公開されている. 以下原文を掲載する.

```
1. License Grant:

FUJITSU LABORATORIES LIMITED ("FLL") hereby grants you a limited,
non-exclusive and royalty-free license to use, copy, modify and
redistribute, solely for non-commercial and non-profit purposes, the
computer program, "Risa/Asir" ("SOFTWARE"), subject to the terms and
conditions of this Agreement. For the avoidance of doubt, you acquire
only a limited right to use the SOFTWARE hereunder, and FLL or any
third party developer retains all rights, including but not limited to
copyrights, in and to the SOFTWARE.

2. Restrictions

(1) FLL does not grant you a license in any way for commercial
purposes. You may use the SOFTWARE only for non-commercial and
non-profit purposes only, such as academic, research and internal
business use.

(2) The SOFTWARE is protected by the Copyright Law of Japan and
international copyright treaties. If you make copies of the SOFTWARE,
with or without modification, as permitted hereunder, you shall affix
to all such copies of the SOFTWARE the copyright notice specified
below.

Copyright 1994-2000 FUJITSU LABORATORIES LIMITED
All rights reserved.

(3) An explicit reference to this SOFTWARE and its copyright owner
shall be made on your publication or presentation in any form of the
results obtained by use of the SOFTWARE.

(4) In the event that you modify the SOFTWARE, you shall notify FLL by
e-mail at risa-admin@flab.fujitsu.co.jp
of the detailed specification for such modification or the source code
of the modified part of the SOFTWARE.

3. Third Party's Software:

Multi-precision floating point number operations (arithmetic,
mathematical function evaluation and many other related functions) of
Risa/Asir are performed by PARI system developed by C. Batut,
D. Bernardi, H. Cohen and M. Olivier. If you use the source code
version of the SOFTWARE, the source code files and related
documentation of PARI are not provided by FLL but are made available
at ftp://megrez.ceremab.u-bordeaux.fr/pub/pari"

If you have questions regarding PARI, please contact
pari@ceremab.u-bordeaux.fr

4. Disclaimer of warranty:
```

```
THE SOFTWARE IS PROVIDED AS IS WITHOUT ANY WARRANTY OF ANY KIND. FLL
MAKES ABSOLUTELY NO WARRANTIES, EXPRESSED, IMPLIED OR STATUTORY, AND
EXPRESSLY DISCLAIMS ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT OF THIRD PARTIES'
RIGHTS. NO FLL DEALER, AGENT, EMPLOYEES IS AUTHORIZED TO MAKE ANY
MODIFICATIONS, EXTENSIONS, OR ADDITIONS TO THIS WARRANTY.

5. Limitation of Liability:

1.  UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, TORT, CONTRACT,
OR OTHERWISE, SHALL FLL BE LIABLE TO YOU OR ANY OTHER PERSON FOR ANY
DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE OR CONSEQUENTIAL
DAMAGES OF ANY CHARACTER, INCLUDING, WITHOUT LIMITATION, DAMAGES
ARISING OUT OF OR RELATING TO THE SOFTWARE OR THIS AGREEMENT, DAMAGES
FOR LOSS OF GOODWILL, WORK STOPPAGE, OR LOSS OF DATA, OR FOR ANY
DAMAGES, EVEN IF FLL SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF
SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.  2. EVEN IF A PART
OF THE SOFTWARE HAS BEEN DEVELOPED BY A THIRD PARTY, THE THIRD PARTY
DEVELOPER SHALL HAVE NO LIABILITY IN CONNECTION WITH THE USE,
PERFORMANCE OR NON-PERFORMANCE OF THE SOFTWARE.

6. Equitable Relief

You agree that your breach of the terms and conditions hereof may
cause FLL irreparable harm for which there may be no adequate remedy
at law, and FLL may therefore seek equitable relief.

7. Miscellaneous

1. You shall comply with all export controls laws and regulations in
your country or in other countries which may be applied in connection
with the SOFTWARE.

2. Use, duplication or disclosure of the SOFTWARE by the U.S
Government is subject to restrictions set forth in subparagraph (a)
through (b) of the Commercial Computer- Restricted Rights clause at
FAR 52.227-19 when applicable, or in subparagraphs (c)(1) and (2) of
the Rights in Technical Data and Computer Software clause at DFAR
252.227-7013, and in similar clauses in the NASA FAR
Supplement. Contractor/manufacturer is FUJITSU LABORATORIES LIMITED,
4-1-1 Kamikodanaka Nakaharaku Kawasaki 211-8588, Japan.
```

大事な部分をまとめておくと, 要するに, 商用でなければ, 改変および再配布はよい, それから, 変更をした場合は, 富士通研究所にそのソースコードおよびその変更点についての解説を送らねばならないという規定がある.

## 26.2 ソースの所在地

Risa/Asir の安定版は, 富士通研究所の web サイトまたは

http://risa.cs.ehime-u.ac.jp

より取得することが可能になる予定である.

本書で利用している Risa/Asir は安定版をもとに改変を加えてある通称 Risa/Asir Kobe Distribution である. ソースコードおよび Windows 用バイナリ (実行可能形式) は,

http://www.math.kobe-u.ac.jp/Asir/index.html

より取得可能である. Asir contrib パッケージ, OX サーバ等を含んだ, OpenXM/Risa/Asir は

http://www.openxm.org

または

http://www.math.kobe-u.ac.jp/OpenXM

より取得可能である. またこのページにある, CVS web サーバの OpenXM_Contrib2/asir2000 をみるとソースコードの変更状況をみることが可能である. この CVS web サーバで, STABLE タグがついてるのが, 富士通研究所の安定版, HEAD タグがついているのが, 本書に収録されている Risa/Asir のソースの最新版である. なお, たびたびでてくる OpenXM というのは, 数学ソフトをたがいに接続しようという実験プロジェクトである.

最新版のソースは, OpenXM パッケージの一部として,

ftp://ftp.math.kobe-u.ac.jp/pub/OpenXM/Head/

の下においてあるので, unix 系コンピュータを利用している場合は, このソースを取得し, 以下のように入力すれば, 多くの unix 系コンピュータで, OpenXM 版の Asir をインストール可能である. くわしくは, ソースコードに含まれる文書を御覧になられたい.

```
cd OpenXM/src
make configure
make install-asir install-asir-doc
cd ../rc
make install
```

なお,

```
cd ../rc
make install
```

は root になり実行する必要がある. /usr/local/bin の下に, Asir をスタートするためのシェルスクリプトをコピーするためである.

OpenXM 版のソースコードは, pari やマニュアル作成のための環境等も同時に配布しているので, make が簡単であろう. なお, 上でいきなり OpenXM/src の下で make install すると, Asir 以外の不要な OpenXM サーバもコンパイルするので, 要注意である. OpenXM 版は, 起動時に OpenXM 対応のライブラリ群をロードするので, 本来の生の Asir をインストールしたい場合は, OpenXM_contrib2/asir2000 の下の INSTALL に従い, Asir をインストールするとよい.

なお, Omoikane Debian GNU Linux ( http://www.omoikane.co.jp) の math distribution CD には Risa/Asir バイナリが標準添付されている. Make がいやなひとは, このディストリビューションを試す手もある.

最新版の Risa/Asir に cvsup をもちいて同期させる方法等は, ソースディストリビューションの文書 OpenXM/doc/Papers/rims2000.tex および cvsup 用 supfile OpenXM/misc/openxm-cvs-supfile を参照いただきたい.

## 26.3 マニュアルの所在地

http://www.math.kobe-u.ac.jp/OpenXM/1.2.1/doc/asir2000/html-jp/man_toc.html

に OpenXM 1.2.1 版に対応する Asir 日本語マニュアル

野呂, 下山, 竹島： Asir User's Manual.

の HTML 版が存在している. ソースディストリビューションの `OpenXM/src/asir-doc` にマニュアルの texinfo ソースがある. PS 版, dvi 版のマニュアルも OpenXM のホームページより取得可能である.

1.2.1 版 OpenXM の Asir-contrib ライブラリのマニュアルは,

`http://www.math.kobe-u.ac.jp/OpenXM/1.2.1/doc/asir-contrib/html-ja/cman-ja_toc.html`

にある.

“Risa/Asir ドリル” はオンライン書籍ですので, 入手法が普通の書籍とは異なります. この本を引用する場合は, 次のようにお願いいたします.
簡単版:

野呂正行, 高山信毅 著: Risa/Asir ドリル, 2003 (入手については
`http://www.math.kobe-u.ac.jp/Asir/index-ja.html` を参照)

詳細版:

野呂正行, 高山信毅 著: Risa/Asir ドリル, 2003
Risa/Asir を用いた (数学) プログラミングの入門書.
`http://www.math.kobe-u.ac.jp/Asir/index-ja.html` より PDF, PS 版を入手可能. 配布, 複製は自由. 講義, セミナー等で利用の際は印刷製本 (Windows 版 Asir CD 付き) したものを実費配布可. takayama@openxm.org まで連絡.

# 索 引