超入門 Emacs/asir

高山信毅

2020 年 9 月 28 日版
コメントは takayama@math.kobe-u.ac.jp まで

目 次

第1章　電卓としての利用

“Emacs/asir 超入門” は cfep/asir 超入門を Emacs/asir 用に書き直したものである．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説明した．

この節では Emacs/asir の起動法, 電卓風, Basic 風の使い方を説明する．Emacs は長い歴史と多くの愛用者を持つすばらしいテキストエディタであるが, 独特のインタフェースを持つため最初は少々戸惑うこともあるかもしれない．しかし慣れれば快適であることを保証する．

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+キー と書くことにする．Control (ctrl) キーを押しながら他のキーを押す操作を CTRL+キー と書くことにする．command キー, alt キー等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. Mac 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．Windows の場合 ¥ と画面に表示されていても \ と同等．またキーボードで ¥ と入力すればプログラムは \ と解釈する．このあたりの事情は文字コードの章で詳しく解説.

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 UTF-8 コード (日本語を含む) 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．

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．

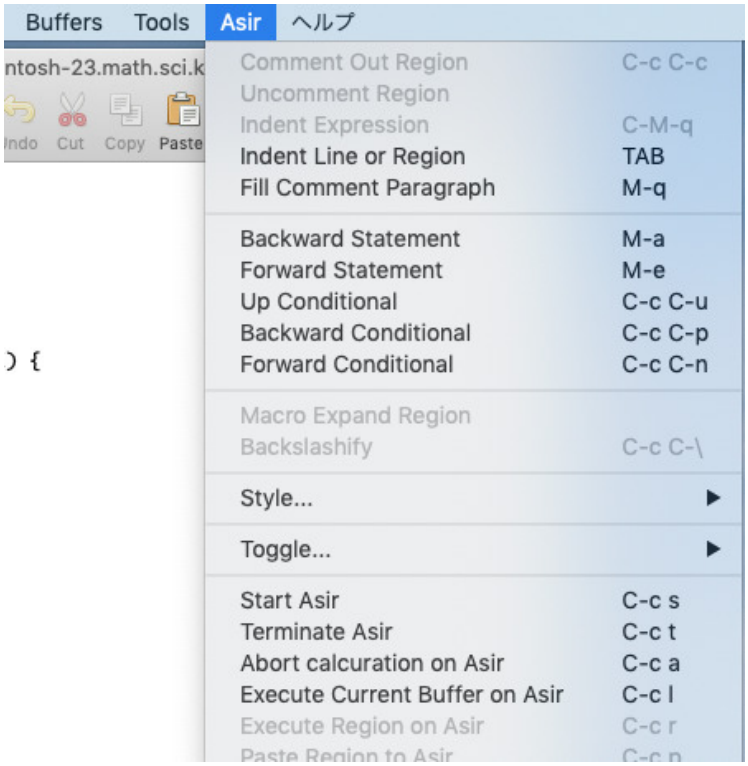


図 1.1: Emacs/asir の asir メニュー

(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.

1.2 Emacs/Asir の起動法と電卓的な使い方

Emacs はテキストエディターであるが, 多彩な拡張機能を持つ. たとえば拡張機能を用いてメールの読み書きをすることも可能である. Emacs/asir は asir 用のプログラムファイルの編集のみならず, この拡張機能を用いて asir の実行などを行う.

さて, Emacs を起動して拡張子が `.rr` のファイルを Visit new file (新規作成) か Open (開く) をすると, 図 1.1 のように Emacs/asir 用の拡張メニュー “Asir” が使えるようになる. [1]. 以下 Emacs/asir を単に asir とよぶ.

[1] アプリケーションの emacs を起動してから, たとえば test.rr を visit new file (新規ファイルの open) をする (図 1.2). または, たとえば `emacs test.rr &` とターミナルから起動する.

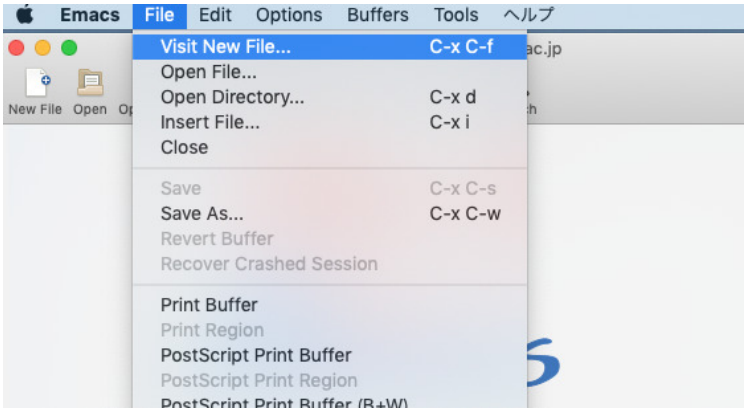


図 1.2: Emacs での新規ファイル作成

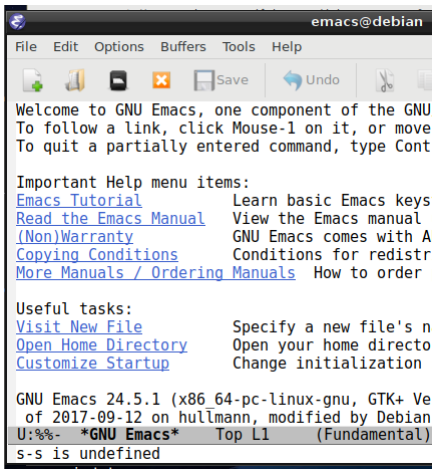


図 1.3: Debian の emacs の画面, 図 1.2 は Mac の emacs の画面, メニューの位置がすこし違う

まず “Asir” メニューの ”Start Asir” を選択すると asir が新しい window で起動する (図 1.4 右側の window, 名前は asir-cmd).

たとえば電卓のように使うにはどのようにすればいいのであろうか? それには, 図 1.5 の emacs のファイル編集画面 (左) に計算したい式やプログラムを入力してファイルを保存する[2].

次に “asir” メニューの ”Execute Current Buffer on Asir” を選択すると (図 1.1), 図 1.5 のように asir 実行画面 (右の window) でプログラムが実行され結果がそのそのウインドウに表示される. この例では $3 \times 4 + 1$ が計算されて, 13 が (右の asir 実行画面に) 表示されている.

“Execute Current Buffer on Asir” メニューで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. なお, ファイルを保存せずに評価しようとすると, ‘‘ファイル名’’ not found in the search path ... と asir 実行画面 (asir-cmd) に表示されるか, もし保存前のファイルがあれば, そのファイルの内容が評価されるので注意してほしい.

Asir を終了するには Asir メニュー (図 1.1) の “Terminate Asir” を実行する.

なお本当に終了してよいか? や, 終了前にファイルを保存しますか? などの確認は図 1.6 のようにファイル編集画面の下部の mini buffer 領域に表示される場合がある. この場合は yes や no を mini

[2]詳しく説明すると, File メニュー (図 1.2) から”Save(保存)” (や”Save As (別名で保存)”) を実行するとファイル編集画面 (左の window) の内容をファイルとして保存できる.

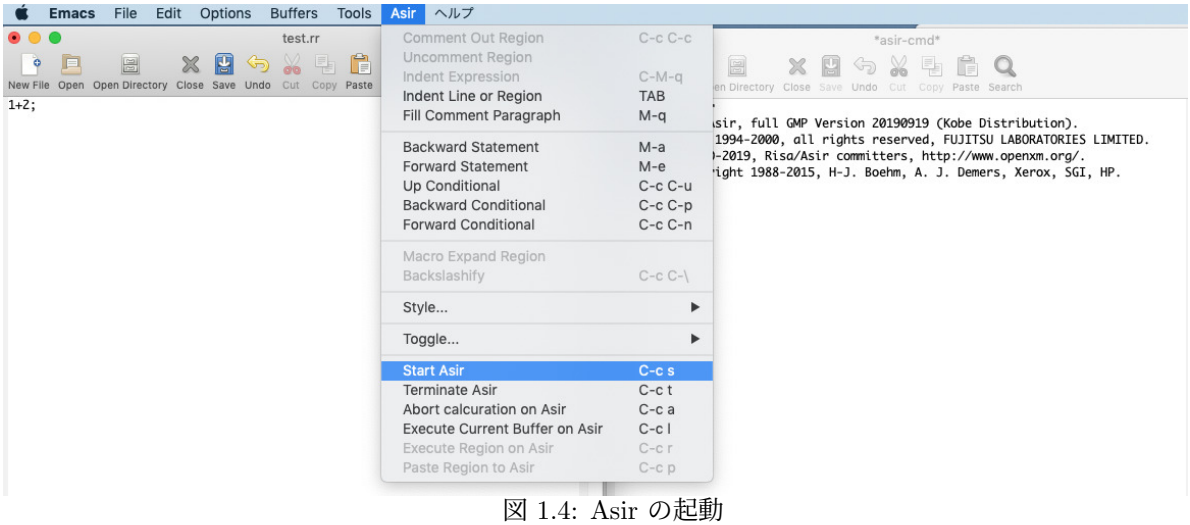


図 1.4: Asir の起動

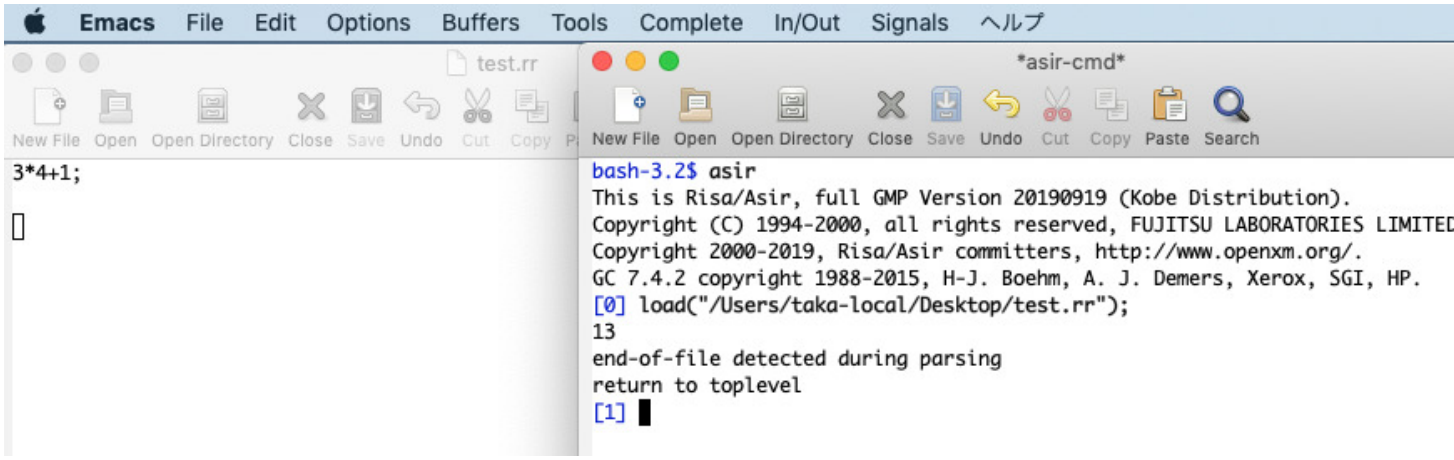


図 1.5: ファイル編集画面 (window, test.rr) と asir の実行画面 (window, asir-cmd)

buffer 領域に入力する.

たとえば “Terminate Asir” メニューで asir を終了せずに emacs を quit (終了) しようとすると, メッセージ `Active process exist; kill them and exit anyway?` (yes or no)(図 1.6) が mini buffer に表示されて yes か no の入力を求められる. (emacs のバージョンによってはこれは図 1.7 のようなダイアログの場合もある.) 一気に asir も終了させたかったら yes でよい.

またたとえばファイル入力画面のファイルを保存せず emacs を quit しようとすると “Save file ...? (y,n, ...)” なるメッセージが mini buffer に表示される (図 1.8). 保存には y を押せば良い. または, 図 1.9 のような dialog が表示される場合もある.

> なお, emacs で訳のわからない状態になったら CTRL+g を押すと脱出できる場合が多いので覚えておくとよい.

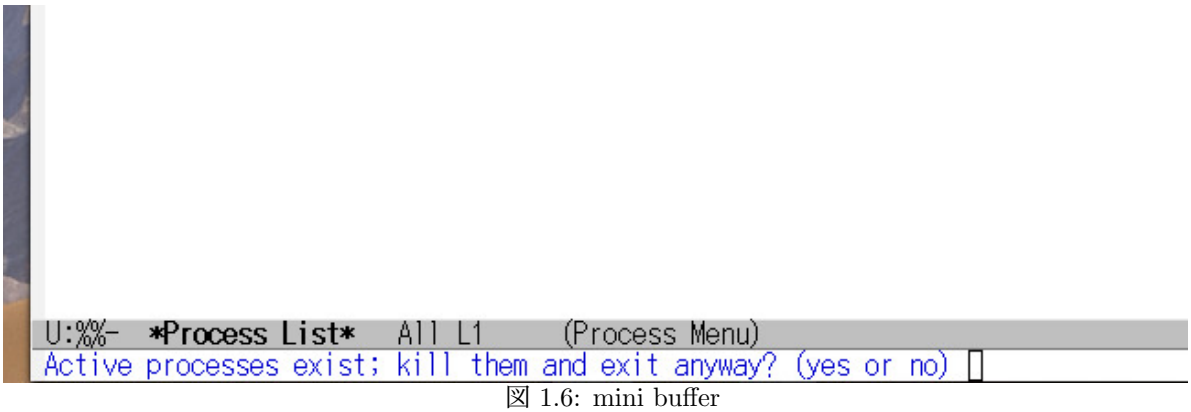


図 1.6: mini buffer

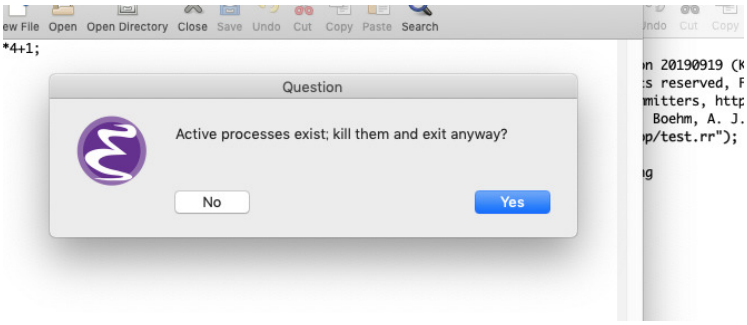


図 1.7: Active process dialog

さて図 1.5 では $3 \times 4 + 1$ の計算をしている. 手順を復習しておく.

1. emacs を起動
2. File メニューの “Visit new file” を選び test.rr を新しく作る.
3. Asir メニューが出現するので, Asir メニューの “Start Asir” を選ぶ.
4. test.rr の編集画面 をクリックし, `3*5+1;` を入力し, File メニューから Save(保存).
5. Asir メニューの “Evaluate Current Buffer on Asir” を選択.

なお, ‘‘`end-of-file detected during parsing. return to toplevel`’’ というメッセージを消すには, 入力ファイルの最後に `end$` と書いておけばよい.

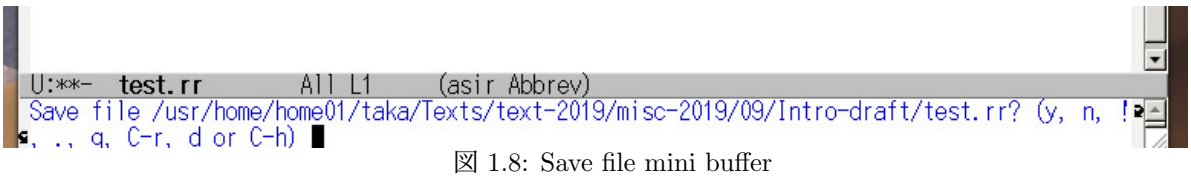


図 1.8: Save file mini buffer

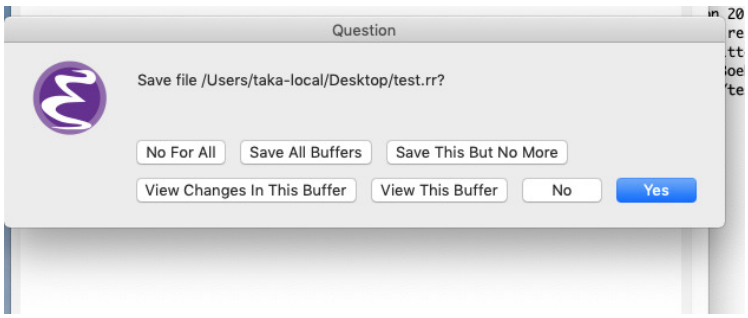


図 1.9: Save file dialog

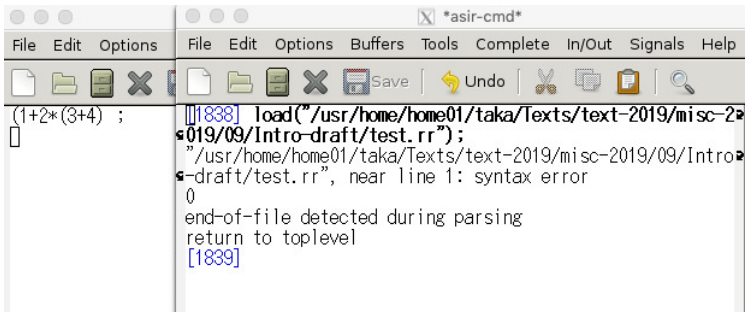


図 1.10: エラー表示

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 × や ÷ 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 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. emacs は括弧を閉じると対応する括弧にカーソルが短時間飛ぶ. 図 1.10 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.

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 1990 年代から 2000 年代にかけて高校の数学の教科書等にも登場した．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．最近流行の python 言語も, 昔 Basic に馴染んだ人たちの一部は Basic みたい, と感想をもらしている．

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．

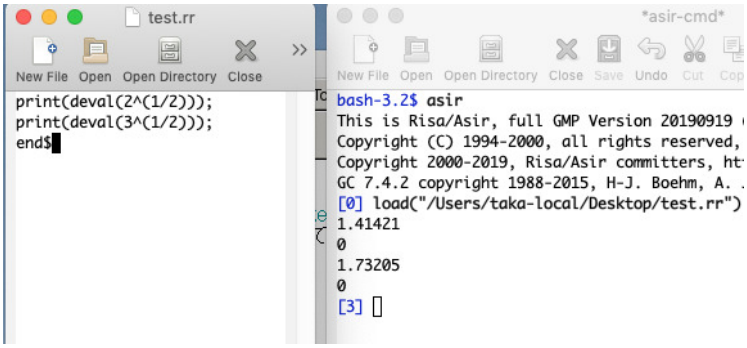


図 1.11: 平方根の計算

例 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 1.11 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. 途中の `0` はなんなのであろうか? 実はこれは `print` 文の戻している値である. [3]

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.

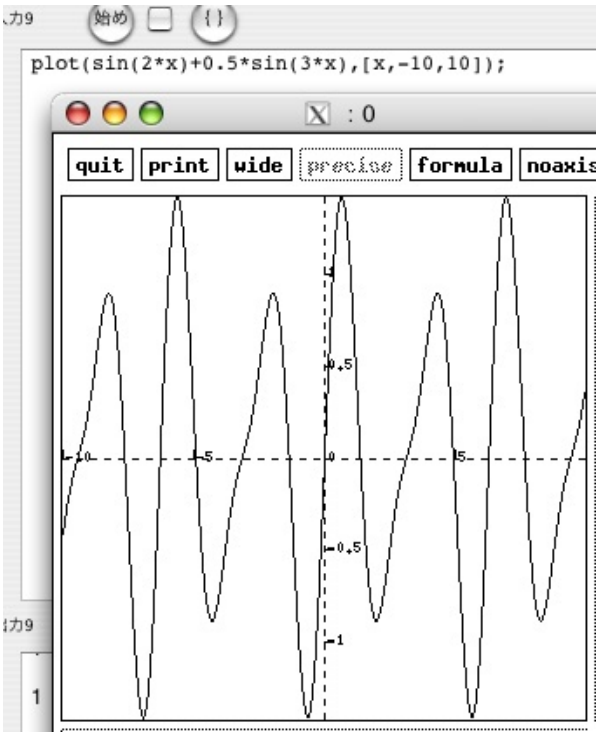


図 1.12: 関数のグラフ

発展学習 `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.

[3] `;` の代わりに `$` を用いるとこの戻り値を表示しない.

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．グラフの形が誤っている場合もある．その場合はどうしてそうなるのか理由を考えよ．

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．

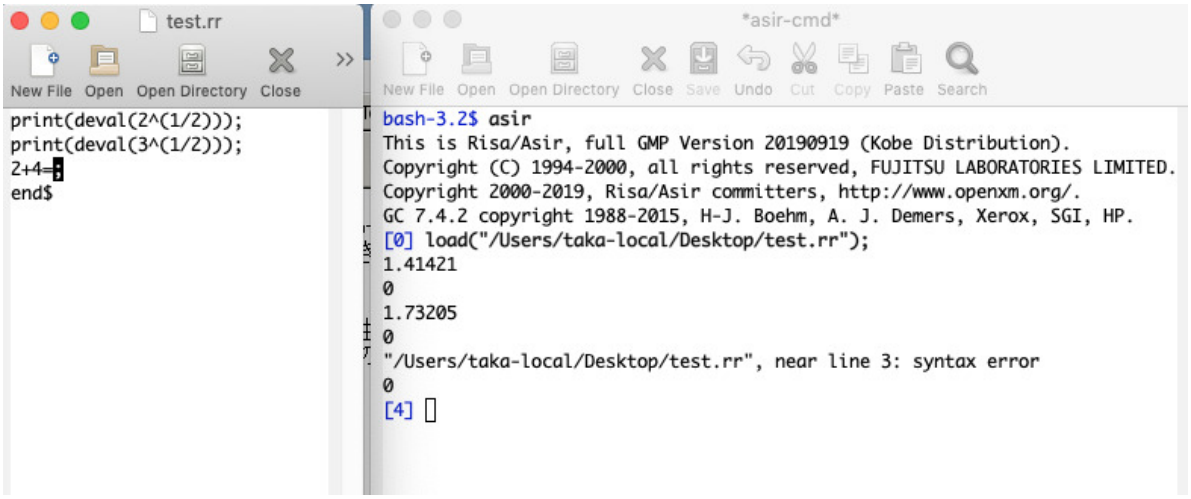


図 1.13: 文法エラー

図 1.13 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, “syntax error(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.

図 1.14 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．

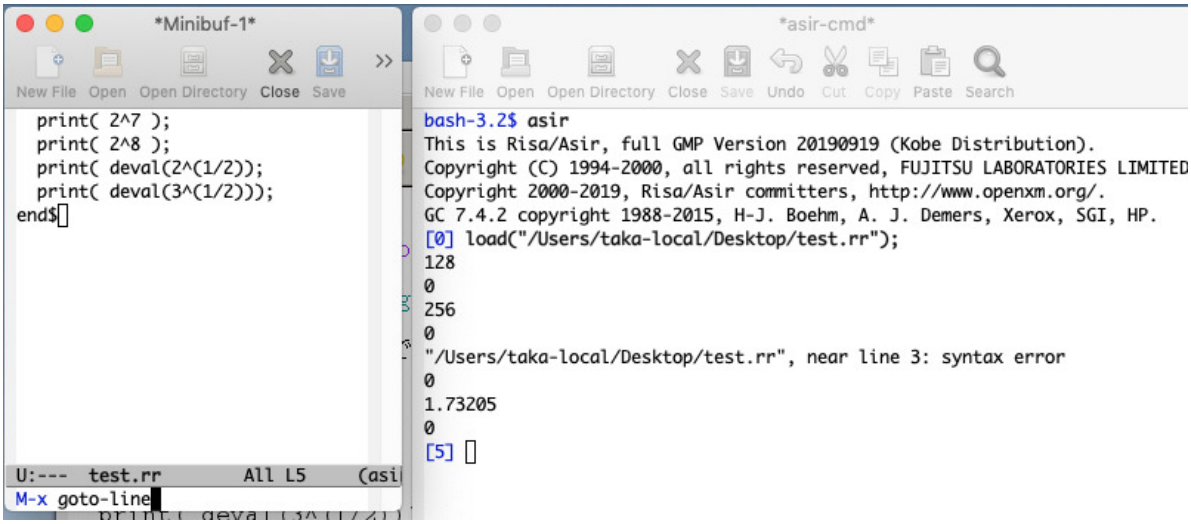


図 1.14: エラー行

エラーメッセージにはエラーをおこした行の番号が含まれている. その行をファイル編集画面で見つけるにはどうすればいいのであろうか? それには Edit メニューの “Go to”, “Go to line” を選択して (図 1.15), jump すべき行を mini buffer へ入力すると, その行へ移動できる. または emacs に対して esc x goto-line と入力して, jump すべき行を mini buffer へ入力してもよい.

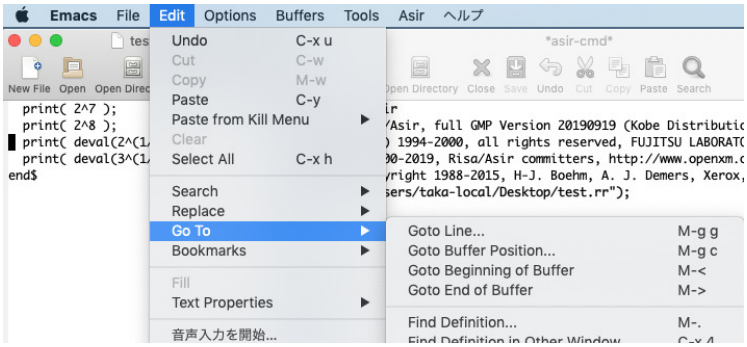


図 1.15: エラー行へ jump する

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.

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.

第2章　変数とプログラム

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．

```
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．なお, print の戻り値を表示させないために ; の代わりに$ を用いている．

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 )$
```

を実行すると最後に 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 2.1 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 2.2 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

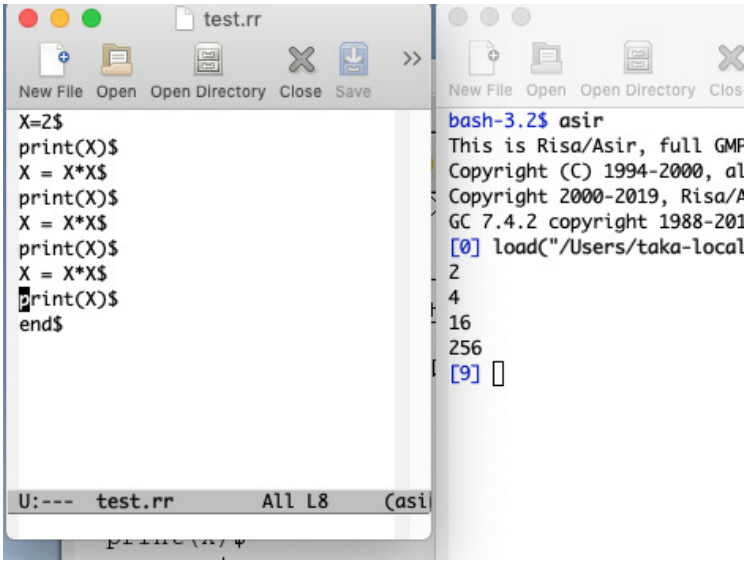


図 2.1: 変数の利用

```
[9] fctr(x^2+2*x*y+y^2);
[[1,1],[x+y,2]]
[10] fctr(x^2-1);
[[1,1],[x-1,1],[x+1,1]]
[11]
```

図 2.2: 因数分解

いる．図 2.2 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．

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 2.1 この節の初めに書いた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 2.3 をみよ.

```
[11] load("/Users/taka-local/Desktop/test.rr");
2
2
4
8
16
32
64
128
256
[14]
```

図 2.3: for 文

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.

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.

解説　さて図 2.3 の出力には 2 が二つある. 一つ目の 2 は X=2; に対応しており, この文の評価値 2 が表示されている. 二つ目の 2 は print(XÎ); により出力された 2 である. 以前の例ではさらに print の戻り値の 0 が表示されていたが, この場合は表示されていないことに注意してほしい. ; で終わる文の値が表示されるのは, それが top level の文であるときのみである. つまり, たとえば { } の中にある ; で終わる文の値が表示されることはない.

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. 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.

上のプログラムでは for の中のひとまとまりのプログラムは字下げ (インデント, indent) して書いてある. これにより for で繰り返される処理が一目瞭然となる. 正しいインデントで読みやすいプログラムを書くことを心がけてほしい. emacs にはインデントを自動で行う機能がある. 自動でインデントさせるには return (enter) の代わりに CTRL+J を打ち込む. インデントが必要な状況では自動的に emacs がインデントをおこなう.

上のプログラムは printf 函数を用いると以下のように簡略に書ける.

```
X = 2;
for (I=1; I<=8; I++) {
   printf("2 の%a 乗は%a\n",I,X^I);
}
```

%a が対応する変数の内容に置換される. この例では最初の %a が I の値に, 二番目の %a が X^I の値に置換される. \ n は改行を意味する.

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 2.2 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. C 言語風の `printf` 文を使うと同じことがもっと簡潔に記述できる.

```
for (I=0; I<2; I = I+0.2) {
   printf("%a : %a\n",I,deval(I^(1/2)));
}
```

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 2.1 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` $\div$ `B` の余りは `A%B` で計算できる.

2.3　実行の中止

実行中のプログラムの実行を中止したい時は asir メニューの “Abort Calculation on Asir” を選ぶ.

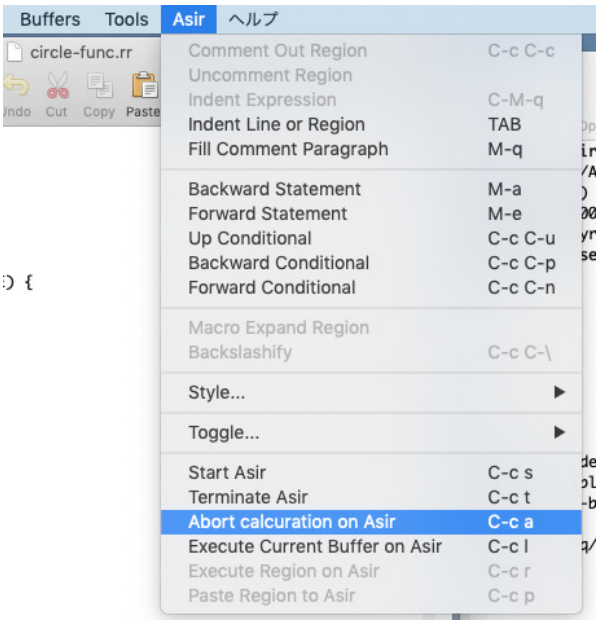


図 2.4 では 10^{100} 回の `Hello` の出力の繰り返しを中止している.

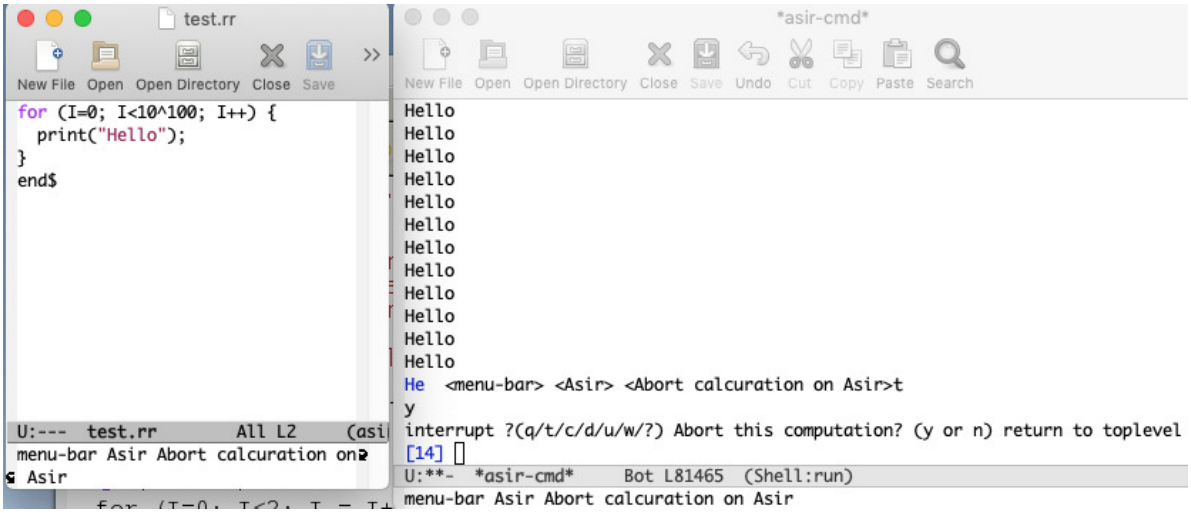


図 2.4: 実行の中止

2.4 マニュアルの利用

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`, `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. `print`, `deval`, `fctr` 等は組み込み関数である. 組み込み関数の詳しい説明を調べるには `http://www.math.kobe-u.ac.jp/Asir/index-ja.html` の “日本語による参考書” の 1 のリンクをクリックして, “Asir のマニュアル”, “日本語, HTML” の “(目次)” をクリックして探せば良い.

第3章　グラフィック

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 3.1 のように線を描画する.

なお 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.

```
import("names.rr")$
```

により asir-contrib が読み込まれる. [1]

3.2　線を引く関数

例 3.1

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

図 3.1 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 3.2 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.

[1] 神戸大学の実習環境では security 設定のためこのライブラリを読み込むのに 1 分以上かかる.

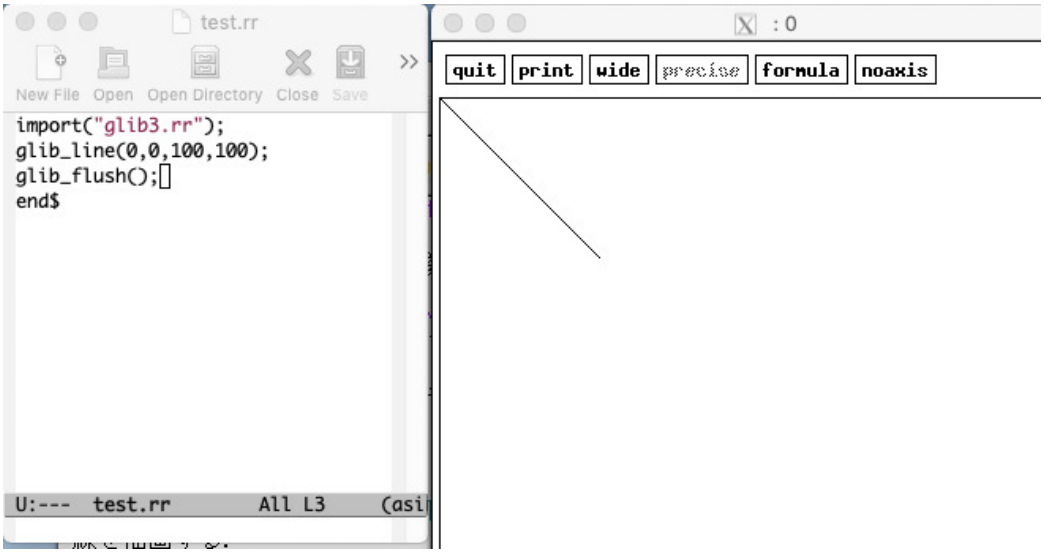


図 3.1: ライブラリのロード

`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 3.2).
`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 3.2: 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 3.2 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.

例 3.2 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 3.3. ——プログラムの解説はまだ書いてない.

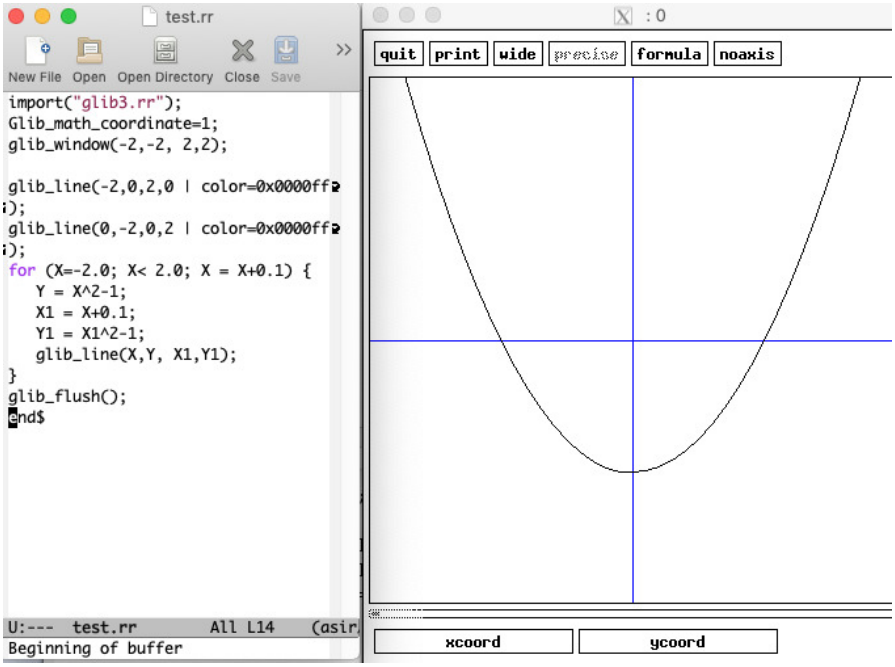


図 3.3: 2 次関数のグラフ

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.

円の例をしばらく離れ, 簡単な関数の例をとり関数の書き方を説明しよう. 前の節では 2 の巾の表を作成するプログラムを書いた. これを元に次のような関数を作る.

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

関数の定義は次のように def 命令で行なう.

```
def  関数名 (引数) {
  関数本体
}
```

上の例では関数名は power_table であり, 引数 (argument) は N である. 関数名は英数字と _ を用いてつける. ただし数字や大文字ではじまる名前をつけることはできない. 処理内容を連想させるような名前をつけるのが望ましい. def 命令では関数を定義するだけで実行は行なわない. 実行させるには, 上の例のように power_table(8); と引数の部分に実際の数字等を入れて呼び出す.

引数は二つ以上あってもよくて, たとえば,

```
def power_table2(X,N) {
  for (I=1; I<=N; I++) {
    print(X^I);
  }
}
power_table2(3,8);
```

なる関数定義と最後の行のその呼び出しは 3^i を $i = 1, \ldots, 8$ の範囲で計算して表示する. このような関数を用意しておけば, 2^i, 3^i, 5^i, $1 \leq i \leq 10$ の表を表示したいとすると,

```
power_table2(2,10)$
power_table2(3,10)$
power_table2(5,10)$
```

と書くだけで良く, プログラムも短くなり, かつ整理されているので, 読みやすくなる.

さて, power_table2(2,3); を実行すると

```
2
4
8
0
```

と表示される. 最後の 0 は一体何であろうか? これは実は関数の値である. 関数の値のことを戻り値ともいう. 戻値を指定するには, return 文を用いる.

```
def twotimes(N) {
  S=2*N;
  return(S);
}
```

関数 twotimes(N) は $2N$ の値を計算して戻す. この定義を書いておいて

```
A=twotimes(10)$
B=twotimes(100)$
print(A+B);
```

を実行すると, A には 20 が代入され, B には 200 が代入され, print 文により 220 が出力される. そのあと 0 が出力されるがこれは print 文の戻り値である.

関数の戻り値 (return value) は return 文で指定する. 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 twotimes は引数の 2 倍を計算して結果を戻す” みたいに使う. 上の例でいえば A=twotimes(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, 処理を分割した利点がすくない. そこでででてきた概念がこの “局所変数” の概念である. 上のプログラム例では, N, S 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:

```
S=3;
twotimes(2);
print(S);
```

このプログラムの S と関数 twotimes のなかの変数 S は別物である. したがって, twotimes の終了時点で関数 twotimes のなかの変数 S の値は 6 であるが, print 文で S の値を表示させてみてもやはり 3 のままである.

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 3.4 のような同心円の図を描くことが可能となる. 関数についてさらに詳しくは “asir ドリル” を参照してほしい.

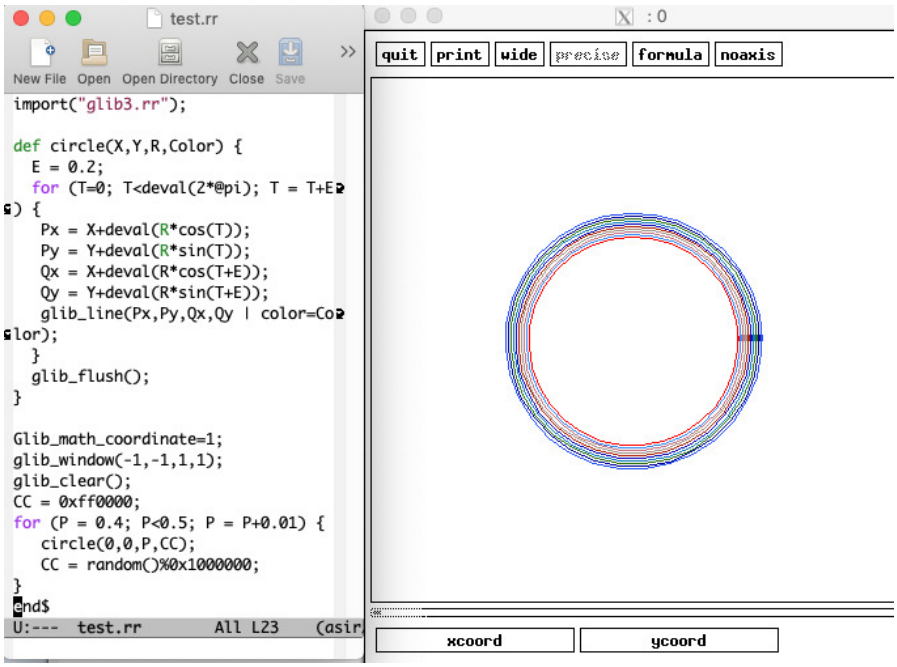


図 3.4: 関数による同心円の描画

```
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;
}
```

(Qx,Qy) と (Px,Py) は E だけ偏角が異る. 円を多面体で近似して描画している.

——プログラムの詳しい解説まだ.

問題 3.1　1. このプログラムの関数を用いて接する半径の同じ円を二つ描画するプログラムを書きなさい.

2. 円を塗り潰す関数を作れ.
3. 分度器を描くプログラムを作れ.
4. (発展課題) この分度器, 糸, おもり, わりばし, 板, 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.

実習の落とし穴

1. 空白や改行は原則自由に挿入していいが, 空白を入れてはいけない表現がある. たとえば

   `0.1`

   と書くべきところを

   `0. 1`

   と 1 の前に空白を入れるとエラーとなる. 数字には空白を入れてはいけない. 理由は asir ドリルの構文解析の章を読むと理解できると思う.

2. `sin x` なる表現は受け付けてくれない. 必ず括弧がいる. つまり `sin(x)` と書く.
3. 掛け算の `*` は省略できない.

発展学習. `[1,2,3]` のように `[]` で囲った表現をリスト (list) と呼ぶ. 関数の戻り値を数の組にしたいときはリストを値として戻すと良い. `print` 文で多くの数を一行で表示したいときもリストを使うと便利である. `L` をリストとするとき `L[0]` でリストの最初 (0 番目) の元, `L[1]` でリストの 1 番目の元, ... を表す. 詳しくは asir ドリル参照.

```
L=[3,2,1];
print(L);
print(L[0]+L[1]+L[2]);
```

3.4 debugger

関数を含むプログラムの実行中に誤りが見つかると asir の実行画面 に誤りの行番号と `(debug)` プロンプトが表示される. これは asir の debugger である. `(debug)` に対して quit を入力すると asir の実行画面 が通常の入力待ちの状態になる. `(debug)` に対して, “Execute Current Buffer on Asir” をやっても正しく動作しないので注意. かならず図のような quit の入力で [数字] の通常の入力待ちの状態に戻す. debugger の詳しい使い方は asir ドリルを参照してほしい.

たとえば図 3.5 では関数 `power_table` の中で, 関数 `pprint` を呼び出している. 4 行目のこのような関数は存在しない (pprint undefined) というエラーメッセージが表示されて, debugger が起動している. `(debug)` に対して `quit` と入力して, asir の実行画面が通常の入力待ち状態 `[1]` に復帰している.

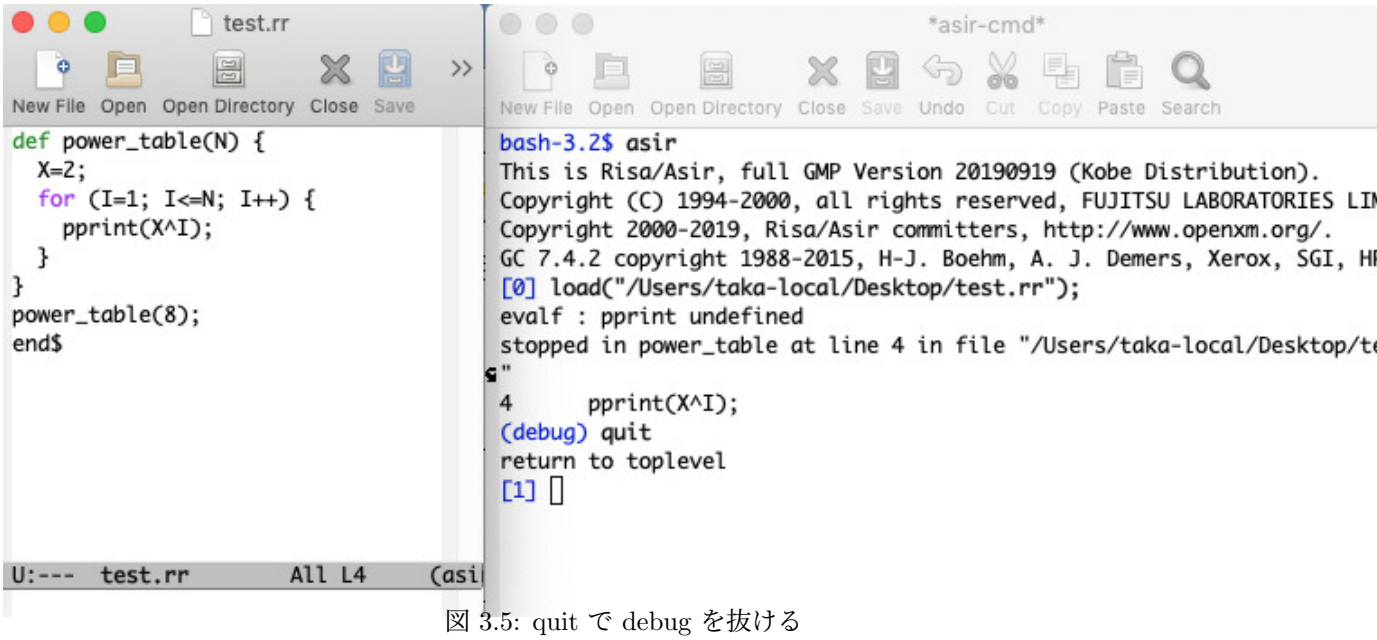


図 3.5: quit で debug を抜ける

第4章　For 文による数列の計算

4.1　超入門, 第2の関門: 漸化式できまる数列の計算

例 4.1 a を正の数とするとき,

$$\begin{aligned} x_{n+1} &= \frac{x_n + \frac{a}{x_n}}{2}, \\ x_0 &= a \end{aligned}$$

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 4.1.

```
[88] load("/Users/taka-local/Desktop/test.rr");
2
2
1.5
1.41667
1.41422
1.41421
1.41421
[92]
```

図 4.1: $\sqrt{2}$ 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ら, 超入門卒業である.

問題 **4.1** 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.

表は次のような形式で書く.

I	0	1	2	3	4
X					
Y					

問題 **4.2** プログラムのバグ (bug) とはなにか?

4.2　円を描く数列

前の章で円を描く関数を紹介した. sin,cos の計算に時間がかかる計算機では, なるべくこれら三角関数を用いないで円を描画する必要がある.

一つの方法は $s = \tan\frac{t}{2}$ とおくと $\cos t = \frac{1+s^2}{1-s^2}$, $\sin t = \frac{2s}{1-s^2}$ と書けるという公式を使う方法である. s はタンジェントで定義されていることを忘れてしまえばよい.

もう一つは数列の計算を用いて, cos や sin の計算をやらずに円を描く方法である.

```
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();
}
```

——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.

問題 4.3 (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). なお図のパーツに対応する関数を定義して (たとえば円や長方形など) 図を描くプログラムを書くこと.

ノート. 著者の場合, 週に 1 コマ講義, 演習 1 コマでは, ここまでで 3 週である. 上のレポート問題が最初のレポート. 3 週目の演習の時間からとりかかり, 5 週目の演習の時間に発表. 4 週目からは asir ドリル.

4.3 Emacs/asir のインストール

4.3.1 MacOS X

emacs, X11, xcode(gcc, make 等) はすでにインストール済みとする.

1. `http://www.math.kobe-u.ac.jp/Asir/index-ja.html` より Mac 用 Emacs/asir をダウンロードする.


4. Mac OS X 用のバイナリをダウンロードする.
   - cfep/asir for MacOS X (Intel Mac 用, テスト版)をダウンロードする.
   - emacs/asir (emacs から使える asir) for MacOS X をダウンロードする.

2. emacs-asir*.dmg ファイルをダブルクリック. Desktop に emacs-asir で始まる名前の disk

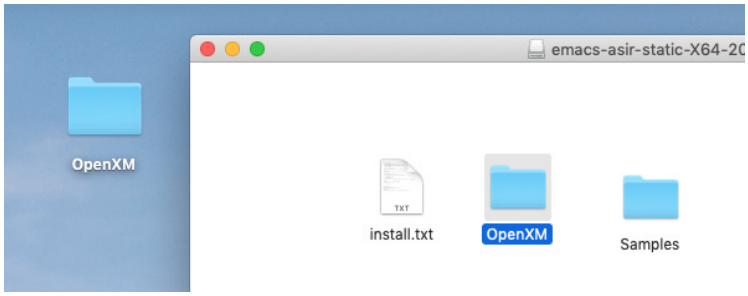


図 4.2: asir が入ってるディスクから drag & drop でコピー

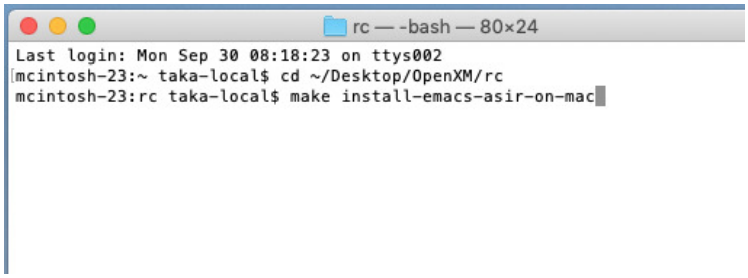
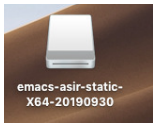


図 4.3: `make install-emacs-asir-on-mac` の後の `return` を押す前



が現れるので, それをダブルクリック,

3. その disk の中の OpenXM を Desktop へコピー, 図 4.2.
4. アプリケーションのユーティリティにあるターミナル (terminal) を起動。以下を入力する.

```
cd ~/Desktop/OpenXM/rc
make install-emacs-asir-on-mac
make norc-fix
```

一度 logout して再度 login.

図 4.3 は, ターミナルを起動して上のコマンドを入力しているところ. `Return` を押すと沢山メッセージがでる. メッセージ中に error が出力されなければ OK. 4.3.7 節に上記が正しく動作したときのメッセージ例を掲載.

なお, Dock にターミナルを置いておくと便利. `emacs` はターミナルより,

```
emacs &
```

で起動. アプリケーションの emacs を double click により起動して, emacs/asir を動作させるには `.emacs.d/init.el` に追加の設定が必要となります (4.3.4 節を参照).

4.3.2 MathLibre 仮想マシンでの利用

MathLibre 仮想マシンは `http://www.mathlibre.org` が配布している debian GNU linux をベースとした数学ソフトウエア環境を仮想マシンとしたものである. 配布元は `http://www.math.kobe-u.ac.jp/vmkm`. emacs や asir はすでにインストール済みなので, 仮想マシンを実行する VMware player または Virtual box をインストールしておけば, あとは仮想マシンのファイルをダウンロードして仮想マシンを起動すればよい (参考動画: `https://youtu.be/sDrgq9juNN0`, usb メモリで配布してる VMware 用 mathlibre-2019 仮想マシンのコピーと実行).

仮想マシンのウインドーサイズが狭い時は枠を drag することで大きくできる. ウインドーは大きくなったが, 仮想マシンの画面が小さいままのときは, 左下の menu アイコン, 設定, モニターの設定, で仮想マシンの画面サイズを設定できる.

仮想マシンは DVD の iso image から起動され, ユーザの行った変更が仮想マシンの仮想ハードディスクに記録される. たとえば `apt-get` などでソフトウエアの更新をすると仮想ハードディスクに更新内容が記録され DVD 由来のファイルシステムに上書きされたように見える. これには union file system という仕組みが使われている.

4.3.3 インストールでのトラブル対策

Q. .rr ファイルを開いても, Asir メニューが出てきません.
A. ターミナルから

```
ls  -l ~/.emacs  ~/.emacs.d/init.el ~/.emacs.d/OpenXM
```

と入力してください. 下記のように表示されますか?

```
bash-3.2$ ls ~/.emacs  ~/.emacs.d/init.el ~/.emacs.d/OpenXM
ls: /Users/ログイン名/.emacs: No such file or directory
/Users/ログイン名/.emacs.d/init.el

/Users/ログイン名/.emacs.d/OpenXM:
asir-mode.el
```

もし `.emacs` ファイルが存在すればそのファイルを `emacs  /.emacs` コマンドで開いて,

```
(load "~/.emacs.d/init.el")
```

と書かれた行があるか調べてください. もしなければこの行を追加します.

`init.el` や `OpenXM` が無い (No such file or directory) と表示されたらターミナルから
`openxm use-asir-mode.sh --local-yes`
と入力することでこれらがコピーされる. `asir-mode.el` は Asir メニューに対応する emacs lisp のファイル, `init.el` から `asir-mode.el` を読み込んでいます.

4.3.4 Emacs のカスタマイズ

Q. emacs の下半分に余計な画面 (図 4.4) が出てきてそれを消したい場合.
A. とりあえず二分しているバーを下へ drag すると余分な画面を小さくできます. 根本解決には, ターミナルから

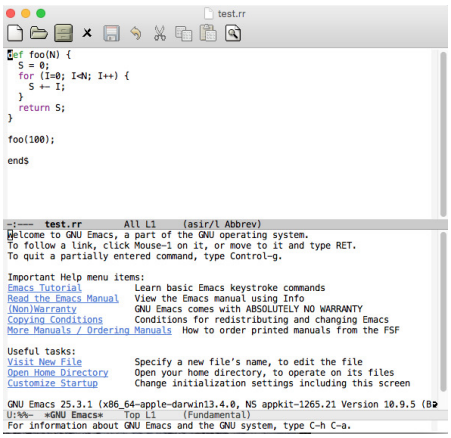


図 4.4: 下半分は不要?

```
emacs ~/.emacs.d/init.el &
```

で `init.el` ファイルを編集.
　init.el の最後に

```
(setq inhibit-startup-screen t)
```

を追加して, emacs を保存終了してもう一度起動.

Q. emacs の window サイズが大きくて下が切れます.
A.

```
emacs  --geometry 80x25
```

と emacs を起動すると横半角 80 字, 縦半角 25 字の画面となります.
　`init.el` に window サイズを設定しておくには, たとえば以下を `init.el` に書き込んでおきます.

```
(setq default-frame-alist
  (append
    '((width . 80)
      (height . 25))
    default-frame-alist))
```

Q. ダブルクリックで emacs を起動すると “Start Asir” が失敗する.
A. `~bin` がコマンドサーチパスに入っていないのが原因です. `https://qiita.com/h12o/items/f0db094fde6640143f42` によると, 下記を `init.el` へ追記しておけばうまくいきます.

```
(if (file-directory-p (expand-file-name "~/bin"))
    (progn
      (add-to-list 'exec-path (expand-file-name "~/bin"))
      (let ((exec-path (reverse (cdr-safe (reverse exec-path)))))
        (setenv "PATH" (mapconcat 'identity exec-path path-separator)))))
```

上記を書き加えた `init.el` の画面は図 4.5.

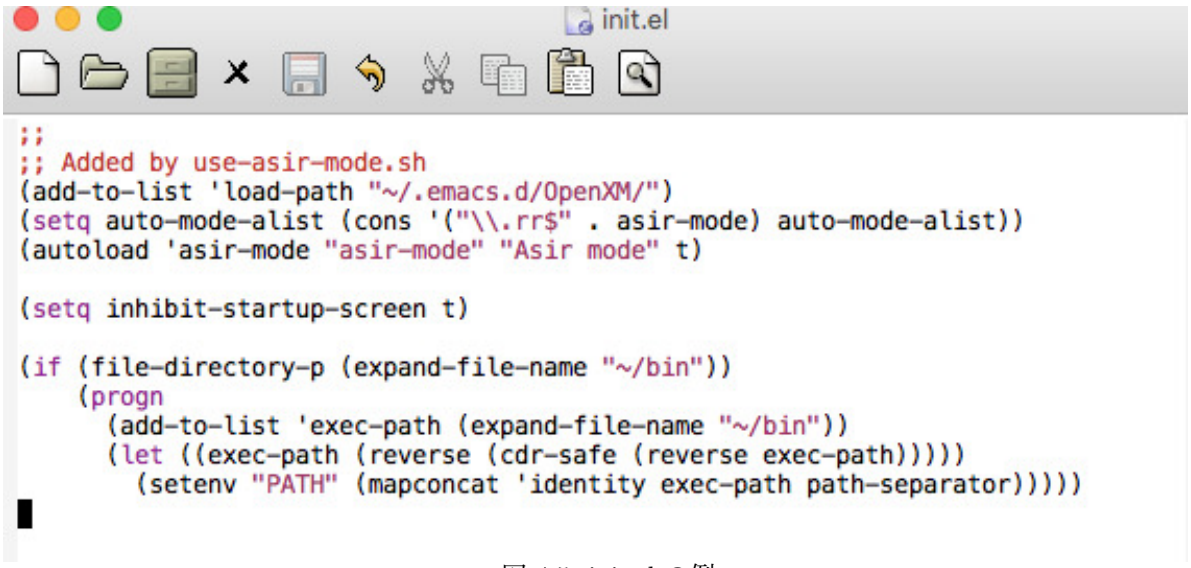


図 4.5: init.el の例

4.3.5 unix

emacs のパッケージは普通あらかじめ入っている. openxm の debian package (openxm*.deb, * はバージョンなど) を `http://www.openxm.org` よりダウンロード.

```
sudo dpkg --install openxm*.deb
```

でインストール. Debian でない場合は `http://www.openxm.org` よりソースコードをダウンロードしてソースの `OpenXM/src` で `make configure ; make install`

さらに `OpenXM/rc` で `make` して, 生成した `openxm` をサーチパスへインストール. ターミナルより

```
openxm use-asir-mode.sh --local-yes
```

で asir-mode.el 等を自動設定する.

4.3.6 Windows

[1] Windows 用の emacs をネットで探してインストール. asirgui を asir の配布サイトからダウンロード, インストール. 同梱してある emacs 用の設定に従い asir-mode.el 等をインストール. なお Windows では日本語など英語以外の出力はできないか不審な動作をします (UTF-8 に対応できてないため). 詳しくはまだ書いてない.

4.3.7 `make install-emacs-asir-on-mac` のメッセージ例

```
lib-shizen-a-018:~ takayama$ cd ~/Desktop/OpenXM/rc
lib-shizen-a-018:rc takayama$ make install-emacs-asir-on-mac
rm -f repl
gcc -o ../bin/rc.repl repl.c
repl.c:50:46: warning: format specifies type 'int' but the argument has type
      'char *' [-Wformat]
        fprintf(stderr,"Your prefix is %d\n",cwd);
                                        ~~    ^~~
```

[1] 一番ややこしいです.

```
                                        %s
1 warning generated.
ln -s ../bin/rc.repl repl
rm -f dot.bashrc
echo "# DO NOT EDIT THIS FILE" >dot.bashrc
./repl bash <bashrc >>dot.bashrc
sh: pstoimg: command not found
sh: pstoimg: command not found
rm -f dot.cshrc
echo "# DO NOT EDIT THIS FILE" >dot.cshrc
./repl csh <cshrc >>dot.cshrc
sh: pstoimg: command not found
sh: pstoimg: command not found
./gen-shell-scripts
install openxm ../bin
touch .done_gen-shell-scripts
mkdir -p ~/bin
cp ../bin/openxm ~/bin
if type "openxm" >/dev/null 2>&1; then \
        echo "openxm is already in the path."; \
    else \
        echo "export PATH=~/bin:$PATH" >>~/.bash_profile ; \
        echo "~/bin is added to the path."; \
    fi
~/bin is added to the path.
~/bin/openxm use-asir-mode.sh --local-yes
Installation completed. Login again or do source ./dot.bashrc to start emacs/asir
lib-shizen-a-018:rc takayama$ make norc-fix
pushd ../bin ; mv openxm openxm.orig ; sed -e 's/ASIR_CONFIG/#ASIR_CONFIG/g' openxm.orig >openxm ; chmod +x openxm ; popd
~/Desktop/OpenXM/bin ~/Desktop/OpenXM/rc
~/Desktop/OpenXM/rc
cp ../bin/openxm ~/bin
lib-shizen-a-018:rc takayama$
```

索 引