Risa/Asir ドリル, 2011
計算数学 I· 同演習用ダイジェスト版

高山信毅, 野呂正行

2011 年 (平成 23 年), 9 月 20 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

追記: このテキストによる講義例として下記のビデオを公開している.

`http://fe.math.kobe-u.ac.jp/Movies/cm/2006-keisan-1-nt.html`

目 次

第1章　超入門 Cfep/asir (MacOS X)

1.1　電卓としての利用

神戸大学の教育用計算機環境が MacOS X に変更されるのに伴い, 筆者が教材として利用していた Windows で動作する 10 進 Basic が利用できなくなった. Cfep/asir はその代用として, 2006 年初頭から開発を進めているシステムである. 10 進 Basic の優れている点の一つは, 丁寧な入門解説が付属していることである. “Cfep/asir 超入門” はこの解説にすこしでも近付こうと努力してみた. Asir の入門テキストに “Asir ドリル” があるが, この超入門では “Asir ドリル” の一章およびその先の入門的内容を丁寧に (少々くどく) 説明した.

この節では MacOS X での cfep/asir の起動法, 電卓風, Basic 風の使い方を説明する. ファイルの保存等 MacOS X の共通の操作方法にはほとんどふれていないが, cfep/asir は MacOS X 標準のファイルの保存等を用いているので, このような部分では他のソフトウエアと利用方法は同一である. 初心者の人は適当な本やガイドを参照されたい.

1.1.1　キー操作と用語の復習

キーボード, マウスの操作の用語.

1. Command キーや 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. command キー, alt キー, ctrl キーについても同様である.

2. SHIFT+a とすると大文字の A を入力できる.

3. 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.

4. 日本語キーボードの場合 \ (バックスラッシュ) は ALT+¥ で入力できる.

5. SPACE キーは空白を入力するキーである. 計算機の内部では文字は数字に変換されて格納および処理される. 文字に対応する数字を文字コードと呼ぶ. 文字コードにはいろいろな種類のものがあるが, 一番基礎的なのはアスキーコード系であり, アルファベットや数字, キーボードに現れる記号などをカバーしている. 漢字はアスキーコード系では表現できない. A のアスキーコードは 65 番である. 以下 B が 66, C が 67, と続く. 空白のアスキーコードは 32 番である. 日本語入力の状態で入力される空白は “全角空白” と呼ばれており, アスキーコード 32 番の空白 (半角空白) とは別の文字である. 全角空白がプログラムに混じっているとエラーを起こす. asir ではメッセージやコメント等に日本語が利用可能であるが, 慣れるまでは英字モードのみを利用することをお勧めする. 右上の言語表示が

となっている状態で cfep/asir に入力しよう.

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. プログラムを読む時に注意. また, プログラムを読む時は 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. 計算機が忙しい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.

1.1.2 Cfep/Asir の起動法と電卓的な使い方

cfep のアイコン (いのぶた君)

をダブルクリックすると図 1.1 のように cfep/asir が起動する. 以下 cfep/asir を単に asir とよぶ.

図 1.1 の入力窓に計算したい式やプログラムを入力して “始め” ボタン

をおすと実行を開始する. 式の計算やプログラムの実行が終了すると, 新しいウインドウ OutputView が開き結果がそのウインドウに表示される. “始め” ボタンをおして実行を開始することを計算機用語では “入力の評価を始める” という.

出力小窓にはシステムからのいろいろな情報が出力されるが, 内容は中上級者向けのものが多い.

ファイルメニュー

図 1.1: cfep/asir の起動画面

から” 保存” や” 別名で保存” を実行すると入力窓の内容をファイルとして保存できる. 出力小窓の内容や OutputView の内容は保存されないので注意してほしい.

cfep/asir を完全に終了するには cfep メニュー

の “cfep を終了” を実行する.

さて図 1.1 では $3 \times 4 + 1$ の計算をしている.

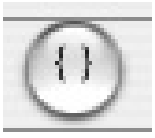
Asir における計算式は普通の数式と似ていて, 足し算は $+$, 引き算は $-$ と書く. かけ算と割算は $\times$ や $\div$ がキーボードにないという歴史的理由もあり, それぞれ `*` と `/` で表現する. 累乗 P^N は `P^N` のように `^` 記号を用いて表す.

式の終りを処理系 (asir) に教える (示す) のに `;` (セミコロン) を書かないといけない. 文末の “。” のような役割を果たす. またかけ算の記号 `*` の省略はできない.

例題 1.1 以下の左の計算式を asir では右のようにあらわす.

$2 \times (3 + 5^4)$	`2*(3+5^4);`
$\left\{\left(2 + \frac{2}{3}\right) \times 4 + \frac{1}{3}\right\} \times 2 + 5$	`((2+2/3)*4+1/3)*2+5;`
$AX + B$	`A*X+B;`
$AX^2 + BX + C$	`A*X^2+B*X+C;`
$\frac{1}{X-1}$	`1/(X-1);`

計算の順序は括弧も含めて普通の数式の計算と同じである. ただし数学ではかっことして, `[,]`,`{,}` などがつかえるが asir では `(,)` のみ. `[,]` や `{,}` は別の意味をもつ. 上の例のように `(,)` を何重にもつかってよい. この場合括弧の対応関係がわかりにくい. 括弧の対応を調べたい範囲をマウスでドラッグして選択し,



ボタンをおすことにより括弧の対応を調べることができる. 図 1.2 の例では `(1+2*(3+4))` と書くべきところを `(1+2*(3+4)` と書いておりエラーが表示されている.

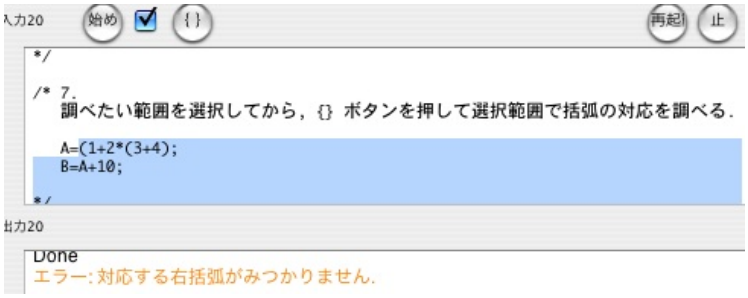


図 1.2: 括弧の対応

質問 “Basic 風の使い方を説明する” と書いてありましたが, Basic って何ですか?

答え コンピュータに仕事をさせるには最終的にはプログラム言語 (計算機への仕事の手順を指示するための人工言語) を用いる．ワープロ等もプログラム言語で記述されている．Basic は最も古いプログラム言語の一つであり, 初心者にやさしく, かつ計算機の仕組みやプログラム言語の理解にも有用である．Basic は高校の数学の教科書等にも登場する．著者はいままで “10 進 BASIC” を初心者向け教材として活用していたが, “10 進 BASIC” が MacOS X で動作しないため cfep を開発した．Asir 言語もプログラム言語であり Basic とよく似ているが, C 言語にもっと近い．

質問 MacOS X って何ですか?

答え ——まだ書いてない．

Asir は数の処理のみならず, $\sqrt{x}$ や三角関数の近似計算, 多項式の計算もできる．左の数学的な式は asir では右のように表す．

π (円周率)	`@pi`
$\cos x$	`cos(x)`
$\sin x$	`sin(x)`
$\tan x$	`tan(x)`
$\sqrt{x}$	`x^(1/2)`

三角関数の角度にあたる部分の x はラジアンという単位を用いて表す．高校低学年の数学では角度を度 (degree) という単位を用いて表すが, 数学 3 以上では角度はラジアンという単位で表す．

> 90 度 (直角) が $\pi/2$ ラジアン, 180 度が π ラジアン．一般に d 度は $\frac{d}{180}\pi$ ラジアンである．

単位ラジアンをもちいると微分法の公式が簡潔になる．たとえば x がラジアンであると $\sin x$ の微分は $\cos x$ である．

$\sin(x)$ や $\cos(x)$ の近似値を求めるにはたとえば

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

と入力する．これは sin(3.14) の近似値を計算する．$\sin\pi = 0$ なので 0 に近い値が出力されるはずである．実際 `0.00159265` を出力する．`deval` (evaluate and get a result in double number precision の略) は 64 bit の浮動小数点数により近似値計算する．64 bit の浮動小数点数とは何かの説明は超入門の範囲外であるが, 計算機は有限の記憶領域 (メモリ) しか持たないので, 小数も有限桁しか扱えないと覚えておこう．64bit は扱える桁数を表している．詳しくは “asir ドリル” を参照して欲しい．

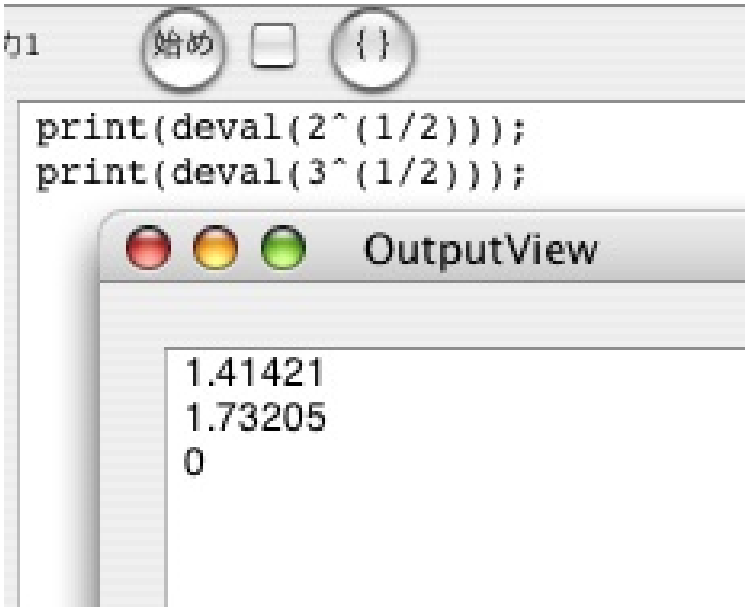


図 1.3: 平方根の計算

例題 1.2 $\sqrt{2}, \sqrt{3}$ の近似値を計算しなさい.
入力

```
print(deval(2^(1/2)));
print(deval(3^(1/2)));
```

出力は図 1.3 をみよ.

上の例のように, セミコロン ; で区切られた一連の命令のあつまりはもっとも単純な asir プログラムの例である. 一連の命令は始めから順番に実行される. `print(式等);` は “式等” の値を計算して値を画面に表示する.

さて出力の `1.41421` (ひとよ ひとよに ひとみごろ) は $\sqrt{2}$ の近似値なので, `print(deval(2^(1/2)));` の実行結果である. さて出力の `1.73205` (ひとなみに おごれや) は $\sqrt{3}$ の近似値なので, `print(deval(3^(1/2)));` の実行結果である. 最後の `0` はなんなのであろうか? 実はこれは最後の `print` 文の戻している値である. むつかしい? 別の例で説明しよう.
入力

```
1+2;
2+3;
3+4;
```

この時出力は (OutputView への表示は)

```
7
```

となる. cfep/asir ではとくに `print` 文をかかない限り最後の文の計算結果 (評価結果) しか出力しない. いまの場合は $3+4$ の結果 7 を出力している.

問題 1.1 1. 2^8, 2^9, 2^{10}, の値を計算して答えを表示するプログラムを書きなさい.

2. 2 の累乗はパソコンの性能説明によく登場する. たとえば検索システム google にキーワード “512 メモリ 搭載” を入力したところ “ビデオメモリを 256M から 512M に倍増させ” など, 数多くの記事がヒットする. このような記事を (意味がわからなくても)10 件あつめてみよう. 512 以外の 2 の累乗でも同じことを試してみよう.

3. (中級) 2 の累乗がパソコンの性能説明によく登場する理由を論じなさい.

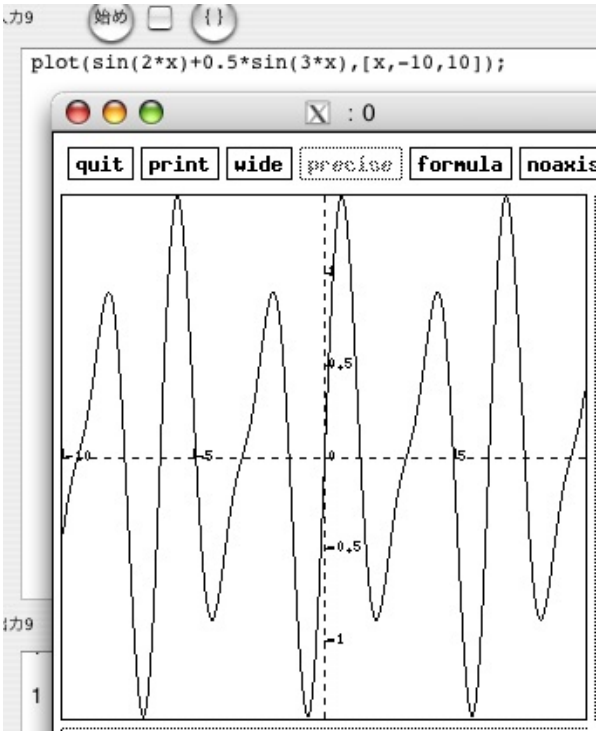


図 1.4: 関数のグラフ

発展学習 X11 環境が動いていれば, `plot(f);` 命令で x の関数 f のグラフを描ける. `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.
入力例

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

問題 1.2 いろいろな関数のグラフを描いてあそんでみよう. 数学の知識を総動員して計算機の描く形がどうしてそうなのか説明を試みてみよう.

1.1.3 エラーメッセージ

入力にエラーがあると, エラーメッセージが表示される.

図 1.5: 文法エラー

図 1.5 では `2+4=` と入力している. 最後に = を書く表現は asir の文法では許されていないので, “文法エラー” と指摘されている.

> 大体これでわかってくれていいじゃない, とこちらがおもっていてもプログラム言語は一切融通がきかない.

なお

```
error([41,4294967295,parse error,[asir_where,[[toplevel,1]]]])
```

の部分は上級者向けの情報なのでとりあえず無視してもらいたい.

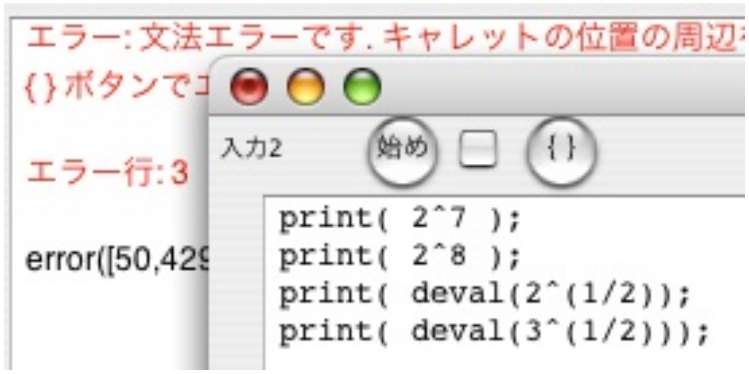


図 1.6: エラー行

図 1.6 では

```
print( 2^7 );
print( 2^8 );
print( deval(2^(1/2));
print( deval(3^(1/2)));
```

と入力している. 3 行目は右括弧がひとつ足りなくて `print( deval(2^(1/2));` が正しい入力である. エラー行の 3 行目にキャレットが自動的に移動しているはずである. なおプログラムの入力ウイ

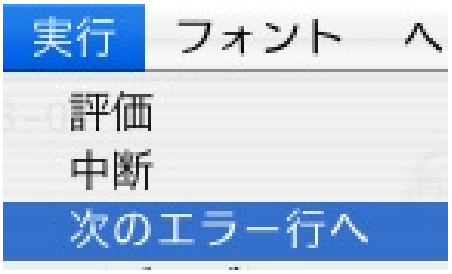
ンドー内でマウスをクリックすると, せっかく自動移動したキャレットの位置が変ってしまう. プログラムの入力ウインドーのタイトルバーでクリックするとよい. なおこの例では

ボタンをもちいてもすぐエラーの場所がわかる.
注意: 表示された行はエラーの発生位置であるが, エラーの原因はその前の方の行にあることも多い. たとえば

```
1+2
2+3;
```

と入力するとエラー行は 2 行目であるが, 原因は 1 行目で ； を書き忘れたことである.

エラー行が複数表示された場合はそれらの中のどこかにエラーがある. 複数あるエラー行に順番にジャンプしていくには, 実行 メニューから 次のエラー行へ を選択する.



問題 1.3 エラーを生じる式またはプログラムを 5 つ作れ.

1.2 変数とプログラム

1.2.1 変数

変数に数値等を記憶しておける. 変数名は大文字で始まる. なお後述するように 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利用される.

2 の累乗を表示する次のプログラムを考えよう.

```
print( 2^1 );
print( 2^2 );
print( 2^3 );
print( 2^4 );
print( 2^5 );
print( 2^6 );
print( 2^7 );
print( 2^8 );
```

このプログラムは変数 X を用いて次のように書いておけば 2 の累乗だけなく 3 の累乗を表示するのに再利用できる (図 1.7).

```
X = 2;
print( X^1 );
print( X^2 );
print( X^3 );
print( X^4 );
print( X^5 );
print( X^6 );
print( X^7 );
print( X^8 );
```

3 の累乗を表示するには X=2 の行を X=3 に変更すればいいだけである.

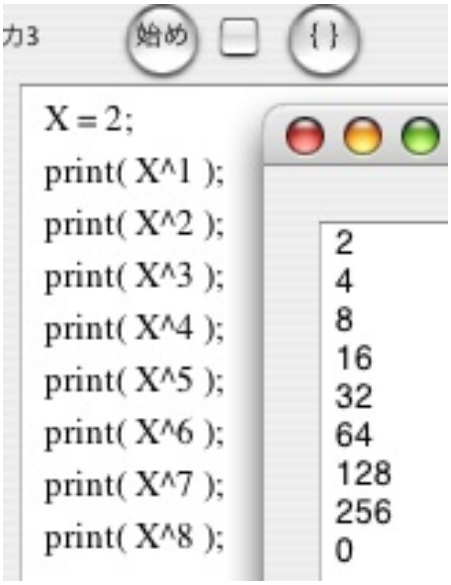


図 1.7: 変数の利用

アルファベットの 大文字 ではじまる英数字の列が asir の変数である. つまり, X, Y, Z はもちろんのこと, Sum とか Kazu とか X1 など 2 文字以上の英数字の列の組み合わせが変数名として許される.

変数を含んだ式をプログラム中で自由につかうこともできる. たとえば

```
X = 2;
A = 1;
print( 2*X^2 -A );
```

を実行すると 7 が表示される.

このような例をみると, 変数の機能は中学数学でならう文字式と似ていると思うだろう. 超入門としてはこれでほぼ正しい理解であるが, よりステップアップしていくには, 次のことを強く記憶しておこう.

変数とは計算機に数値等を保存しておくメモリ上の場所の名前である.

さて, 超入門, 第一の関門である.

= 記号は次のような形式でつかう:

変数名 =**式**;

これはまず右辺の式を計算しそのあとその計算結果を左辺の変数に代入せよという意味. = 記号は右辺を計算してその結果を左辺へ代入せよという 命令 だと思って欲しい.
たとえば, `X=1` は `X` が `1` に等しいという意味ではなく, `1` を 変数 `X` に代入せよという意味である.

ここでいいたいことは,

= 記号の意味が数学での意味と違うよ!

ということである. これで混乱する入門者も多いのでプログラム言語によっては “2 を変数 `X` に代入せよ” を `X:=2` と書く場合もある (たとえばプログラム言語 Pascal).

次のプログラムは 2, 2^2, 2^4, 2^8 を計算して表示する.

```
X=2;
print(X);
X = X*X;
print(X);
X = X*X;
print(X);
X = X*X;
print(X);
```

出力が図1.8のようになる理由を説明しよう. まず1行目で変数 `X` に2が代入される. 次に3行目ではま

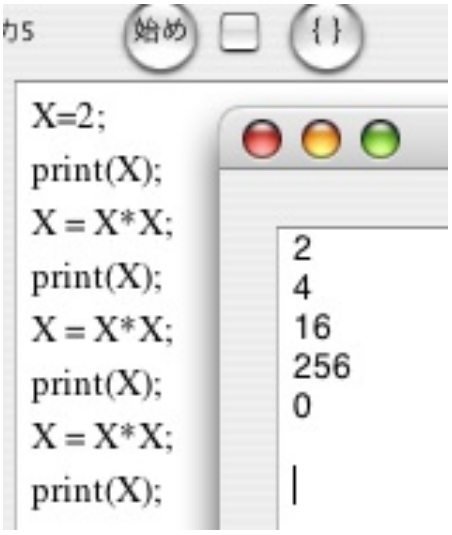


図 1.8: 変数の利用

ず右辺の式を計算する. この場合 `X` の値は 2 であるので, 2×2 で結果は 4 である. この計算が終った後結果の 4 が変数 `X` に代入される. 5 行目では右辺の式は 4×4 なので, その計算結果の 16 が 左辺の

変数 X に代入される.

発展学習 Asir は多項式計算もできる. 実は Asir は計算機で記号的に数式を処理するための数式処理システムでもある.

1. 小文字ではじまる記号は多項式の変数である. 数学とちがって変数の名前は一文字とはかぎらない. たとえば `rate` と書くと, $rate$ という名前の多項式の変数となる. 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. `fctr(` *poly* `)` は *poly* を有理数係数の範囲で因数分解する. `fctr` は factor の短縮表現である.

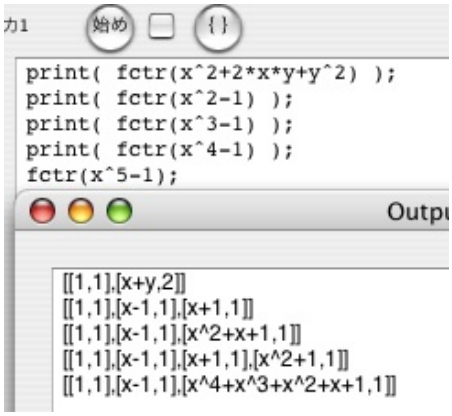


図 1.9: 因数分解

図 1.9 の `fctr` の出力の最初は $x^2+2xy+y^2$ が $1^1 \times (x+y)^2$ と因数分解されることを意味している. 図 1.9 の `fctr` の出力の 2 番目は x^2-1 が

$$1^1 \times (x-1)^1 \times (x+1)^1$$

と因数分解されることを意味している.

1.2.2 くりかえし

くりかえしや判断をおこなうための文が asir には用意されている. この文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の機能をためしてみよう.

例題 1.3 図 1.7 のプログラムは次のように繰り返し機能 — `for` 文 — を用いて簡潔に書ける.

```
X = 2;
for (I=1; I<=8; I++) {
   print( X^I );
}
```

実行結果は図 1.10 をみよ.

繰り返し関連の表現の意味を箇条書にしてまとめておこう.

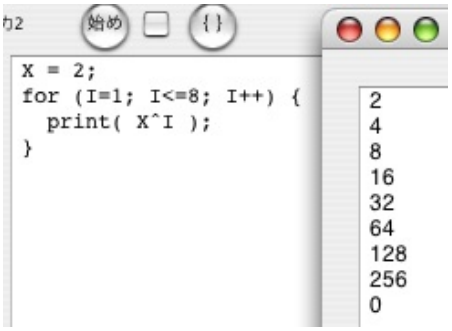


図 1.10: for 文

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.

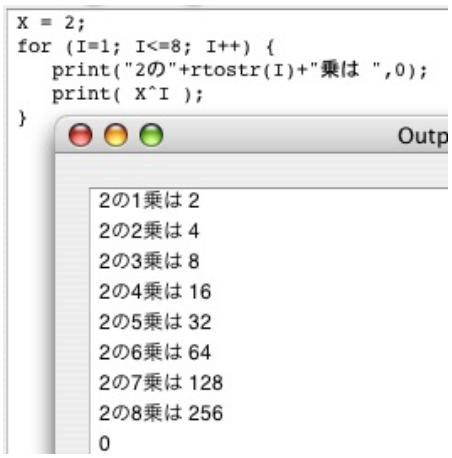


図 1.11: for 文

for のあとの {, } の中には複数の文 (命令) を書ける.

```
X = 2;
for (I=1; I<=8; I++) {
   print("2 の"+rtostr(I)+"乗は ",0);
   print( X^I );
}
```

この例では日本語を含むので前の節で述べたように日本語空白をプログラム本体にいれないようにして, 注意深くプログラムを入力してもらいたい. 実行結果は図 1.11 をみよ. print("2 の"+rtostr(I)+"乗は ",0); の部分を簡単に説明しておこう. まず最後の 0 は出力のあと改行しない, つまり次の

print 文の出力をそのまま続けよという意味. " でかこまれた部分は文字列と呼ばれている. これはこのまま表示される. rtostr(I) は数字 I を文字列表現に変換しなさい, という意味 (超入門としては難しい?). あと文字列に対して + を適用すると文字列が結合される.

雑談 (江戸時代の数学の本にあった問題の改題)

殿様: このたびの働きはあっぱれであった. 褒美はなにがよいか?

家来: 今日は一円, 明日は 2 円, 明後日は 4 円と, 前日の 2 倍づつ, これを 4 週間続けてくださるだけで結構でございます.

殿様: なんともささやかな褒美じゃのう. よしよし.

さて, 家来はいくら褒賞金をもらえるだろう? これもまた 2 の累乗の計算である. Cfep/asir で計算してみよう.

例題 1.4 for による繰り返しを用いて $\sqrt{x}$ の数表をつくろう.

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

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, たとえば 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.

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 ("asir ドリル" を参照). とりあえず,

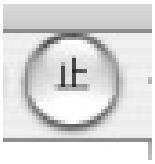
> 整数や分数の計算は Asir 上で正確に実行されるが, 小数についてはそうでない.

と覚えておこう.

問題 1.4 あたえられた *10* 進数を *2* 進数へ変換するプログラムを作れ. ヒント: A ÷ B の余りは A%B で計算できる.

1.2.3 実行の中止

実行中の計算やプログラムの実行を中止したい時は中止ボタン



をクリックする.

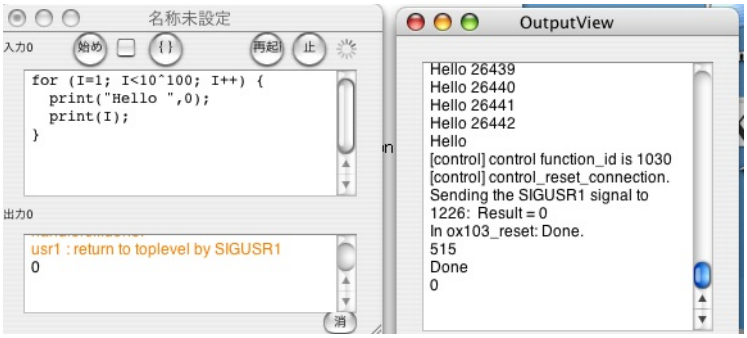


図 1.12: 実行の中止

図 1.12 では 10^{100} 回の `Hello ` の出力の繰り返しを中止している.
cfep は開発途上のシステムのため

```
[control] control function_id is 1030
[control] control_reset_connection.
Sending the SIGUSR1 signal to 1226:  Result = 0
In ox103_reset: Done.
515
Done
```

このような開発者専用のメッセージも出力されるが, とりあえずこのようなメッセージがでたら中止が成功したということである.

1.2.4 エンジン再起動

Cfep/asir では次のように 3 つのプロセスが互いに通信しながら動作している.

cfep ⇔ コントローラ (ox_texmacs) ⇔ 計算エンジン (ox_asir)

計算エンジン (計算サーバ) を再起動したり別のものにとりかえたりできる.
エンジン再起動ボタン

をクリックすると, 現在利用している計算エンジンを停止し, 新しい計算エンジンをスタートする. 選択範囲のみを実行するモードでないかぎり利用上で中止との違いはあまりないが, 再起動のときのメッセージ

にもあるように, 別の計算エンジンを起動することも可能である. この例では unix shell も起動できる.

また, “実行” メニューから “エンジンを自動スタートしない” モードを選んでる場合に計算エンジンを手動でスタートするには, このボタンを用いる.

発展学習 cfep は Cocoa FrontEnd view Process の略である. cfep は Objective C という言語および xcode 2 という開発環境を用いて Cocoa というフレームワークのもとで開発されている. cfep の Objective C のプログラムの一部をみてみよう.

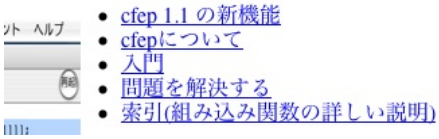
```
for (i=0; i<oglCommSize; i++) {
  gc = [oglComm objectAtIndex: i];
  [self execute: gc];
}
```

asir と同じような `for` 文があるね.

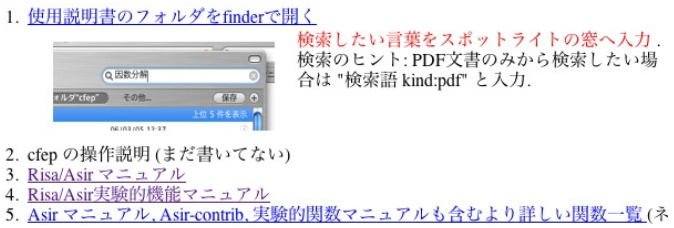
1.2.5 ヘルプの利用

Cfep/asir での “関数” とは数学の関数のように引数を与えると計算して値をもどし, かつある仕事 (表示等) をする手続きの集まりである. 例えば `print`, `deval`, `sin`, `fctr` 等は関数である. 関数を自分で定義することも可能である. これについては後の説明および “asir ドリル” を参照.

あらかじめ定義ずみの関数を “組み込み関数” とよぶ. 組み込み関数の詳しい説明を調べるには “cfep のヘルプ” から

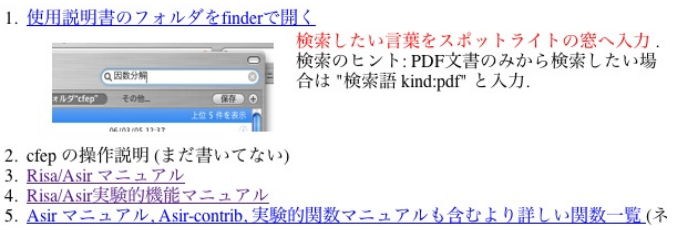


の “索引” を選び, 索引



の “Risa/Asir マニュアル” を選び, “Risa/Asir マニュアル” の最初のページの関数一覧から調べたい関数を探す. たとえば `fctr` (因数分解用の関数) はこの一覧の中にある.

検索には spotlight の活用も有益であろう. 索引



の “使用説明書のフォルダを finder で開く” を選ぶと使用説明書のフォルダが開くので, ここを spotlight で検索するといろいろな発見があるであろう. ちなみに, この超入門や asir ドリルはこのフォルダの pdf フォルダの中にある. (なおここからの spotlight 検索は何故か遅いので, メニューバーの splotlight からの検索の方がいいかもしれない.)

1.3 グラフィック

1.3.1 ライブラリの読み込み

Asir 言語で書かれている関数定義の集合がライブラリである. ライブラリを読み込むには `import` コマンドまたは `load` コマンドを用いる. マニュアルに記述されている関数でライブラリの読み込みが前提となってるものも多い. たとえば, 線を引くコマンド `glib_line(0,0,100,100);` を実行しても, “glib_line が定義されていません” というエラーが表示される. グラフィックコマンドのライブラリ読み込むコマンド

```
import("glib3.rr");
```

を実行しておくと図 1.13 のように線を描画する.

Asir-contrib プロジェクトにより集積されたライブラリの集合体が asir-contrib である. Asir-contrib を読み込んでしまうと, ほとんどの関数について import が必要かどうか気にする必要はなくなるが, 大量のライブラリを読み込むために時間がかかるのが欠点である. asir-contrib は 実行 メニューから読み込める.

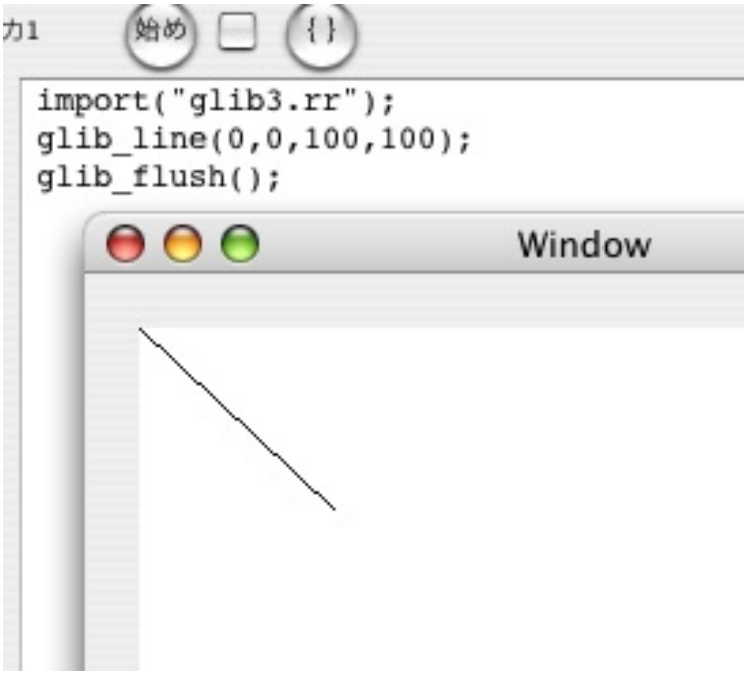


図 1.13: ライブラリのロード

1.3.2 線を引く関数

例題 1.5

```
import("glib3.rr");
glib_line(0,0, 100,100);
glib_flush();
```

図 1.13 が描画結果である．y 座標は画面が下へいくほど大きくなる．図 3.1 を参照．左上の座標は $(0,0)$, 右下の座標が $(400,400)$. `glib_line` で $(0,0)$ から $(100,100)$ へ線を描画. `glib_flush` は画面を更新するはたらきがある．flush しないと, 描画結果が画面での表示に反映しない場合がある．

glib3.rr 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. 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. ただし x 座標は, 右にいくに従いおおきくなり, y 座標は 下に いくに従い大きくなる (図 3.1).
glib_clear()	全ての OpenGL オブジェクトを消去し, 描画画面をクリアする.
glib_putpixel(X,Y)	座標 (X,Y) に点を打つ.
glib_set_pixel_size(S)	点の大きさの指定. 1.0 が 1 ピクセル分の大きさ.
glib_line(X,Y,P,Q)	座標 (X,Y) から 座標 (P,Q) へ直線を引く
glib_remove_last()	一つ前の OpenGL オブジェクトを消す.

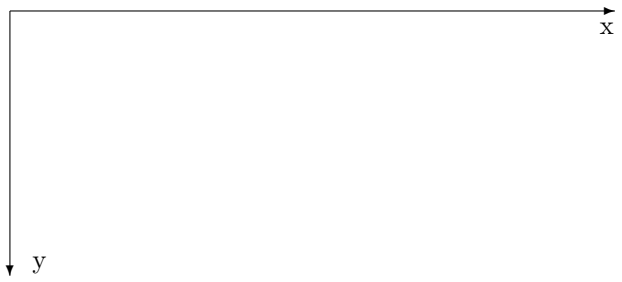


図 1.14: 座標系

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. 16 進数については “asir ドリル” を参照. 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. 16 進数を知らない人用に, 色とその 16 進数による表現の対応表をあげておく.

0xffffff	白
0xffff00	黄
0xff0000	赤
0x00ff00	緑
0x0000ff	青
0x000000	黒

(あとは試して下さい)

さて, 図 1.14 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, glib3.rr 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.

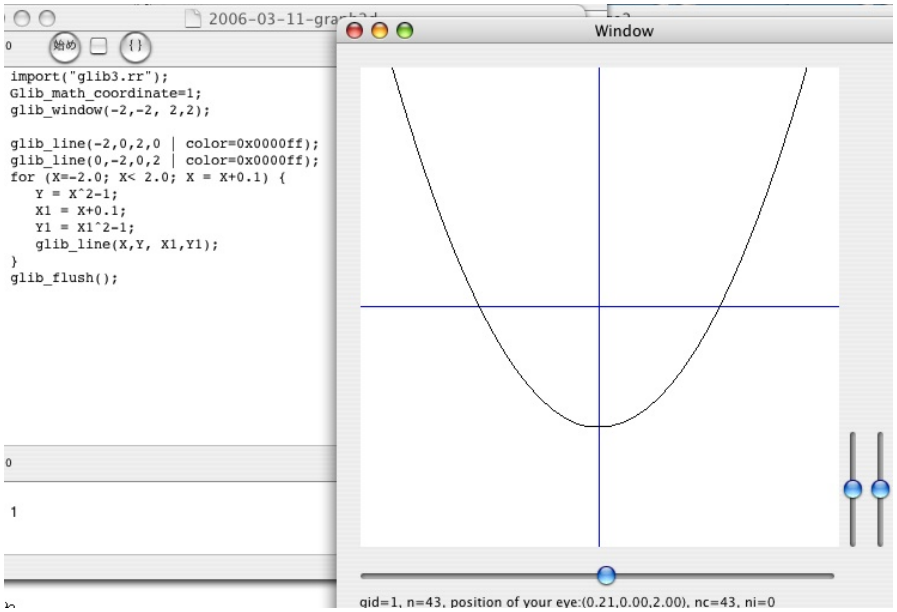


図 1.15: 2 次関数のグラフ

例題 1.6 2 次関数 $y = x^2 - 1$ のグラフを書いてみよう.

```
import("glib3.rr");
Glib_math_coordinate=1;
glib_window(-2,-2, 2,2);

glib_line(-2,0,2,0 | color=0x0000ff);
glib_line(0,-2,0,2 | color=0x0000ff);
for (X=-2.0; X< 2.0; X = X+0.1) {
   Y = X^2-1;
   X1 = X+0.1;
   Y1 = X1^2-1;
   glib_line(X,Y, X1,Y1);
}
glib_flush();
```

実行結果は図 1.15. ——プログラムの解説はまだ書いてない.

1.3.3 円を描く関数を作ってみよう

```
import("glib3.rr");
Glib_math_coordinate=1;
glib_window(-1,-1,1,1);
glib_clear();
E = 0.2;  X = 0; Y = 0;  R = 0.5;
for (T=0; T<=deval(2*@pi); T = T+E) {
  Px = X+deval(R*cos(T));
  Py = Y+deval(R*sin(T));
  Qx = X+deval(R*cos(T+E));
  Qy = Y+deval(R*sin(T+E));
  glib_line(Px,Py,Qx,Qy);
  glib_flush();
}
```

——プログラムの解説はまだ書いてない.

上のプログラムでは cos, sin を用いて円を描いている. 中心, 半径を変更したり, 色を変更したりしながらたくさんの円を描くには, どのようにすればよいであろうか? “関数” を用いるとそれが容易にできる.

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. 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.

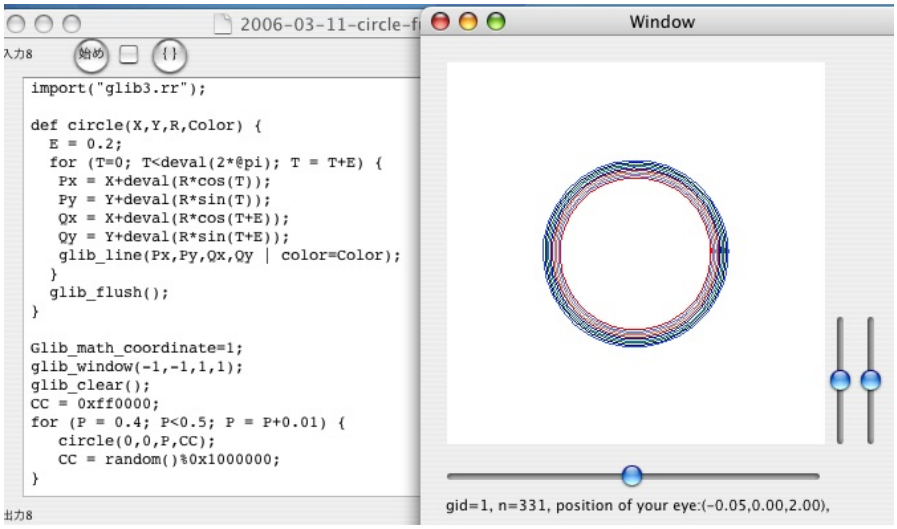


図 1.16: 関数による同心円の描画

さて円を描く例にもどろう. 以下のように関数 `circle(X,Y,R,Color)` を定義 (`def`) する. この関

数を R や $Color$ を変化させながら呼ぶことにより, 図 1.16 のような同心円の図を描くことが可能となる. 関数について詳しくは “asir ドリル” を参照してほしい.

```
import("glib3.rr");

def circle(X,Y,R,Color) {
  E = 0.2;
  for (T=0; T<deval(2*@pi); T = T+E) {
   Px = X+deval(R*cos(T));
   Py = Y+deval(R*sin(T));
   Qx = X+deval(R*cos(T+E));
   Qy = Y+deval(R*sin(T+E));
   glib_line(Px,Py,Qx,Qy | color=Color);
  }
  glib_flush();
}

Glib_math_coordinate=1;
glib_window(-1,-1,1,1);
glib_clear();
CC = 0xff0000;
for (P = 0.4; P<0.5; P = P+0.01) {
   circle(0,0,P,CC);
   CC = random()%0x1000000;
}
```

——プログラムの詳しい解説まだ.

問題 1.5 1. 分度器を描くプログラムを作れ.

2. (発展課題) この分度器, 糸, おもり, わりばし, 板, cfep/asir によるプログラム等を用いて, 木やビルの高さを測定する機械とソフトウエアシステムを開発せよ.

問題 1.6 (これは発展課題) cfep には OpenGL インタプリターが組み込んである. OpenGL は 3 次元グラフィックスを用いるソフトウエア作成のために用いられる約 150 種類のコマンドから構成されているパッケージで 3 次元グラフィックスの標準規格のひとつでもある. cfep 1.1 ではその中の 10 弱のコマンドを利用できる.

この OpenGL インタプリターを用い, 多面体 (polygon) を材料にし, cfep 上級編, OpenGL のプログラムを参考に “家” を書いてみよう.

1.4 For 文による数列の計算

1.4.1 超入門, 第 2 の関門: 漸化式できまる数列の計算

例題 1.7 a を正の数とするとき,

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2},$$

$$x_0 \quad = \quad a$$

できまる数列 $x_0, x_1, x_2, \ldots$ は $\sqrt{a}$ にどんどん近付くこと (収束すること) が知られている. $a = 2$ の時, $x_1, x_2, \ldots, x_4, x_5$ を計算するプログラムを書いてみよう.

```
A = 2.0;
X = A;
for (I=0; I<5; I++) {
  Y = (X+A/X)/2;
  print(Y);
  X = Y;
}
```

このプログラムの実行結果は図 1.17.

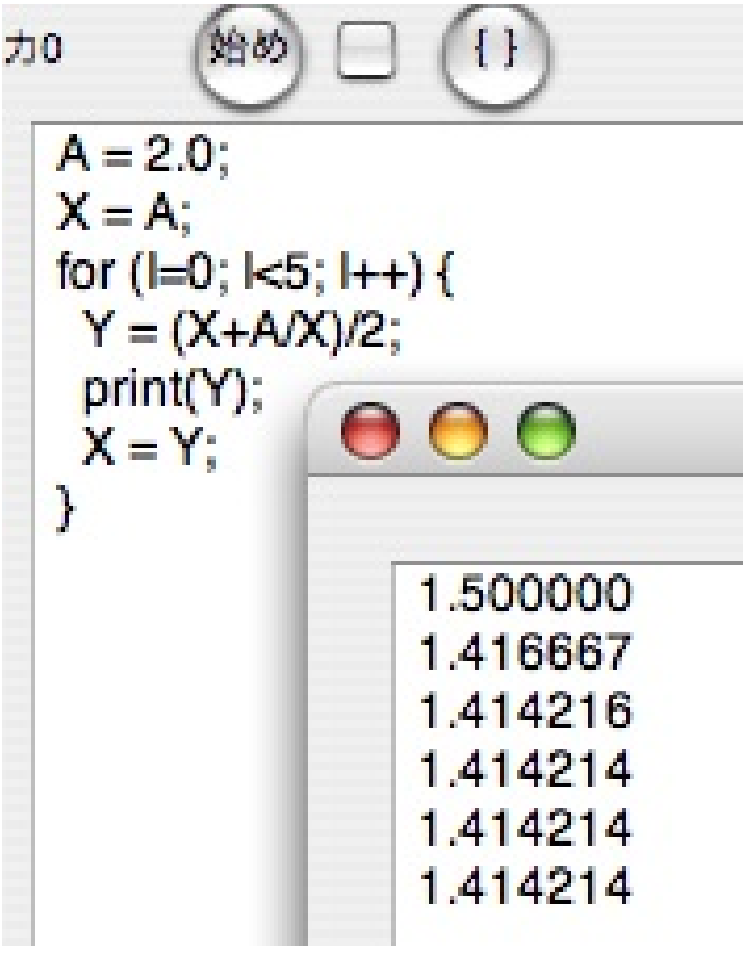


図 1.17: $\sqrt{2}$ に収束する数列

超入門での関門は

```
Y = (X+A/X)/2;
X = Y;
```

の意味を完全に理解すること

である. 変数の章で説明したように,

変数名=**式**;

はまず右辺の式を計算しそのあとその計算結果を左辺の変数に代入せよという意味である. したがって, `Y = (X+A/X)/2;` は現在の `X` と `A` に格納された数字をもとに `(X+A/X)/2` の値を計算し, その結果を変数 `Y` へ代入せよ, という意味である. また

`X=Y` は `X` が `Y` に等しいという意味ではなく, 変数 `Y` に格納された数字を 変数 `X` に代入せよという意味である.

このように考えれば, 上のプログラムが x_1, x_2, x_3, x_4 の値を順番に計算して print している理由が理解できるであろう. 自分が計算機になったつもりで, 変数の中の数値がどのように変化していくのか, 書きながら理解して頂きたい. これがはっきり理解でき, 応用問題が自由に解けるようになった, 超入門卒業である.

問題 **1.7** 変数 `I`, `X`, `Y` の値は `for` ループ内でどのように変化するか? `Y= (X+A/X)/2` の行が実行される前のこれらの変数の値を表にしてまとめよ. `print([I,X,Y])` をはさむことによりこの表が正しいことをたしかめよ.

問題 **1.8** プログラムのバグ (bug) とはなにか?

1.4.2 円を描く数列

数列の計算を用いると, cos や sin の計算をやらずに円を描くことができる.

```
import("glib3.rr");
Glib_math_coordinate=1;
glib_window(-2,-2, 2,2);
glib_clear();
E = 0.1;
C1 = 1.0; C2=1.0;
S1 = 0.0; S2=E;
for (T=0; T<=deval(2*@pi); T = T+E) {
    C3 = 2*C2-C1-E*E*C2;
    S3 = 2*S2-S1-E*E*S2;
    glib_line(C1,S1, C2,S2);
    C1=C2; S1=S2;
    C2=C3; S2=S3;
    glib_flush();
}
```

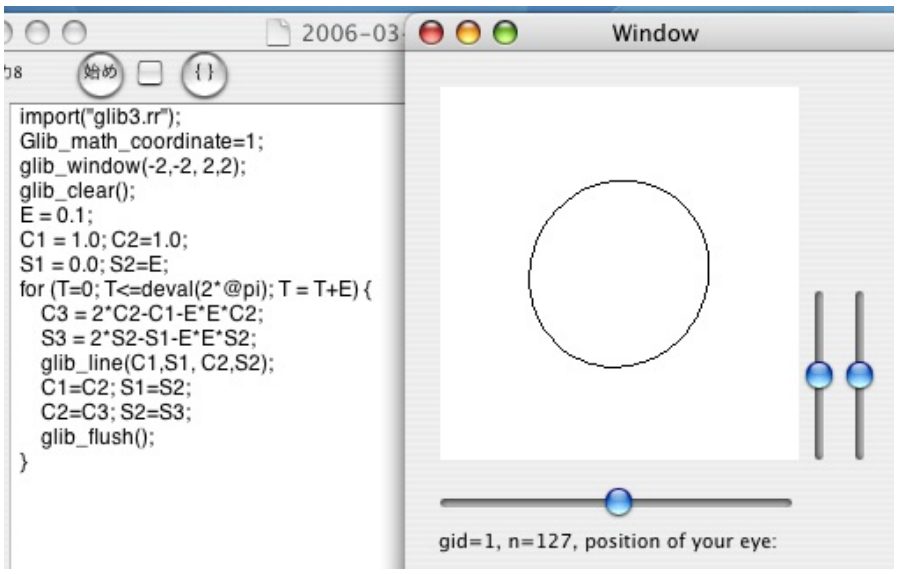


図 1.18: cos, sin を使わずに円を描く

このプログラムの実行結果は図 1.18.

——プログラムの解説まだ書いてない.

ヒント: 微分方程式 $d^2x/dt^2 = -x, d^2y/dt^2 = -y$ を t をラジアンとして差分法で近似的に解いている.

この話題は, 数列の計算と差分方程式によるシミュレーションに続く. これについてはまた稿をあらためて書いてみたい.

以上で超入門は終了である. 続きは “Asir ドリル” を読んでね. 特に配列と関数をマスターすると数学プログラムには重宝する.

なお, asir ドリルに紹介してあるプログラムは `end$` または `end;` が最後に書いてある場合が多いが, cfep/asir ではこの `end` を書いてはいけない. `end` は計算エンジンの停止命令であり, 実行されると “計算中の表示” がでて無応答となる. (一応, 行頭にあるこれらの命令は自動的に削除するようになってはいる.)

問題 1.9 (レポート問題の例)
なにか図を描くプログラムを書きなさい. (定番ドラエモンでもよい)

1.5 cfep 上級編

1.5.1 TEX によるタイプセット (実験的)

出力を TeX でタイプセットするには “実行” メニューから “出力を TeX でタイプセット” を選択する. `latex`, `dvipng` がインストールされていないと動作しない. これらはたとえば `fink` から TEX をインストールしたり, `ptex_package_2005v2.1.dmg` などで Mac 用の pTeX をインストールしておけばよい. TEX を用いた仕上り例は図 1.19 を見よ. なお, TEX でタイプセットする場合ホームの下に `OpenXM_tmp` なる作業用のフォルダが作成される. タイプセットは実験機能のため, このフォルダの中の作業用ファイルは自動では消去されない. 時々手動で作業ファイルを消去されたい.

1.5.2 選択範囲のみの実行

画面上の “選択範囲のみを実行” をチェックすると, “始め” ボタンをおしたとき, 選択範囲のみが評価される. 選択範囲がない場合はキャレット位置の行が自動選択されて実行される. ALT+Enter と組み合わせてこの機能を使うと, ターミナルから asir を利用するのにちょっと似てくる. 図 1.19 はこのような実行をしている例である.

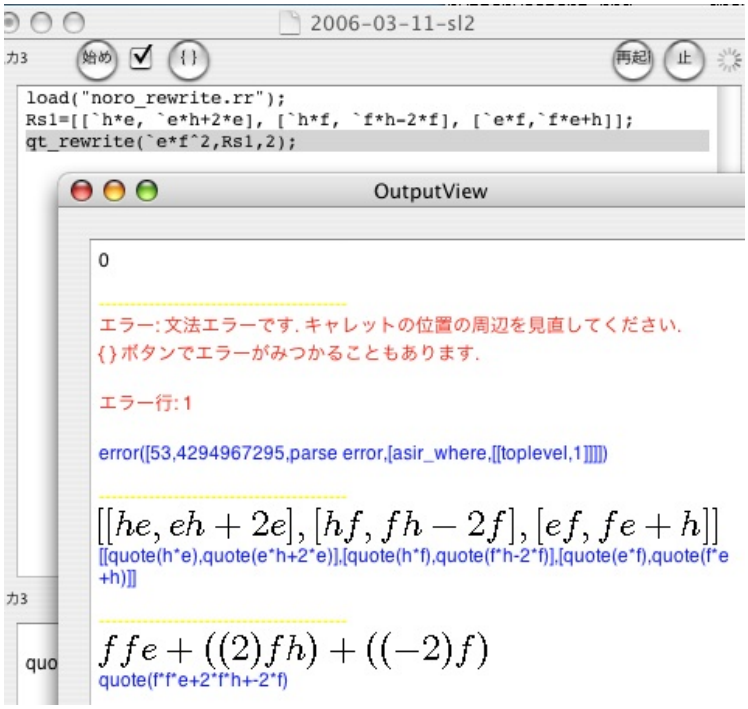


図 1.19: ターミナル風

質問 cfep のインタフェースでデバッグをしながらプログラムを開発するにはどのようにやるとよいか?

答え cfep は初心者向きのインタフェースなので, 大規模なプログラム開発を想定していないが, 私は次のようにライブラリの開発をしている.

1. 必要な関数を書く. 下の例では `sum1`.
2. 関数をテストする入力をコメントの形でその関数の近くに書いておく. 下の例ではコメントにある `sum1(10,1);` 等.

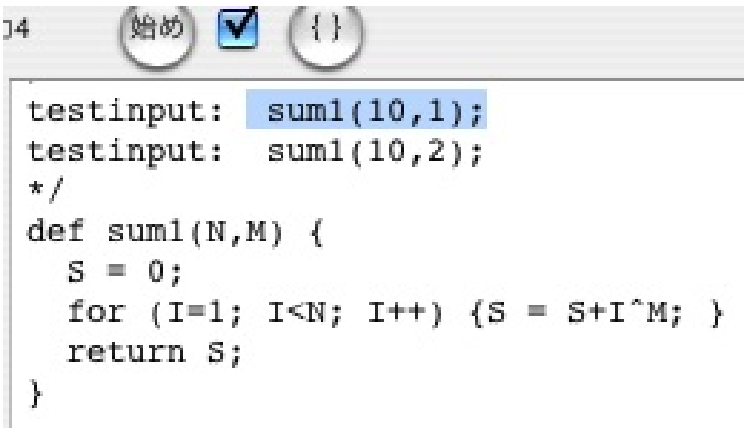


図 1.20: “選択範囲のみを実行” の活用

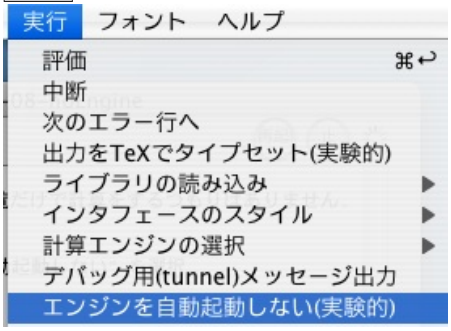
```
/*
testinput:  sum1(10,1);
testinput:  sum1(10,2);
*/
def sum1(N,M) {
  S = 0; i=1;
  for (I=1; I<N; I++) {S = S+I^M; }
  return S;
}
```

1. “始め” ボタンで関数定義をロード. この時点で文法エラーなどがあればメッセージにしたがって修正.
2. そのあと “選択範囲のみを実行” のモードに変更してコメント内の testinput を実行.
3. 実行時のエラーの行番号への移動は ”選択範囲のみを実行” のモードを解除してから行う.

1.5.3 エンジンを起動しない

質問 テキスト編集またはテキストの閲覧だけで計算をするつもりはありませんが.

答え “実行” メニューで “エンジンを自動起動しない” を選択.



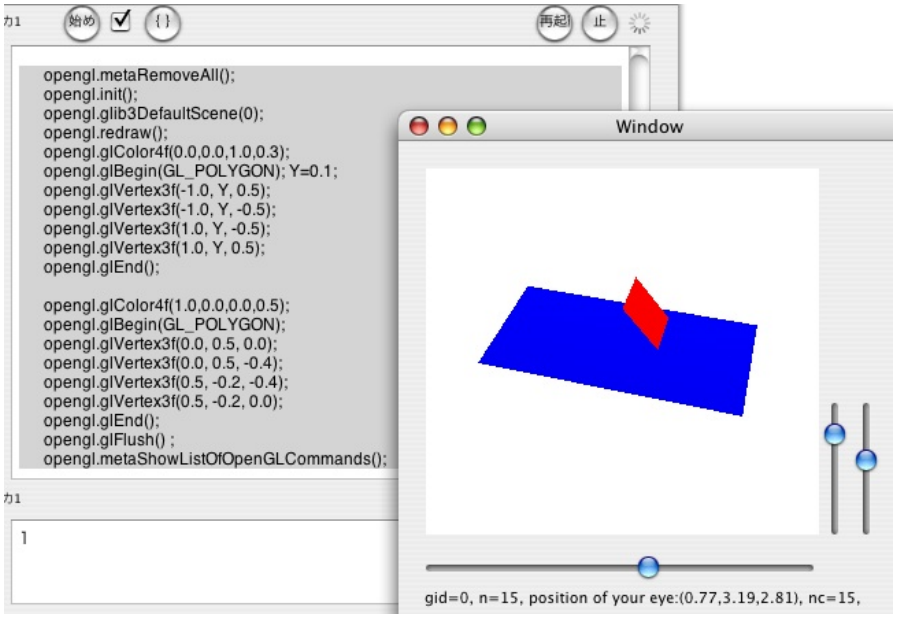


図 1.21:

あとでエンジンを起動したい場合は “再起” ボタンをおしてエンジンを起動する.

1.5.4 OpenGL インタプリタ

Cfep には OpenGL インタプリターが組み込んである. OpenGL は 3 次元グラフィックスを用いるソフトウエア作成のために用いられる約 150 種類のコマンドから構成されているパッケージで 3 次元グラフィックスの標準規格のひとつでもある. cfep 1.1 ではその中の 10 弱のコマンドを利用できる. 詳しくは `cfep.app/OpenXM/lib/asir-contrib/cfep-opengl.rr` を参照.

OpenGL ではまず OpenGL グラフィックオブジェクトを配置し, それから視点の位置から見た画像を描画する方法を用いる. したがって, システムは常に OpenGL グラフィックオブジェクトの集合を保持している. `glib_remove_last()` 命令はその最後の要素を削除する命令である. `cfep-opengl.rr` ライブラリでは, `opengl.metaRemoveLast()` 関数で最後の要素を削除できる.

```
import("cfep-opengl.rr");
opengl.metaRemoveAll();
opengl.init();
opengl.glib3DefaultScene(0);
opengl.redraw();
opengl.glColor4f(0.0,0.0,1.0,0.3);
opengl.glBegin(GL_POLYGON); Y=0.1;
opengl.glVertex3f(-1.0, Y, 0.5);
opengl.glVertex3f(-1.0, Y, -0.5);
opengl.glVertex3f(1.0, Y, -0.5);
opengl.glVertex3f(1.0, Y, 0.5);
opengl.glEnd();

opengl.glColor4f(1.0,0.0,0.0,0.5);
opengl.glBegin(GL_POLYGON);
opengl.glVertex3f(0.0, 0.5, 0.0);
opengl.glVertex3f(0.0, 0.5, -0.4);
opengl.glVertex3f(0.5, -0.2, -0.4);
opengl.glVertex3f(0.5, -0.2, 0.0);
opengl.glEnd();
opengl.glFlush() ;
opengl.metaShowListOfOpenGLCommands();
```

このプログラムでは 2 枚の長方形を描いている. このプログラムの出力は図 1.21. ——詳しい説明はまだ.

OpenGL の画面には普通の数学のように (x, y) 座標がはいっており, 画面から手前側が z 座標が正の方向, 画面の向こう側が z 座標が負の方向である. “目” から原点方向を見た画像が図 1.21 にあるように 3 つのスライダーを用いて目の位置を動かせるので, OpenGL オブジェクトをいろいろな角度からみることが可能である. 下のスライダーが目の x 座標, 右の二つのスライダーがそれぞれ目の y, z 座標である. 目の動きに慣れるには, 次の二つのデモ画面をためすと面白いだろう.

```
import("cfep-opengl.rr");
opengl.glib3DefaultScene("mesa demo/ray");
```

```
import("cfep-opengl.rr");
opengl.glib3DefaultScene("cfep demo/icosahedron");
```

1.5.5 asir 以外の計算エンジンの利用

cfep から asir 以外の OpenXM 準拠の計算エンジンも利用できます. たとえば, 実行, 計算エンジンの選択で, kan/sm1 を利用することも可能です. cfep, `ox_texmacs`, `ox_sm1` が相互に通信しながら計算しています.

なお kan/sm1 の `run` コマンドは使えません.

`[(parse)（ファイル名） pushfile] extension`

で代用して下さい.

第2章　Risa/Asir 入門

2.1　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 2.1 [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 2.1

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

例題 2.2 [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 2.1 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 2.3 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. 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 2.1: 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 2.1 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.

2.2 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
```

2.3 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 6 章でくわしく説明する. 詳しいことをやる前に, この段階で局所変数と引数についてはある程度理解していると, プログラミングが楽になると思う. 6.2 を読むといいであろう.

2.4 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3章　制御構造

3.1　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 3.5 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.
```

3.2　プログラム例

例 3.1 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. Windows では “ファイル → 開く” を用いて読み込む.

注意 3.1 cfep/Asir を利用している場合, この例は次のように入力窓に入力し実行した方が簡単だろう.

```
def main(A,B) {
     print(A+B);
     print("A kakeru B=",0);
     print(A*B);
}
main(43,900)$
```

以下の例でも同様である. 実際に Mac で実習をやるとこの注意を聞いてくれない人が多い!
上のように読み替えるのをお忘れなく.

あと, asir ドリルに紹介してあるプログラムは `end$` または `end;` が最後に書いてある場合が多いが,
cfep/asir ではこの `end` を書いてはいけない. `end` は計算エンジンの停止命令であり, 実行されると "計算中の表示" がでて無応答となる. (一応, 行頭にあるこれらの命令は自動的に削除するようになってはいる.)

もちろん適当なテキストエディタ (emacs 等) で `cond1.rr` をたとえばホームディレクトリに作成しておいて,

```
load(getenv("HOME")+"/cond1.rr");
main(43,900)$
```

を入力窓に入力して実行してもよい.

例 3.2 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);
}
```

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, 6 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);
}
```

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 3.3 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);
}
```

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 3.1 [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 3.4 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 6 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);
}
```

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 3.5 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()$
```

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 (総当たり探索) 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 3.6 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);
}
```

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 3.3 節の解説を参照.

注意 3.2 cfep/asir を利用している場合は, 必ず load("glib"); を load("glib3.rr"); へ置き換えること．また glib_flush(); を main 関数の最後の行に加えること．

例 3.7 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);
  }
}
```

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 3.8 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]);
  }
}
```

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 3.9 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);")$
```

`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. なおプログラム中の `x` は多項式変数の小文字. 大文字で入力しないように.

注意 3.3 念のため cfep/asir 用の入力も記載しておく.

```
load("glib3.rr")$
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));
  glib_flush();
}
taylor(4);
```

例 3.10 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);")$
```

例 3.11 次に for の 2 重ループを作ってみよう.

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 (倍精度浮動小数) の精度で数値計算する. 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.

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("  >>>");
   }
}
```

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 3.12 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]);
          }
       }
    }
  }
}
main()$
```

実行例

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

$1 \leq x < 10, 1 \leq y < 10, 1 \leq z < 10$ 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 3.2 [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.

時間計測は, 次の命令で行うとよい.

```
T0 = time();
計測したいプログラム
T1 = time(); print(T1[0]-T0[0]);
```

問題 3.3 引数 N の階乗を返す関数を作れ.

問題 3.4 引数 N, I に対し, 2 項係数 $\binom{N}{I}$ を返す関数を作れ. .

問題 3.5 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 3.6 $a_1 = A$, $a_2 = B$, $a_{n+2} = Pa_{n+1} + Qa_n$ $(n \geq 1)$ で定義される数列の第 N 項を求める関数を作れ. 引数を P, Q, A, B, N とせよ.

問題 3.7

```
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.

3.3 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. 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. ただし x 座標は, 右にいくに従いおおきくなり, y 座標は 下に いくに従い大きくなる (図 3.1).
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 3.1: 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);
```

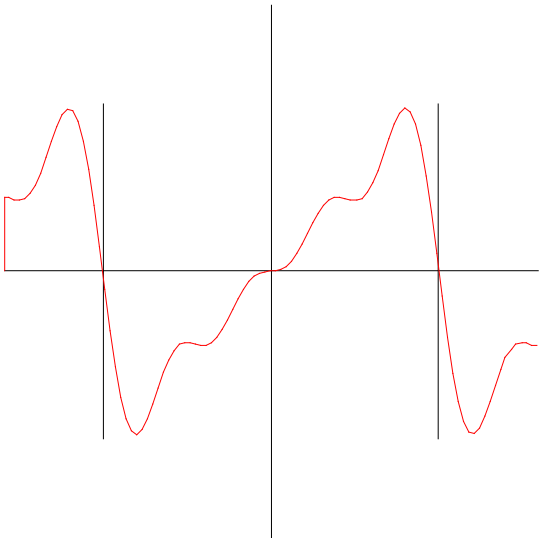
と入力すると, 色 0xff0000 で線分をひく. ここで, 色は RGB の各成分の強さを 2 桁の 16 進数で指定する. この例では, R 成分が ff なので, 赤の線をひくこととなる. なお, 関数 glib_putpixel も同じようにして, 色を指定できる.

さて, 図 3.1 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,

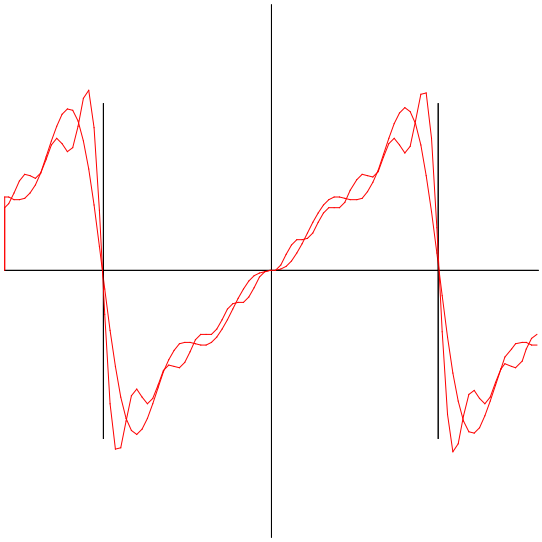
```
Glib_math_coordinate=1;
```

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4章　制御構造とやさしいアルゴリズム

4.1　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 4.1 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.

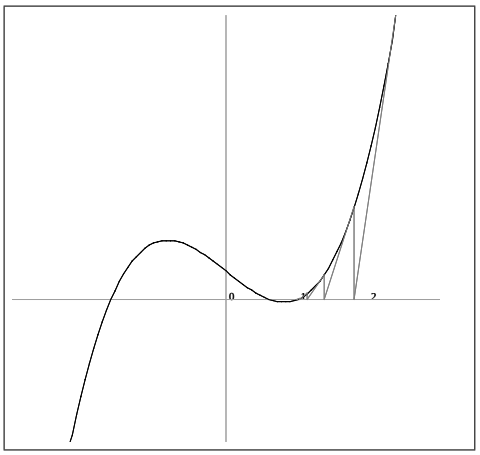


図 4.1: Newton 法が収束する例

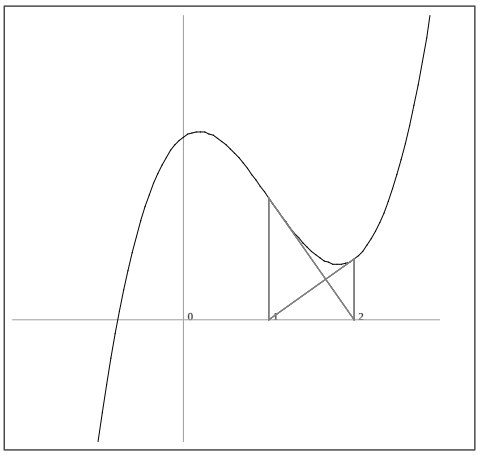


図 4.2: Newton 法が収束しない例

たとえば図 4.2 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);` (右辺を計算してから結果を左辺の変数 Q に代入)
の実行で Q に x_{k+1} の値がはいる. while ループの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. 次の `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 α 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 4.1 (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 α, 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, ε_{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 ξ_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,

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

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 ε_n なら $n+1$ ステップ目では誤差が ε_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 α は約 1.41421 である. このとき $\frac{f''(\alpha)}{2f'(\alpha)}$ の値は 0.707109 である. $|x_n - \alpha| < 10^{-p}$ なら, ほぼ $|x_{n+1} - \alpha| < 0.71 \times 10^{-2p}$ 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)$
2.0000000000000000000000000000000000000000
1.5000000000000000000000000000000000000000
1.4166666666666666666666666666666666666670
1.4142156862745098039215686274509803921 62
1.4142135623746899106262955788901349101 25
1.4142135623730950488016896235025302436 20
[432] eval(2^(1/2));
1.4142135623730950488016887242044430844 93
```

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.

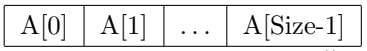
4.2　最大値と配列

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
```

4.3 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 4.2 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 4.3 $e(N) = N/D$ (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 4.4

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

と

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

(マチンの公式) を使って π の有理数近似値を求める関数を書け.

4.4 章末の問題

問題 4.5 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 10 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 4.6 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 4.7 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 4.8 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 4.9 *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 1:

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5章　ユークリッドの互除法とその計算量

5.1　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 5.1 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.

5.2　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 5.1 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.

5.3　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 5.2 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 5.2 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 5.3 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 5.4 m 桁の数の GCD の計算は, 互除法で $O(m)$ 時間でできる.

問題 5.1 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 5.2 $a = 1000000000000283$ 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. 関数 `pari(nextprime,10^12)` を用いると 10^{12} より大きい素数を一つ生成できる. この関数を用いて素因数分解が困難な数 a, b を作り, 上と同様なことを試みよ.

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.

5.4　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 (10^9Hz) 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.

$c \leftarrow 0$
for ($i = 0$; $i < n$; i++)
  $t \leftarrow a_i b + c$
  $m_i \leftarrow t \bmod 10$
  $c \leftarrow \lfloor t/10 \rfloor$
end for
$m_n \leftarrow c$

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++)
  $t \leftarrow f_i + g_i + c$
  if $t \geq 10$ $f_i \leftarrow t - 10$, $c \leftarrow 1$
  else $f_i \leftarrow t$, $c \leftarrow 0$
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.

5.5　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.

5.6　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6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.

6.1　リストとベクトル (配列)

ベクトルは, 4.2 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 9 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 2×2 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.

リストへは $A[0] = 1$ のような代入はできないが, ベクトル (配列) の場合はできる. リストは実行中にその長さを自由に変更できるのに対し, ベクトルでは実行中に長さの変更はできない.

6.2　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.

“戻り値”(return value) という言い方は計算機言語特有の言いまわしである. “関数 nsum は和を計算して結果を戻す” みたいに使う. 上の例でいえば A=nsum(10) としたとき, 戻り値が関数の値として変数 A に代入される.

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 6.1 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 6.1: 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 6.2 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 I)	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 I)	3
S(nsum の S)	6
N(nsum の N)	3

図 6.2: 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]
```

6.3 プログラム例

例題 6.1 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);
}
```

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 6.2 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 6.3 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();
```

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 6.4 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 6.5 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]);
}
```

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;
  }
}
```

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 6.1 (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 6.3 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 6.3: 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 6.2 (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?

6.4　デバッガ (より進んだ使い方)

2.2 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" (http://www.math.kobe-u.ac.jp/OpenXM にある) の第 5 章を参照してほしい.

6.4.1　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      }
(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) 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)
```

```
(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.

6.4.2　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.

これを実行すると止まらない.

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

```
[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   }
(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.

6.5 章末の問題

問題 6.3 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();
```

問題 6.4 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();
```

問題 6.5 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();
```

問題 6.6 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 6.7 [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 7 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 6.8 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 6.9 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 6.10 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.

フローチャート 雑感

プログラムの論理の流れを表す方法には様々なものがある. 人間は図での表現を好むようなので, 論理の流れを図示する方法がいろいろとある. 一番歴史が古いものでは, フローチャート (flow chart). これは `if` 文と `for` 文しか使わないアルゴリズムの表現や, 機械語等の論理の流れを記述するのに便利である. この本の前半にでてくる程度のアルゴリズムの記述にも最適であろう.

フローチャートでは構造化プログラミングには使えないというので, 出現したのが構造化フローチャート. たとえば PAD 等がある. この時代は “アルゴリズムとデータ構造” が掛け声であった. Asir でのプログラムの論理の流れを図示するにはちょうどいい.

1980 年代からオブジェクト指向プログラミングが出現したが, オブジェクト指向プログラミング用の図示の方法が UML である. 最近は “デザインパターン” が掛け声である.

筆者 (N) は徹底的な?数学の教育を受けてきているので, プログラムを書くときも, 短いプログラムを書いて実験. それから紙と鉛筆で, まず関係する定理, 命題を作り, 変数を作り, 必要と思われる関数を列挙し論理を整理. 時々, いまだにフローチャートでアルゴリズムを表現するときもある (なかなか便利) が, 基本は関数の仕様策定とデータ構造の仕様策定. オブジェクトの設計を紙に図示して考えるときも UML でなくオリジナル. 論理が怪しいと思えば, 数学的証明を試みる. などなど, 自己流である. これで何十万行のシステムがきちんと動くんだからいいや... である.

常にコンピュータスクリーンに向いプログラムを書いてる学生を見掛けることが多いが, “ちょっとまって. 紙と鉛筆をつかって論理を整理してみて” とか “テスト用のデータをなん通りか生成してプログラムを使わずに計算してみて” といっている.

第7章　入出力と文字列の処理, 文字コード

7.1　文字コード

7.1.1　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利用されている.

問題 7.1 (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 7.2 (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
```

7.1.2　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, という評価もあったが現状での圧倒的な普及度を考えて, JIS X0208 附属書 1 で シフト JIS コードを規定している. しかしながら 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. シフト JIS を漸進的に捨てて, Unicode ベースの UTF8 を利用しようというのが最近の趨勢であろう.

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.

日本語 EUC コードは古い unix 系のコンピュータでおもに使われていたコード系であり (現在では UTF8 が一般的), 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

JIS 漢字コードは 1 バイト目 21H から 7EH まで, 2 バイト目も 21H から 7EH までの範囲を動くコード系である. 漢字を利用できる以前のパーソナルコンピュータでは, 文字はすべて 1 バイトで表現しており, A1H から DFH の範囲にカタカナを割り当てて利用した. このシステムと互換性をたもちながら, 漢字を扱えるようにしたのが, シフト JIS 漢字コードである. JIS 漢字コードの範囲を 1 バイト目 81H から 9FH. 2 バイト目 40H から FCH の範囲, および 1 バイト目 E0H から EFH. 2 バイト目 40H から FCH の範囲, へある規則で写したコード系になっている. たとえば 厩 (3139H) はシフト JIS 漢字コードでは 8958H となる. 実際の変換規則などについては google で “シフト JIS” をキーワードにして調べてみるとよい.

UTF8 はアスキーコードはそのまま, 漢字コードなどは 2 バイト, または 3 バイトに変換して表現した形式である.

```
0xxxxxxx
110yyyyx   10xxxxxx
1110yyyy   10yxxxxx   10xxxxxx
```

最初の表現方法は 7 bit のデータを表現できる, 2 番目の表現方法は 11 bit のデータを表現できる, 3 番目の表現方法は 16 bit のデータを表現できる, 実際のコード系については google で “UTF8” をキーワードにして調べてみるとよい. たとえば 厩 (3139H) は UTF8 では E58EA9 (11100101 10001110 10101001) となる.

JIS コード, shift JIS コード, 日本語 EUC コード, UTF8 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 7.3 (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　゛　。
```

7.1.3 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 7.4 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.

7.2 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 7.4 節をみよ.

7.3　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 7.5 (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.

7.4　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. 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 ≤ 255) を 16 進表記に直す. 関数 ishift(A,N) は, A を 2 進表記を bit 列と見て, N が正の場合は右シフト, すなわち 2^N で割った商, N が負の場合は左シフト, すなわち 2^{-N} 倍を返す. 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("");
}
```

例題 7.1　ファイル dump.rr のダンプをとってみなさい.

入力例 7.1

```
[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.

なお新しい版の asir では fprintf や string_to_tb などの関数が導入され, 文字列とファイルの処理がより簡単にかつ効率よく行なえるようになっている. 詳しくはマニュアル “実験的仕様の関数” を参照してほしい.

7.5　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8章　再帰呼び出しとスタック

8.1　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 8.1 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. 6 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 8.1 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

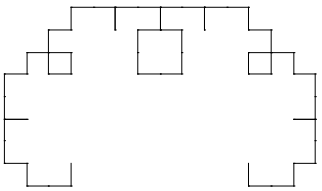
```
def fib(N) {
   if (N == 1) return(1);
   if (N == 2) return(1);
   return(fib(N-1)+fib(N-2));
}
```

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 8.2 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 8.1 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.)

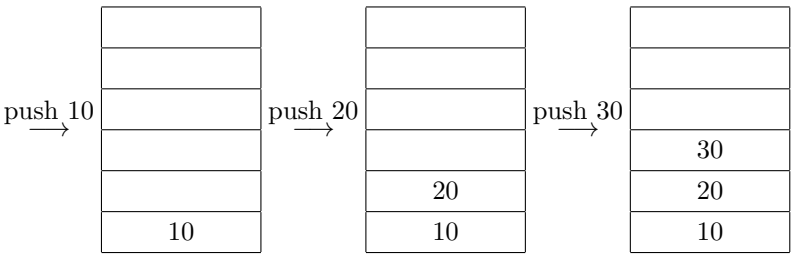
問題 8.2 次のような定規を描くプログラムを再帰を用いて書け.

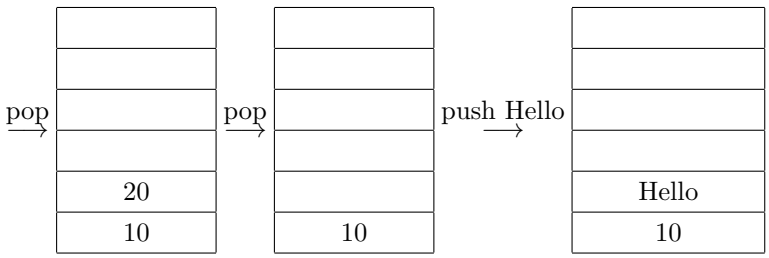
8.2 スタック

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 8.2 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.

問題 8.3 次の式を casio() が計算できるように後置形式になおせ.

1. $1+2+3$
2. $(1+2)*3$
3. $3+4*(5+6)$

補足: 図 8.1 の C 曲線を書くプログラムの説明. 関数 cCurve は C 曲線を構成する線分のリストを引数とし, 次のレベルの C 曲線を構成する線分のリストを戻値とする. リストの扱いについては 9 を参照. ここではリストの結合 (append) およびリストの先頭を除いた残りを戻す cdr(くったー と読む) の初歩的リスト処理関数を用いる.

今 cCurve の引数 P がリスト

```
[  [0,0],  [1,0]  ]
```

だとする. これは [0,0] と [1,0] を結ぶ線分を意味する. 線分の座標を取出すにはどうすればいいのであろうか? 配列 (ベクトル) と同じで P[0] により P の 0 番目, つまり [0,0] を取り出せる. P[1] は P の 1 番目, つまり [1,0] である. ではたとえば P[1][1] はどういう意味であろうか? P[1] は

[1,0] (再びリスト) なので, P[1][1] は [1,0] の 1 番目を意味する. よって 0 が P[1][1] の値となる.

関数 cCurve の中では線分の座標を取出したあと, [[0,0], [1,0]] を

```
[  [0,0], [1/2,-1/2], [1/2,-1/2],  [1,0]  ]
```

に作り直してこれを値として戻している.

次の実行例をみればわかるようにこの戻値に対してもう一度 cCurve を適用すると, 今度は 4 本の線分を戻す.

```
[1255] A=cCurve([ [0,0], [1,0] ]);
[[0,0],[1/2,-1/2],[1/2,-1/2],[1,0]]
[1256] B=cCurve(A);
[[0,0],[0,-1/2],[0,-1/2],[1/2,-1/2],[1/2,-1/2],[1,-1/2],[1,-1/2],[1,0]]
```

これを繰り返していき複雑な C 曲線を描くわけである.

問題 8.4

1. プログラムの残りの部分の仕組みを例を用いて解説しなさい.
2. 線分の座標のリストを生成するのは比較的重たい計算となる. 座標リストを生成せずに C 曲線を描画するプログラムを作りなさい.

問題 8.5

1. google で “C 曲線” を検索せよ. どのような記事があるか?
2. C 曲線を書くプログラムを模倣し, 同じような原理で自分オリジナルの図を描くプログラムを作成せよ.

```
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);")$ */
main(8)$
end$
```

図 8.1: 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 8.2: スタック電卓 `casio.rr`

第9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 9.1 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 9.2 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 9.1 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 9.2 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.)

9.1 リストに対する基本計算

リストについては第 6 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 5 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.

1	次のセルのアドレス	⟶	2	次のセルはもうないよとの印

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 9.3 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.

9.2 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 9.3 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 9.4 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 9.5 第 6 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 9.6 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 9.7 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 9.8 リストの要素を一列に並べる関数を書け.

$$[1, 2, [3, [4, 5]], 6, [7, 8]] \to [1, 2, 3, 4, 5, 6, 7, 8]$$

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 9.9 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10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．

10.1　バブルソートと クイックソート

10.3 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 10.1 [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 10.1　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 10.1 [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.

10.2　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.

10.3　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$
```

10.4　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.

10.4.1 ヒープ

次の性質を満たす図を考える (図 10.1). 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.

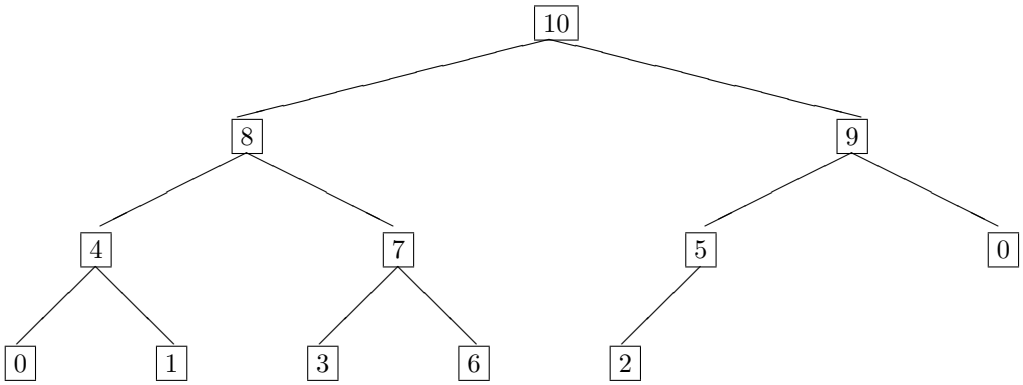


図 10.1: ヒープの例

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.

10.4.2　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 $A[1], \ldots, A[\lfloor \frac{N}{2} \rfloor]$.
- $A[I]$ の子は $A[2I]$, $A[2I+1]$ (もしあれば).
- $A[I]$ の親は $A[\lfloor \frac{I}{2} \rfloor]$

問題 **10.2** 上の性質を証明せよ.

10.4.3　`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 **10.3** このプログラムの動作を説明せよ.

問題 **10.4** 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.

10.4.4　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 $A[\lfloor \frac{N}{2} \rfloor]$ 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.

$A[\lfloor \frac{N}{2} \rfloor + 1]$ 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 **10.5** $L = [11, 9, 5, 15, 7, 12, 4, 1, 13, 3, 14, 10, 2, 6, 8]$ 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 10.1 ヒープソートの計算量は $O(N \log_2 N)$ である.

問題 **10.6** 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.

10.5 章末の問題

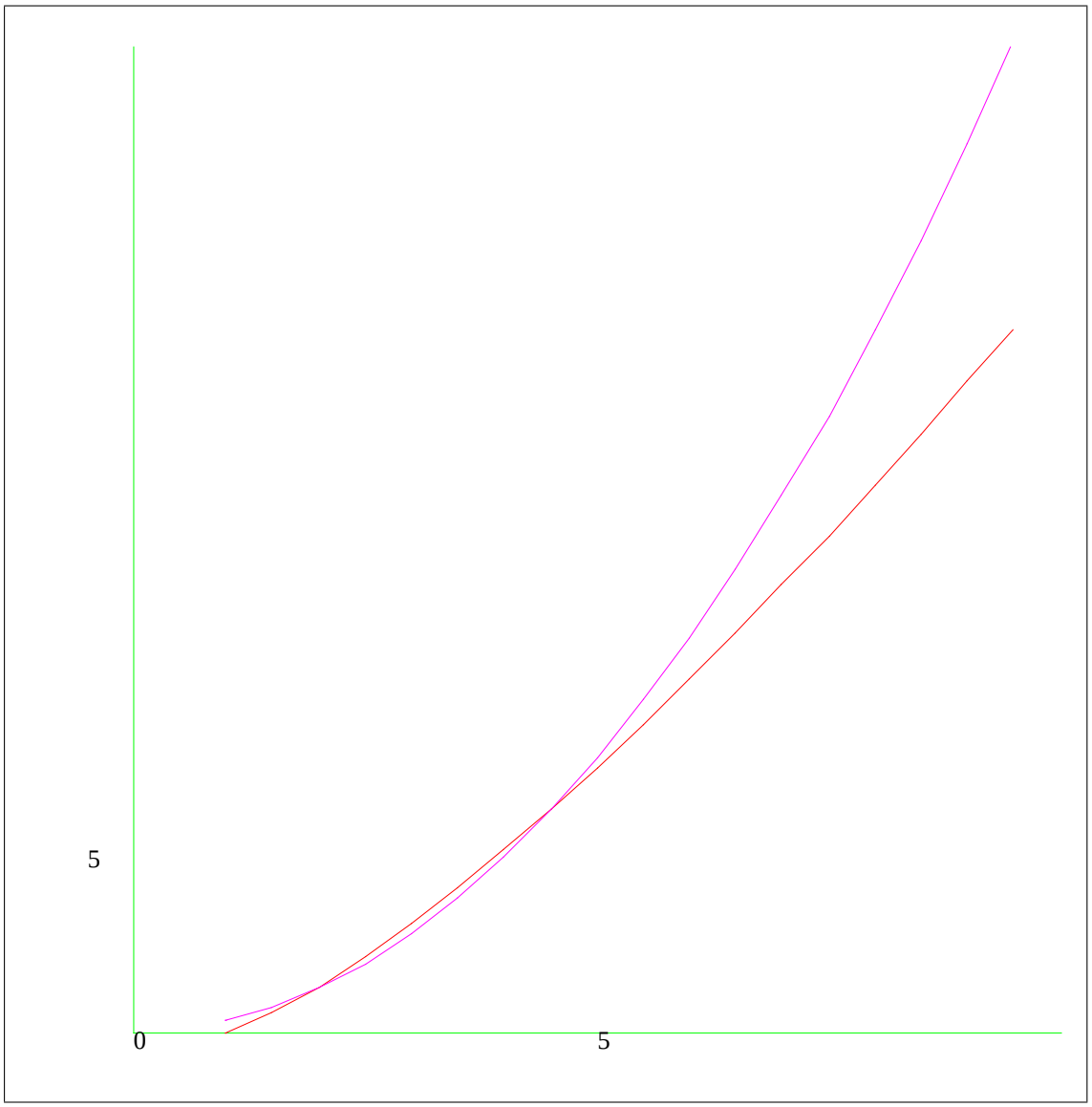
1. Selection sort, Insertion sort, Merge sort, Shell sort はどのようなソート法か調べ, これらおよび buble sort, quick sort について計算時間を 500 個から 4000 個程度のデータについて比較せよ. この計測データをもとに shell sort の計算量 $O(f(n))$ の $f(n)$ を推定しなさい.

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 4 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.)

4. buble sort, quick sort について計算時間を 500 個から 5000 個程度のデータについて計測し, 計算時間をグラフ表示するプログラムを書きなさい.

5. 1,2,3,4 を用いて長さ 4 のソート用のデータをすべて作れ. 重複も許すものとする. このデータを用いてあなたの書いたソートプログラムの正しさを確かめよ.



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11章　1 変数多項式の GCD とその応用

11.1　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.

11.2　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 11.1 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 11.2 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 11.3 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, **??** 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.

11.3　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 11.4 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 11 章 (本章) および **??** 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12章　RSA 暗号系

12.1　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 12.1 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 12.1 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 12.2 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 12.2 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 12.1 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.

12.2 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 12.1 で考察した.

定理 12.3 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 12.2 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 12.3 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, `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.

12.3 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,

$$\mathtt{X}^{\mathtt{A}} = \prod_{i\,:\,a_i \neq 0} \mathtt{X}^{2^i}$$

とかける. encode(X,A,N) では, X, $\mathtt{X}^2$, $\mathtt{X}^4$, ... 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

$$\mathtt{X}^{\mathtt{A}} \bmod \mathtt{N} = \prod_{i\,:\,a_i \neq 0} \mathtt{X}^{2^i} \bmod \mathtt{N}$$

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

[14663494090011329650434277764996684859263420110662305743007865202299126408269 6]
[361] decrypt(@@);
Block_size = 32
The input message to decrypt =
[14663494090011329650434277764996684859263420110662305743007865202299126408269 6]
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 12.2 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 12.3 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?

索 引