Fortran 90/95 プログラミング I

目次

1　はじめに

Fortran は FORmula TRANslation の略で，数値計算に適したプログラミング言語である．C 言語, オブジェクト指向，MATLAB の出現によってプログラミング言語が大きく進展している中，FORTRAN 言語においても，数値計算のためのプログラム開発が効率的に行え，より高速な計算処理を目指した様々な機能拡張・整備・強化が長年なされてきた．ここでは，Fortran 90, 95 について簡単に説明する．

1.1　Fortran 90

1.1.1　Fortran 90 の新しい機能

- 自由形式（Free source form）：「2. コンパイル」参照．
- 配列演算など：「9. 配列」参照．
  - 全体配列（whole arrays），部分配列（array sections）の演算
  - 配列組込み関数，配列に関する新機能
- 再帰（recursion）：「13. 外部関数副プログラム」参照．
- 新しい形式のプログラム単位モジュール（Modules）：「16. モジュール」参照．

- 総称ユーザー定義手続（generic user-defined procedures）：「16. モジュール」参照.
- インタフェースブロック（interface blocks）：「14.2. インタフェースモジュール」参照.
- ユーザー定義（構造）データ型と演算子（operators）
- ポインタ（pointer）
- 拡張性と冗長性

1.1.2 Fortran 90 の改善された機能

- ソーステキスト（source text）の追加機能：「2. コンパイル」参照.
  - 小文字（lowercase characters）の使用
  - セミコロン (;) で区切って，1 つのプログラム行に複数の文を記述.
  - Fortran 文字集合に新たな文字が追加
  - 名前の長さの上限が 31 文字（下線 (_) を含む）
- 数値計算のための機能改善：「4. データ型」参照.
  - 組込みデータ型の種別パラメタ（kind type parameter）

  - 新しい組込み関数など
- 制御構文の追加：「7. 条件分岐 (IF)，選択分岐 (CASE)」，「8. 繰り返し計算 (DO)」参照.
  - `CASE` 構文（construct）
  - `EXIT` 文，`CYCLE` 文
  - `WHILE`
  - 構文名
- 組込み手続（intrinsic procedures）の追加：「11. 配列組込み関数」参照.
- 省略可能な手続引数（optional procedure arguments）：「13. 外部関数副プログラム」参照.
- 宣言文（specification statements）の追加
  - `INTENT` 文，`OPTIONAL` 文
  - Fortran 90 `POINTER` 文
  - `PUBLIC` 文，`PRIVATE` 文
  - `TARGET` 文
- 属性の宣言方法の追加
  - `PARAMETER` 属性，`SAVE` 属性

  - INTRINSIC 属性など
- 入出力（input/output）機能の追加
  - OPEN 文（statement）のキーワード追加
  - INQUIRE 文のキーワード追加
- 有効範囲と結合

1.2 Fortran 95

1.2.1 Fortran 95 の新しい機能

- CPU_TIME 組込みサブルーチン（intrinsic subroutine）
- NULL 組込み関数（intrinsic function）
- FORALL 文（statement），FORALL 構文（construct）：「10. 配列選別代入」参照.
- PURE ユーザー定義手続（user-defined procedures）：「13. 外部関数副プログラム」参照.
- ELEMENTAL ユーザー定義手続（user-defined procedures）：「13. 外部関数副プログラム」参照.

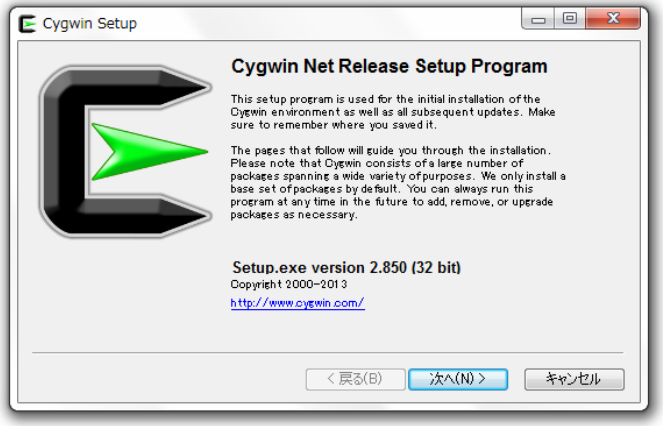
1.2.2 Fortran 95 の改善された機能

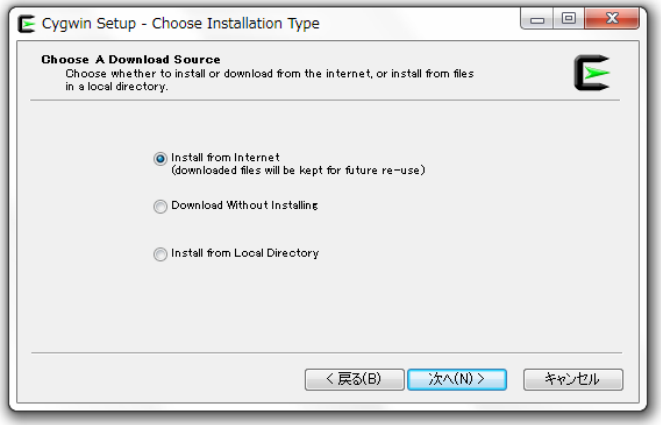
- 変数群入力中で使用可能な注釈
- 拡張された SIGN 組込み関数：「5. スカラ組込み関数」参照.
- 拡張された CEILING 組込み関数, FLOOR 組込み関数
- 拡張された MAXLOC 組込み関数, MINLOC 組込み関数：「11. 配列組込み関数」参照.
- 割付け配列（allocatable arrays）の自動割付け解除（automatic deallocation）：「9. 配列」参照.
- 拡張された WHERE 構文（construct）：「10. 配列選別代入」参照.
  - 入れ子の WHERE 構文と選別 ELSEWHERE 文
  - WHERE 構文名
- END INTERFACE 文で使用可能な総称識別子
- 構造型（derived-type）構造の暗黙的初期値指定
- ポインタ（pointer）の初期値
- -0.0 の印字
- 長さゼロの書式

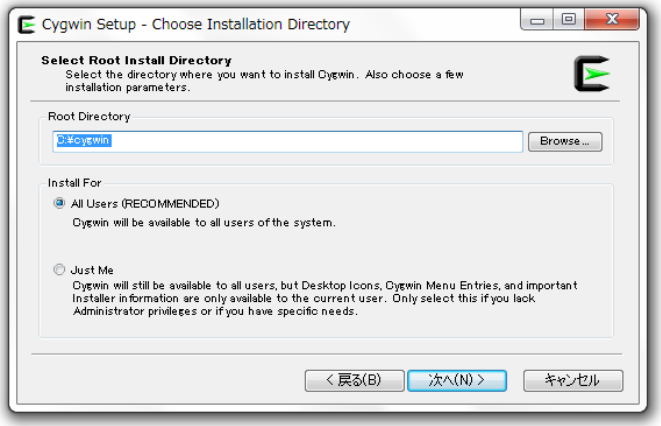
2　コンパイル

2.1　コンパイラーのインストール

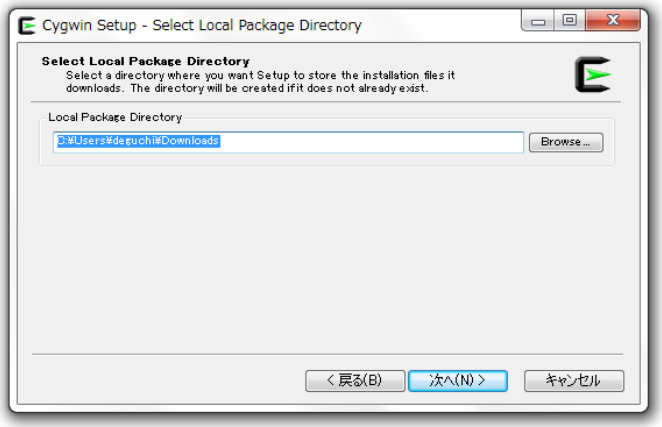
■GNU Fortran（Cygwin Terminal）　`setup-x86.exe` を実行すると，Linux 環境を Windows にインストールすることができ，Fortran 90/95 のコンパイラができるようになる．ここでは用いないが，C 言語はもちろんのこと，その他プログラミング言語全般の開発環境を構成することもできる．

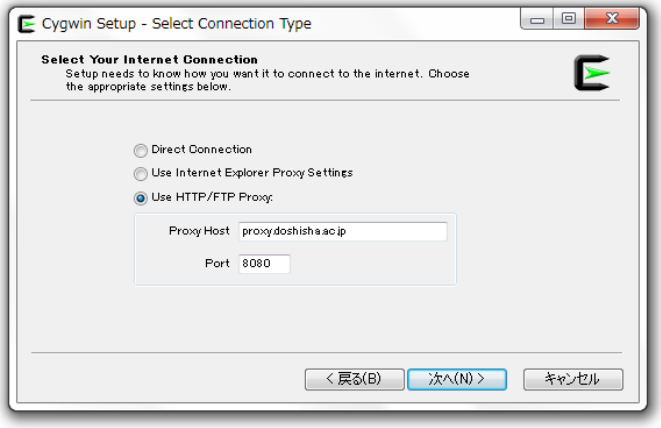


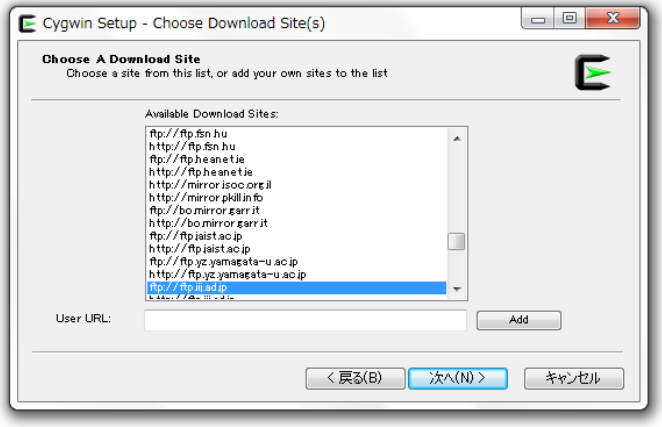




ダンロード先を指定する．

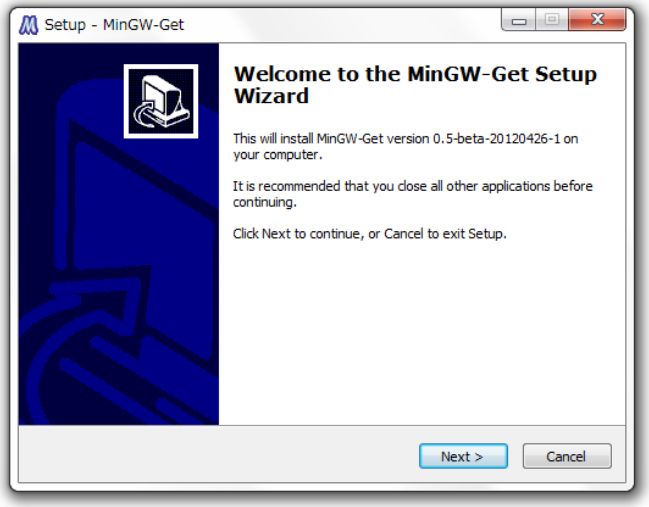


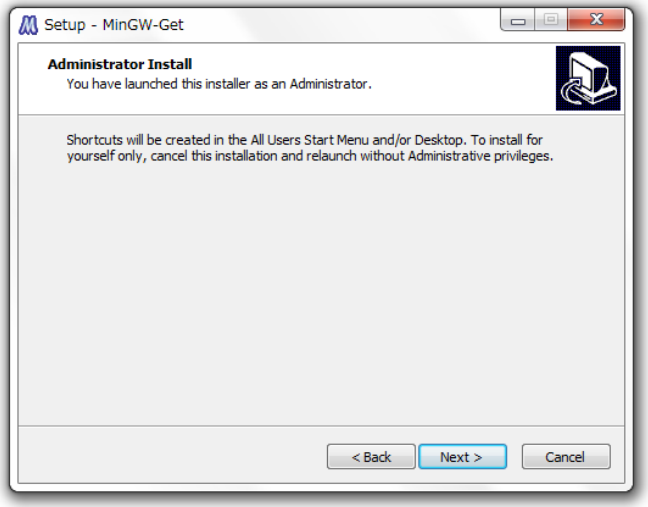


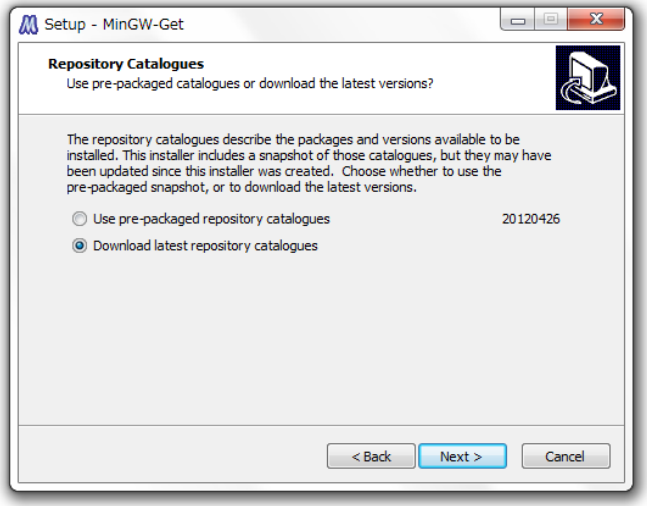


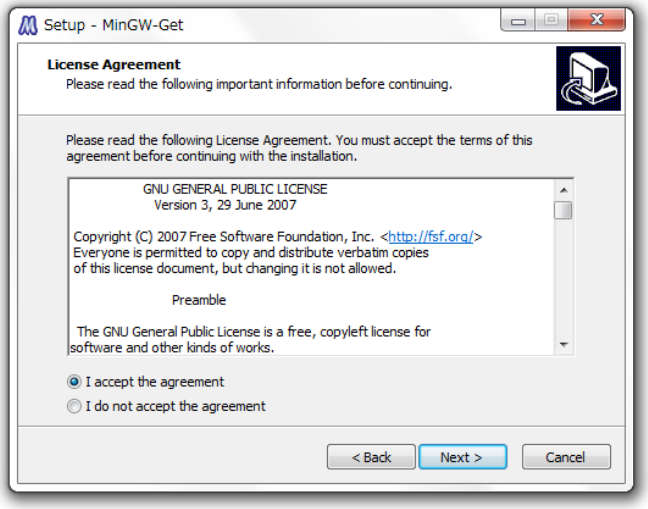
以降，省略.

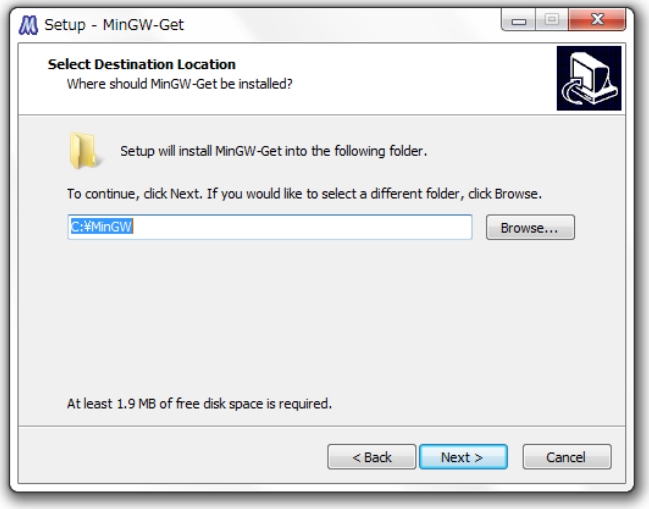
■GNU Fortran（MinGW Shell） `mingw-get-inst-20120426.exe` を実行すると，GNU Fortran 環境のみをインストールできる.

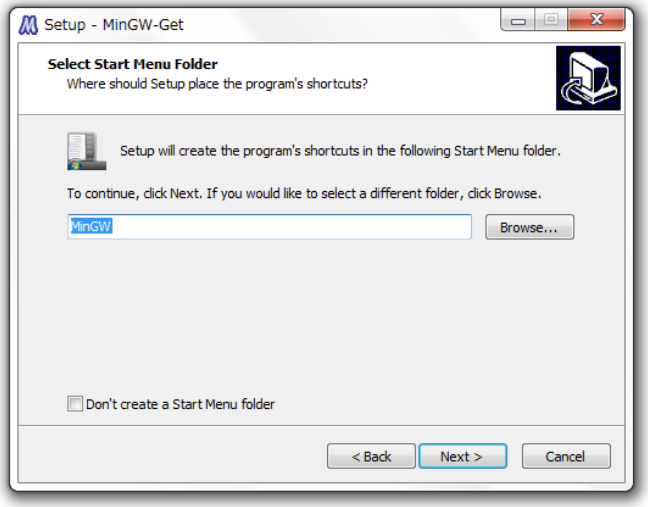




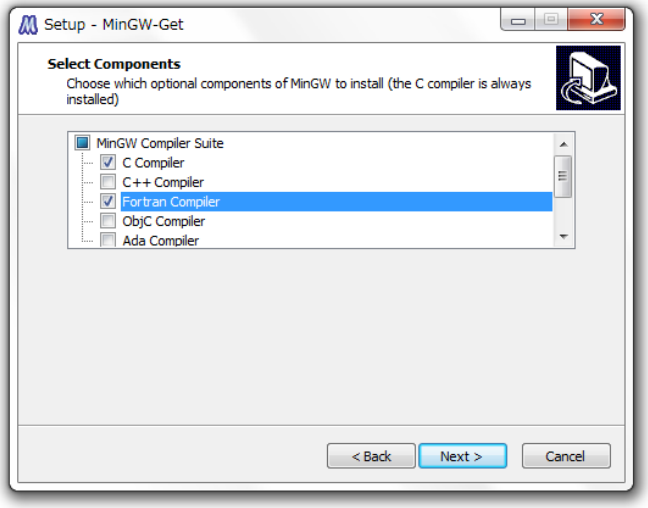


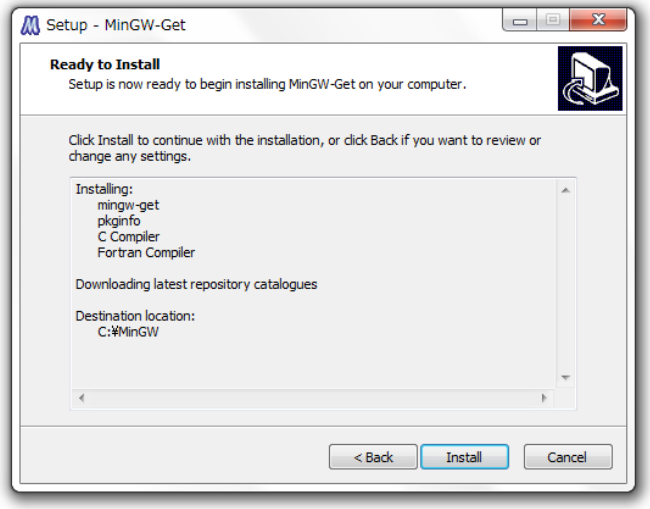


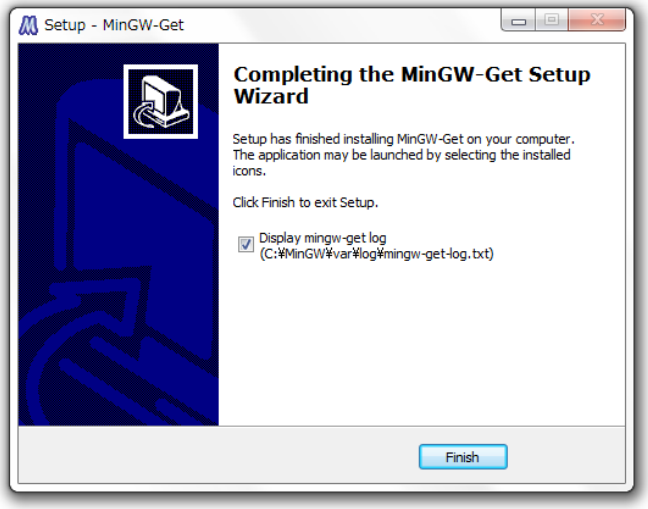




その他のプログラミング言語の開発環境を構成することもできる.







以降, 省略.

2.2　コマンド行でのプログラムのコンパイル，リンク，実行

ここでは，Windows での実行をとりあげ説明する.

■GNU Fortran（Cygwin Terminal），（MinGW Shell）　コンパイル，リンク：

```
gfortran -o 実行ファイル名 ソースファイル名
```

実行（拡張子 exe は省略）：

```
./実行ファイル名
```

あるいは，実行ファイル名を省略すると，実行ファイル名は a.exe となる.

```
gfortran ソースファイル名
./a
```

これを 1 行で次のように実行することもできる.

```
gfortran ソースファイル名; ./a
```

なお，GNU Fortran（Cygwin Terminal）の場合，実行には `cygwin1.dll` が必要となる．

■Visual Fortran（Command Prompt）　コンパイル，リンク：

```
ifort -o 実行ファイル名 ソースファイル名
```

実行（拡張子 exe は省略）：

```
実行ファイル名
```

あるいは，実行ファイル名を省略すると，実行ファイル名はソースファイル名の拡張子を exe としたファイル名となる.

```
ifort ソースファイル名
ソースファイル名
```

なお，NAG Fortran の場合，コマンド名は nagfor.

2.3　コマンド行での操作について

■GNU Fortran（Cygwin Terminal），（MinGW Shell）　設定によっては，Emacs のキー操作を一部に使用できる．よく使うコマンドは次のとおり．

- `pwd`：　ディレクトリ名を表示する．
- `cd`：　ディレクトリを変える．
- `ls`：　カレントディレクトリ内のファイル名を表示する．
- `cat`：　テキストファイルの内容を表示する．

コマンドを";"で区切れば，複数のコマンドを 1 行で実行できる．また，パイプを用いれば，1 行の実行で 1 つのコマンドよる出力を次の入力に受け渡すことができる．

■Visual Fortran（Windows の通常の Command Prompt）

- `cd`：　ディレクトリを変える．
- `dir`：　ディレクトリ名，ファイル名を表示する．
- `type`：　テキストファイルの内容を表示する．

2.4　主プログラム

■最初のプログラム　画面に出力するプログラムのソースコードを示すと，例えば次のようなる（f1_hello_print.f90）.

```
program main
  print *,'hello,  world' ! 標準出力
end program main
```

コンパイル，リンク，実行すると，

プログラムには，必ず一つの主プログラムが必要となる.

■PRINT 文（PRINT statement）　標準出力専用．フリーフォーマットの場合，次のように*とする（一般的な用法は後述）．

```
PRINT *, io-list
```

（互換性：FORTRAN 77〜）

例えば，変数を出力する場合，

```
PRINT *, var-1, var-2, var-3, .....
```

（互換性：FORTRAN 77〜）

- *io-list*：出力する変数名，式，文字列を並べた出力並び（output list）．
- *var-i (i=1,2,3, ...)*：変数名（variable name）

■PROGRAM 文（PROGRAM statement），END 文（END statement）　主プログラムの最初に PROGRAM 文，最後に END PROGRAM 文を書く．

```
PROGRAM プログラム名
    .....
    .....
END [PROGRAM プログラム名]
```

（互換性：FORTRAN 77～）

なお，PROGRAM 文は次のように省略することができ，最後に END 文を書く．

```
    .....
    .....
END
```

（互換性：FORTRAN 77～）

したがって，次のようなプログラムでもよい．

```
print *,'hello,  world'
end
```

2.5　ソースコードの形式

■自由形式（free source form）　（互換性：Fortran 90～）

- 第 1～132 カラムに記述する.
- 感嘆符（exclamation mark）！ に続く文は，行末までコメント文になる.
- 行末に アンド記号（ampersand character）`&`をつけると，行を継続できる．次の行の文頭に アンド記号`&`を付けると，前の文末と連結できる（省略可）.
- セミコロン（semicolon）`;`を文末につけると，1 行に複数の文を書くことができる.
- 拡張子は `f90` とする（GNU Fortran ではコン パイル・オプション`-ffree-form`）.

■固定形式（fixed source form）　（互換性：FORTRAN 77～）

- 第 1 カラム：　`c`，`C`，`*`，！ の文字を入れると，コメント行になる.
- 第 1～5 カラム：　文番号（statement label）1-99999.
- 第 6 カラム：　継続行を示す文字（`0`，スペース以外の文字）を入れる.
- 第 7～72 カラム：　通常のプログラムコードを書く.
- 拡張子は `f` とする（GNU Fortran ではコン パイル・オプション`-ffixed-form`）.

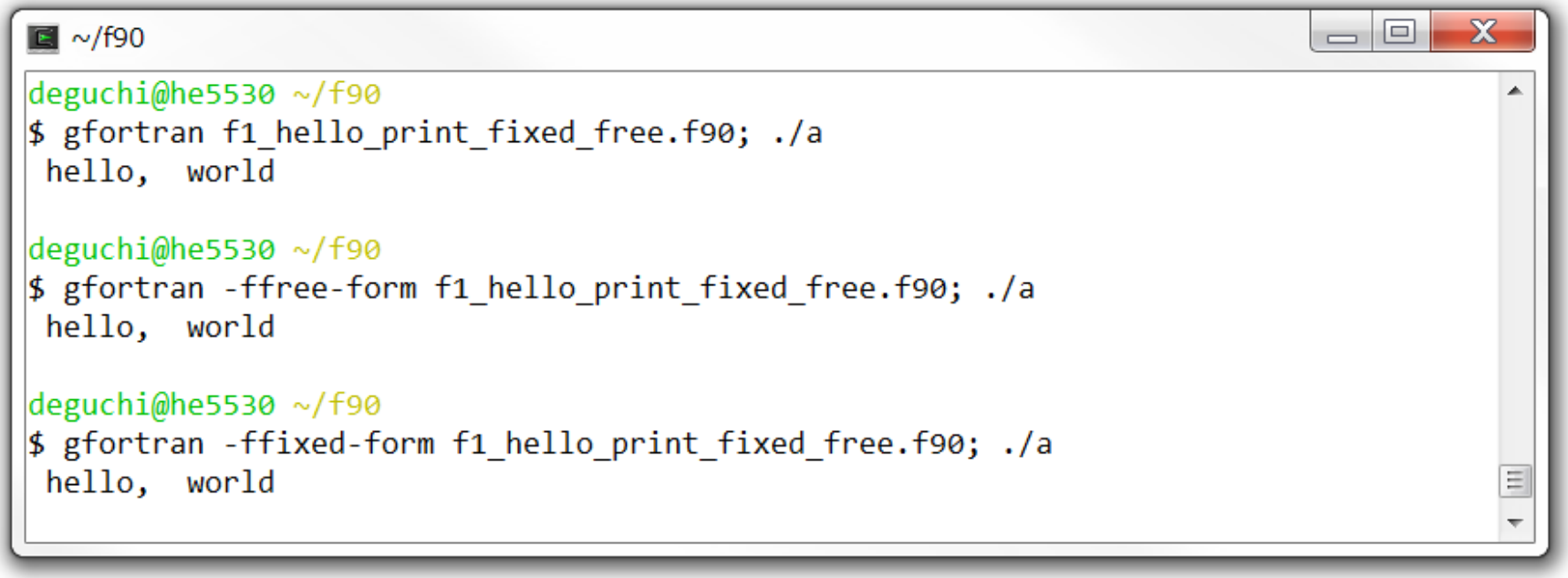
■両方の形式を満足するプログラム例

- コメント文を示す文字には，感嘆符 ！ を用いる.
- 継続行を示す文字には，アンド記号 & を用い，第 73〜132 カラム，2 行目以降には第 6 カラムにも記述する.
- 第 1 カラム：　！ の文字，あるいは文番号のみに使用する.
- 第 1〜5 カラム：　文番号（1-99999 の整数値）.
- 第 7〜72 カラム：　ソースコードを記述する.

このようにソースコードを記述すれば，両方の形式を満足するので，ソースコードのファイル名の拡張子は，f，f90 のどちらでもよい（f1_hello_print_fixed_free.f90）

```
!000000001111111111222222222233333333334444444445555555555666666666777
!234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234
      program main
      print *,'hello,',                                                  &
     &         '  world'
      stop ! 省略可
      end program main
```

コンパイル，リンク，実行すると，



ただし，FORTRAN 77，Fortran 90/95 の互換性を前提としている.

3　入出力の概要

■WRITE 文（WRITE statement）　値や文字の出力．フリーフォーマットの場合，次のように*とする．PRINT 文とは異なり，出力先を指定できる（一般的な用法は後述）．

```
WRITE( unit ,*)  io-list
```

（互換性：FORTRAN 77〜）

標準出力の場合，*unit* を*（あるいは 6）とし，例えば次のようになる．

```
WRITE(*,*)  'char-1', var-1, expr-1, 'char-2', var-2, expr-2, .....
```

（互換性：FORTRAN 77〜）

- *unit*：装置指定子（unit specifier）．符号無整数．
- *io-list*：出力する変数名，式，文字列を並べた出力並び（output list）．文字列の場合，アポストロフィ（apostrophe）' あるいは引用符（quotation marks）" で囲む．
- *char-i (i=1,2, ...)*：文字列
- *var-i (i=1,2, ...)*：変数名
- *expr-i (i=1,2, ...)*：式

■STOP 文（STOP statement）　プログラムの実行終了（*[ ]* は省略可を示す）．ただし，STOP 文の次に END 文がある場合，STOP 文は省略できる．

```
STOP [stop-code]
```

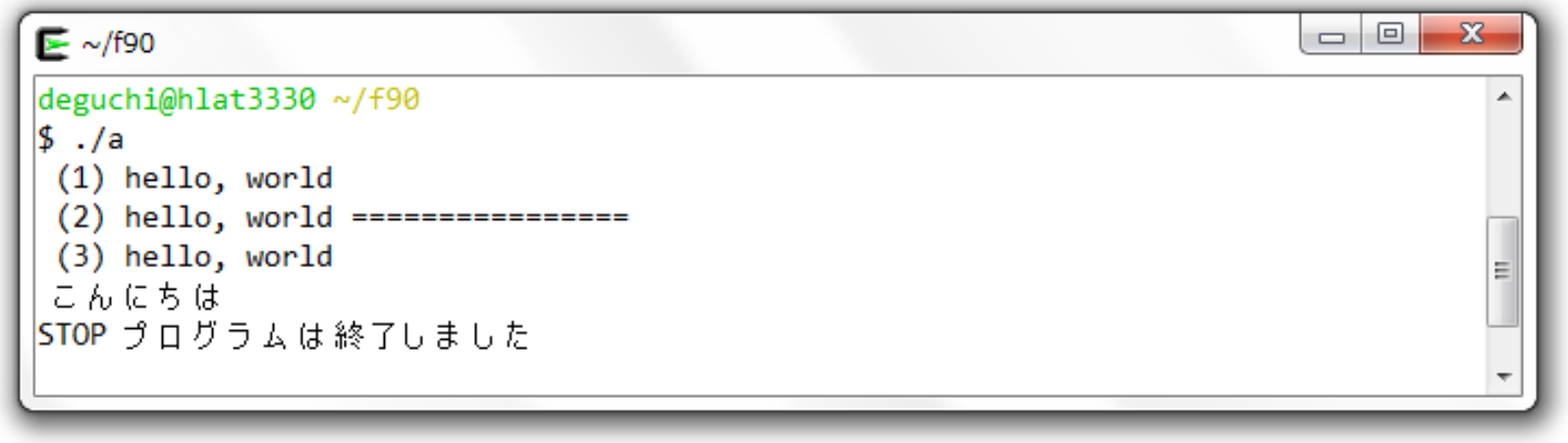
（互換性：FORTRAN 77～）

- *stop-code*：符号無整数（最大 5 桁まで），あるいはアポストロフィ（apostrophe）’で囲んだ文字列（省略可）．

■出力に関するプログラム例　(f1_write.f90).

```
program f1_write                          ! コメント文は日本語でもよい.
  write(6,*) '(1) hello, world'           ! 標準出力の装置番号は 6 あるいは*.
  write(*,*) "(2) hello, world",&         ! 出力する文字をアポストロフィー'
             ' ================'           ! あるいは引用符"で囲む.
  print *, '(3) hello, world'             ! 標準出力用の PRINT 文
  write(7,*) '(4) hello, world'           ! 装置番号を変えると，ファイル出力.
  write(6,*) 'こんにちは' ! 日本語も出力できる
  stop 'プログラムは終了しました' ! STOP 文による出力は，標準エラー出力.
end program f1_write                      ! Fortran の文は半角.
```

実行すると,



■READ 文（READ statement）　値の読み込み（ファイルからの入力等，詳細は後述）．フリーフォーマットの場合，次のように*を用いる．

```
READ( unit ,*) io-list
```

（互換性：FORTRAN 77〜）

標準入力の場合，*unit* を*（あるいは 5）とし，例えば次のようになる．

```
READ(*,*) var-1, var-2, .....
```

（互換性：FORTRAN 77〜）

- *unit*：装置指定子（unit specifier）．符号無整数．
- *io-list*：入力する変数名を並べた入力並び（input list）．
- *var-i (i=1,2, ...)*：変数名

または，PRINT 文のように次のような記述でもよい．

```
READ *, io-list
```

（互換性：FORTRAN 77〜）

■リダイレクション（redirection）　通常，標準入力はキーボードで，標準出力はディスプレイとなっているが，これを変える指示（リダイレクト指示）を行うことができる.

- <：　標準入力をファイル指定する.
- >：　標準出力をファイル指定する.
- 2>：　標準エラー出力をファイル指定する.
- »：　標準出力先に既存のファイルを指定し，追加の書き込みをする.

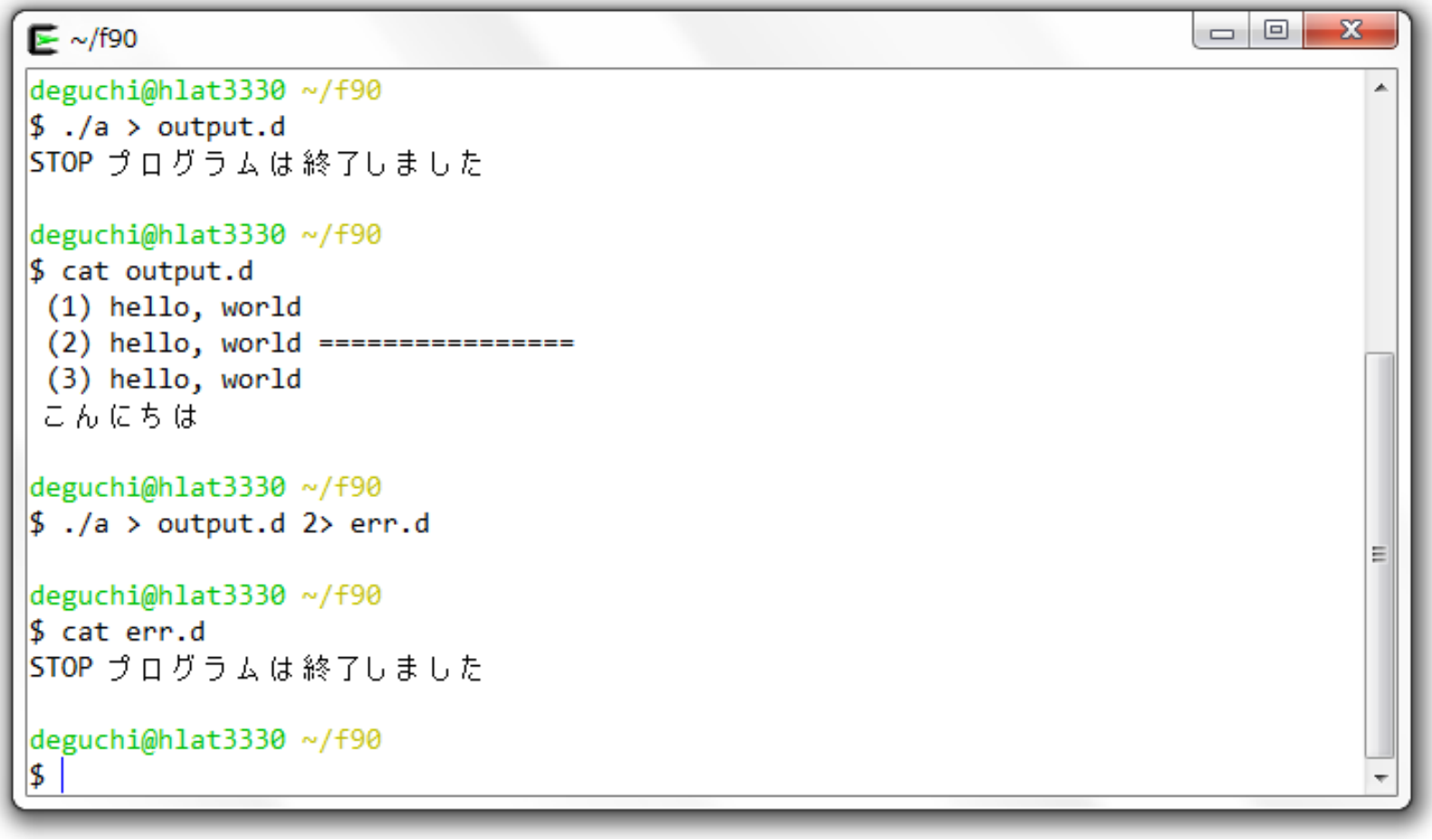
例えば，コマンド行でプログラムを実行する際，次のようにすると，ファイルから入力し，ファイルへと出力できる（ファイルがなければ作成される）.

```
実行プログラム名 < 入力ファイル > 出力ファイル 2> エラー出力ファイル
```

ファイルがすでに存在し，そのファイルの最後に追加して書き込む場合は，

```
実行プログラム名 >> 出力ファイル（既存のファイル）
```

■リダイレクションによるファイル出力の例　プログラム f1_write をリダイレクションを用いて実行すると次のようになる.



まず，最初の実行例では，プログラム f1_write の実行の際，リダイレクションにより標準出力をファイル output.d に出力している．そして，次の実行例では，さらに，標準エラー出力もファイル err.d に出力している.

4　データ型

■変数名（variable name）

- 最初の 1 文字は必ず英字（アルファベット）.
- 使用できる文字は，半角のアルファベット（alphabet）A-Z, a-z，数字（digits）0-9, アンダースコア（underscore）_ に限られている.
- 文字の数は Fortran 90/95 では半角 31 文字まで，Fortran 2003 では半角 63 文字まで.
- 大文字（uppercase letters）と小文字（lowercase letters）は区別されない（C 言語とは異なる）.

■組込みデータ型（intrinsic data types）

- 整数型（integer data type）：　`INTEGER`
- （単精度）実数型（real data type）：　`REAL`
- 倍精度実数型：　`DOUBLE PRECISION, REAL(8), REAL*8`
- （単精度）複素数型（complex data type）：　`COMPLEX`

- 倍精度複素数型： `COMPLEX(kind(0d0))`, `COMPLEX(8)`, `COMPLEX*16`
- 論理型（logical data type）： `LOGICAL`
- 文字型（character data type）： `CHARACTER`

■暗黙の型宣言

- 頭文字が `i-n` のとき整数型.
- 頭文字が `a-h`, `o-z` のとき単精度実数型.

■`IMPLICIT` 文　データの頭文字を指定して型を宣言できる．FORTRAN 77 ではよく用いられてきたもので，例えば，倍精度での計算プログラムでは,

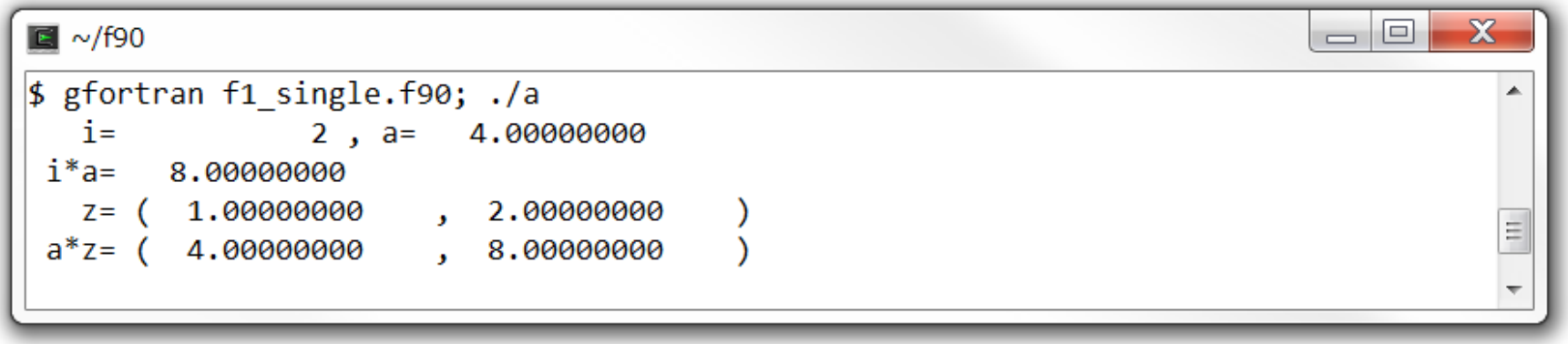
- 整数型：　「暗黙の型宣言」を使用.
- 倍精度実数型：　`IMPLICIT REAL*8(a-h,o-y)`
- 倍精度複素数型：　`IMPLICIT COMPLEX*16(z)`

■`IMPLICIT NONE` 文　「暗黙の型宣言」を明示的に使わないことの宣言．Fortran 90/95 では，コンパイル時のチェック強化のために，強く推奨されている.

■データ型（単精度）に関するプログラム例 （f1_single.f90).

```
program f1_single
  implicit none
  integer i ! 整数型
  real a    ! 単精度実数型
  complex z ! 単精度複素数型
  i = 2; a = 4.0; z = (1.0,2.0)
  write(6,*) '  i=',i,', a=',a
  write(6,*) 'i*a=',i*a
  write(6,*) '  z=',z
  write(6,*) 'a*z=',a*z
end program f1_single
```

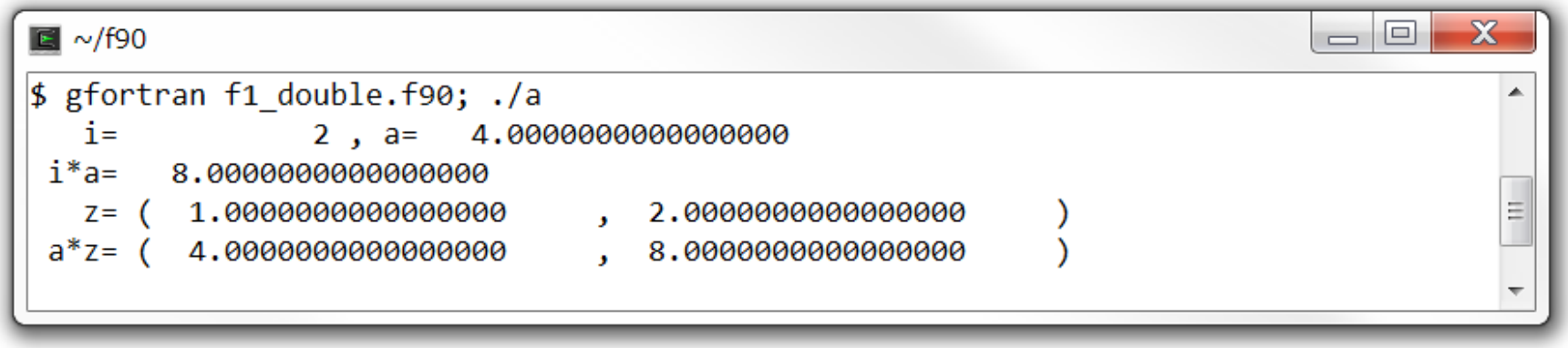
コンパイル，リンク，実行すると，



■データ型（倍精度）に関するプログラム例　f1_single.f90 を倍精度にすると次のようになる（f1_double.f90）.

```
program f1_double
  implicit none
  integer i     ! 整数型
  real(8) a     ! 倍精度実数型
  complex(8) z  ! 倍精度複素数型
  i = 2; a = 4.0D0; z = (1.0D0,2.0D0)
  write(6,*) '  i=',i,', a=',a
  write(6,*) 'i*a=',i*a
  write(6,*) '  z=',z
  write(6,*) 'a*z=',a*z
end program f1_double
```

コンパイル，リンク，実行すると，



5　スカラ組込み関数

5.1　数学関数（演算）

■引数が 1 つの組み込み関数　表 1 に引数が 1 つの組み込み関数を示す.

表 1　組み込み関数（引数の数は 1）

関数	引数の数	説明	通常の代数式
`sqrt(x)`	1	平方根	$\sqrt{x}$
`abs(x)`	1	絶対値	$\lvert x \rvert$
`sin(x), cos(x), tan(x)`	1	三角関数	$\sin x,\ \cos x,\ \tan x$
`sinh(x), cosh(x), tanh(x)`	1	双曲線関数	$\sinh x,\ \cosh x,\ \tanh x$
`asin(y)`	1	逆正弦関数	$\sin^{-1} y\ (-\pi/2 \le y \le \pi/2)$
`acos(y)`	1	逆余弦関数	$\cos^{-1} y\ (0 \le y \le \pi)$
`atan(y)`	1	逆正接関数	$\tan^{-1} y\ (-\pi/2 \le y \le \pi/2)$
`exp(x)`	1	指数関数	e^x
`log(x)`	1	自然対数（底 e）	$\log_e x = \ln x$
`log10(x)`	1	常用対数（底 10）	$\log_{10} x$
`conjg(z)`	1	共役複素数	z^*

ただし，*x*，*y* は実数，*z* は複素数を示す.

■組み込み関数（倍精度）を用いたプログラム例 （f1_double_math_a.f90).

```
program f1_double_math_a
  implicit none
  integer i,j ! 整数型
  real(8) a,b,e,pi ! 倍精度実数型
  pi = 3.14159265358979 3D0
  i = 2; a = i; j = a; b = -a
  write(6,*) ' 整数型 i =',i,', 実数型 a =',a
  write(6,*) ' 整数型 j =',j,', 実数型 b =',b
  write(6,*) ' 絶対値：abs(b) =',abs(b)
  write(6,*) ' 平方根：sqrt(a) =',sqrt(a)
  e = exp(1.0D0)
  write(6,*) 'exp(1) =',e,', 自然対数：log(e) =',log(e)
  write(6,*) ' 常用対数：log10(b) =',log10(a)
  a = sin(pi/2.0D0)
  write(6,*) 'sin(pi/2) = a =',a,'arcsin(a) =',asin(a)
  a = cos(0.0D0)
  write(6,*) 'cos(0) = a =',a,'arccos(a) =',acos(a)
  a = 3.0D0*pi/4.0D0
  b = tan(a)
  write(6,*) 'a =',a,', tan(3*pi/4) =',b
  write(6,*) 'arctan(b) =',atan(b)
```

```
  write(6,*) 'pi =',pi,4.0D0*atan(1.0D0)
end program f1_double_math_a
```

■引数が 2 つの組み込み関数　表 2 に引数が 2 つの組み込み関数を示す.

表 2　組み込み関数（引数の数は 2）

関数	引数の数	説明	通常の代数式
atan2(*s*, *c*)	2	逆正接関数	$\tan^{-1}(s/c)\ (-\pi \leq y \leq \pi)$
mod(*x*, *y*)	2	x/y の浮動小数点剰余	
max(*x*, *y*)	2	x, y の大きい方	
min(*x*, *y*)	2	x, y の小さい方	

ただし, *s*, *c* は実数, *x*, *y* は整数あるいは実数を示す.

■引数が 2 つの組み込み関数を用いたプログラム例　(f1_double_math_b.f90).

```
program f1_double_math_b
  implicit none
  real(8) a,b,pi ! 倍精度実数型
  pi = 3.14159265358979 3D0
  a = 3.0D0*pi/4.0D0
  b = tan(a)
  write(6,*) 'a =',a,', tan(3*pi/4) =',b
  write(6,*) 'arctan(b) =',atan(b),', atan2：',atan2(sin(a),cos(a))
  write(6,*) ' 剰余：mod(pi,1) =',mod(pi,1.0D0)
end program f1_double_math_b
```

■べき乗　x^y の計算は，x**y となる.

■整数乗，実数乗に関するプログラム例 （f1_double_math2.f90）.

```
program f1_double_math2
  implicit none
  integer ix,iy,iz ! 整数型
  real x,y,z ! 単精度実数型
  x = -1.0; ix = int(x); y =  2.0; iy = int(y)
  write(6,*) 'x =',x,', ix =',ix
  write(6,*) 'y =',y,', iy =',iy
  write(6,*) '整数（正）乗: ix**iy =',ix**iy,', x**iy =',x**iy
  write(6,*) '実数（正）乗: ix**y =',ix**y,', x**y =',x**y
  y = 0.0; iz = int(y)
  write(6,*) 'x =',x,', ix =',ix
  write(6,*) 'y =',y,', iy =',iy
  write(6,*) '整数乗: ix**iy =',ix**iy,', x**iy =',x**iy
  write(6,*) '実数乗: ix**y =',ix**y,', x**y =',x**y
  y = -2.0; iz = int(y)
  write(6,*) 'x =',x,', ix =',ix
  write(6,*) 'y =',y,', iy =',iy
  write(6,*) '整数（負）乗: ix**iy =',ix**iy,', x**iy =',x**iy
  write(6,*) '実数（負）乗: ix**y =',ix**y,', x**y =',x**y
  x = 0.0; ix = int(x); y =  2.0; iy = int(y)
  write(6,*) 'x =',x,', ix =',ix
```

```
  write(6,*) 'y =',y,', iy =',iy
  write(6,*) '整数（正）乗: ix**iy =',ix**iy,', x**iy =',x**iy
  write(6,*) '実数（正）乗: ix**y =',ix**y,', x**y =',x**y
  x = 1.0E-10; y = -2.0; iy = int(y)
  write(6,*) 'x =',x
  write(6,*) 'y =',y,', iy =',iy
  write(6,*) '整数（負）乗: x**iy =',x**iy
  write(6,*) '実数（負）乗: x**y =',x**y
end program f1_double_math2
```

5.2　型変換

■整数型への変換

- 実数 *real-var* →整数（小数点以下切り捨て）：　`INT(`*real-var*`)`
- 実数 *real-var* →整数（小数点以下四捨五入）：　`NINT(`*real-var*`)`
- 実数 *real-var* →整数（*real-var* を下回らない最少の整数値）：　`CEILING(`*real-var*`)`
- 実数 *real-var* →整数（*real-var* を超えない最大の整数値）：　`FLOOR(`*real-var*`)`

■単精度実数型への変換

- 整数 *int-var* →単精度実数：　`FLOAT(`*int-var*`)`，`REAL(`*int-var*`)`
- 複素数 *complex-var* → *complex-var* の実部（単精度）：　`REAL(`*complex-var*`)`
- 複素数 *complex-var* → *complex-var* の虚部（単精度）：　`AIMAG(`*complex-var*`)`

■倍精度実数型への変換

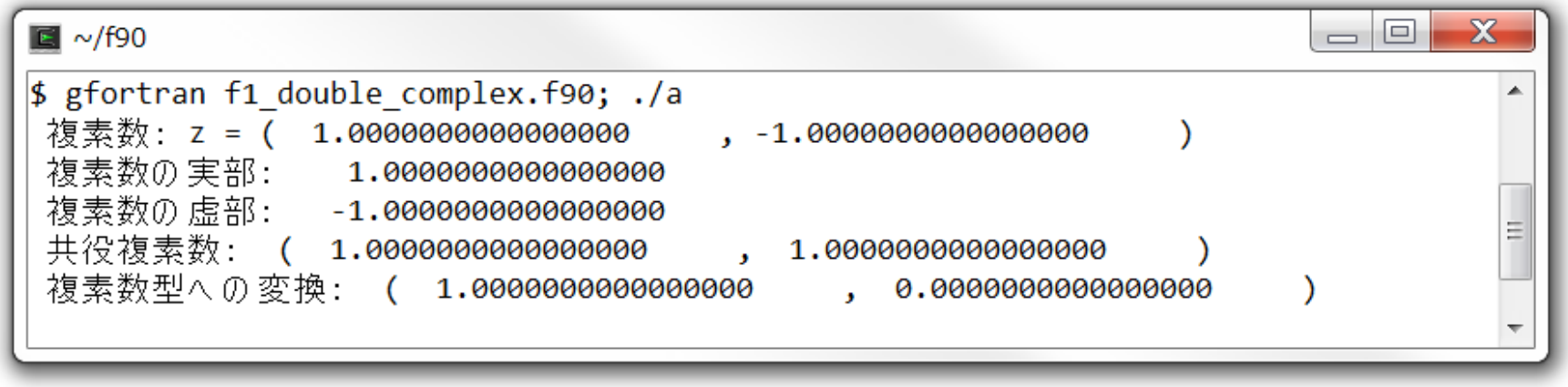
- 整数 *int-var* →倍精度実数：　`DBLE(`*int-var*`)`

- 実数 *real-var* →倍精度実数： DBLE(*real-var*)
- 複素数 *complex-var* → *complex-var* の実部（倍精度）： DBLE(*complex-var*)
- 複素数 *complex-var* → *complex-var* の虚部（倍精度）： DIMAG(*complex-var*)

■型変換に関するプログラム例 （f1_double_complex.f90）.

```
program f1_double_complex
! 型宣言 ------------------------------
  implicit none
  real(8) a,b ! 倍精度実数型
  complex(8) z ! 倍精度複素数型
! 計算，出力 ---------------------------
  z = (1.0D0,-1.0D0)
  write(6,*) ' 複素数：z =',z
  a = dble(z)
  write(6,*) ' 複素数の実部：',a
  b = dimag(z)
  write(6,*) ' 複素数の虚部：',b
  write(6,*) ' 共役複素数：',conjg(z)
  z = cmplx(a)
  write(6,*) ' 複素数型への変換：',z
end program f1_double_complex
```

コンパイル，リンク，実行すると，



6　書式付入出力

6.1　書式指定

■文番号を用いる FORMAT 文　入出力の書式を FORMAT 文により記述する．END 文より前であれば，FORMAT 文はどこに記述してもよいが，通常，END 文の直前に記述する．

```
READ(5, stmt-label ) input-list
stmt-label FORMAT( format-list )
```

（互換性：FORTRAN 77～）

```
WRITE(6, stmt-label ) output-list
stmt-label FORMAT( format-list )
```

（互換性：FORTRAN 77～）

- *stmt-label*：文番号（1～99999 のスカラ整数値）．
- *input-list*，*output-list*：入力および出力データ変数の並び
- *format-list*：書式制御文

■FORMAT 文を用いない書式指定　書式制御文を ’（ と ）’ で囲んで直接入出力文で指定する．文番号は不要．

```
READ(*,'( format-list )') input-list
WRITE(*,'( format-list )') output-list
```

（互換性：Fortran 90〜）

```
READ '( format-list )', input-list
PRINT '( format-list )', output-list
```

（互換性：Fortran 90〜）

6.2　編集記述子ほか

■書式制御文に用いる主なデータ編集記述子

- A 形：　文字，*n* 文字のとき，A*n*（*n* を省略すると，文字データの幅）
- I 形：　整数，*n* 桁のとき（符号含），I*n*
- F 形：　実数（固定小数点形式），指数部無 *n* 桁（符号含），少数点以下 *m* 桁のとき（$n \geq m + 2$），F*n.m*
- E 形：　実数（浮動小数点形式），指数部付 *n* 桁（符号含），少数点以下 *m* 桁のとき（$n \geq m + 7$），E*n.m*　（READ 文で用いる場合は，全体の *n* 桁のみが有効）
- D 形：　倍精度実数（浮動小数点形式），E 形と同様，D*n.m*
- G 形：　絶対値の大きさによって，F 形と E 形を使い分ける．G*n.m*
- ES 形：　実数（浮動小数点形式），仮数部の絶対値が 1 から 10 の間（理工表記指数），ES*n.m*

複数のとき，k 個であれば，k(...) と指定できる．ただし，*n*（記述欄の桁数），*m*, *k*（反復回数）は整数．

■その他の編集記述子

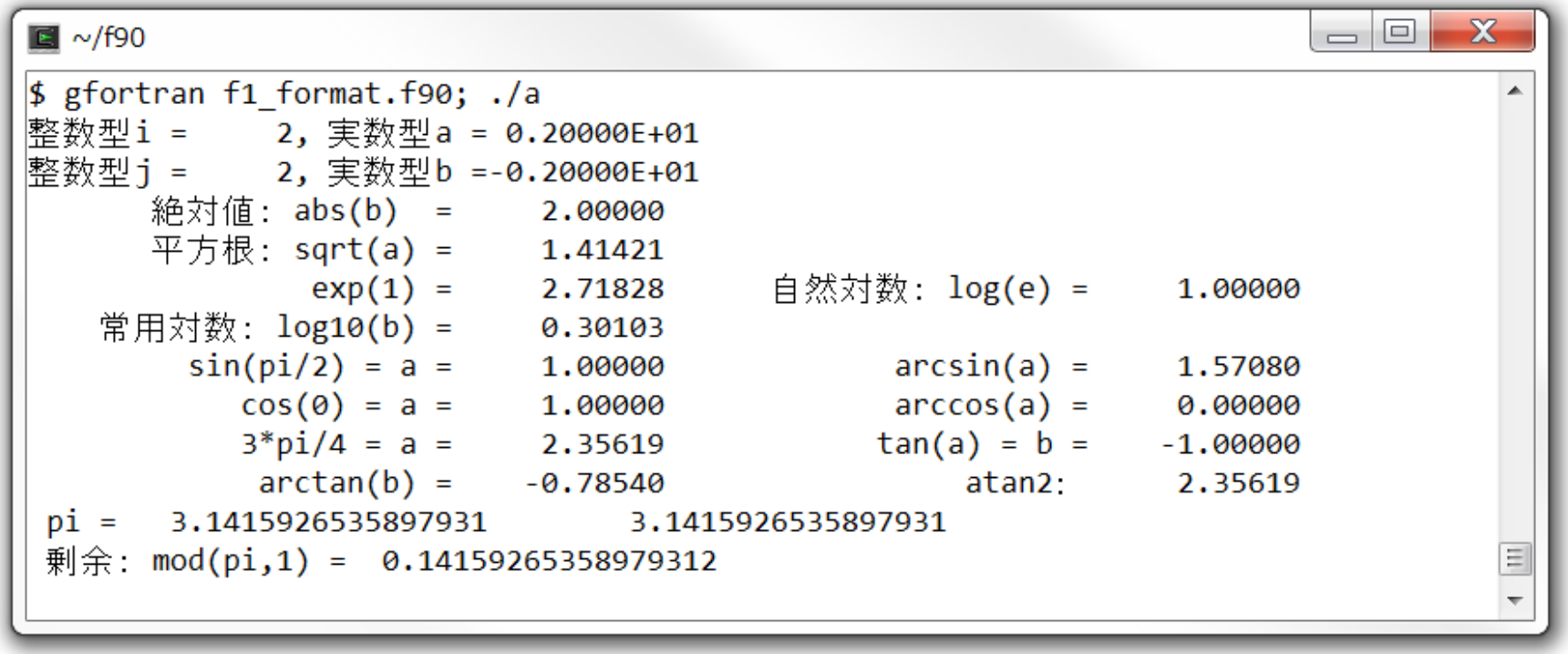
- X 形：　空白，n カラムのとき，nX（n を省略すると，$n = 1$）
- 複素数について：　実部，虚部を上のように各々指定すればよい．
- /：　改行
- advance='no'：　改行抑止
- ,$：　改行抑止

■書式付出力を用いたプログラム例　先に示したプログラム f1_double_math.f90 に書式を付けると（f1_format.f90），

```
program f1_format
  implicit none
  integer i,j ! 整数型
  real(8) a,b,e,pi ! 倍精度実数型
  pi = 3.14159265358979 3D0
  i = 2; a = i; j = a; b = -a
  write(6,'("整数型 i =",i6,", 実数型 a =",e12.5)') i,a
  write(6,600) j,b
  write(*,'(a24,f12.5)') ' 絶対値：abs(b)  =',abs(b)
```

```
  write(*,610) ' 平方根：sqrt(a) =',sqrt(a)
  e = exp(1.0D0)
  write(6,'(2(a24,f12.5))') 'exp(1) =',e,' 自然対数：log(e) =',log(e)
  write(6,'(a24,f12.5)') ' 常用対数：log10(b) =',log10(a)
  a = sin(pi/2.0D0)
  write(6,'(2(a24,f12.5))') 'sin(pi/2) = a =',a,'arcsin(a) =',asin(a)
  a = cos(0.0D0)
  write(6,'(2(a24,f12.5))') 'cos(0) = a =',a,'arccos(a) =',acos(a)
  a = 3.0D0*pi/4.0D0
  b = tan(a)
  write(6,'(2(a24,f12.5))') '3*pi/4 = a =',a,'tan(a) = b =',b
  write(6,'(2(a24,f12.5))') 'arctan(b) =',atan(b),'atan2：
',atan2(sin(a),cos(a))
  write(6,*) 'pi =',pi,4.0D0*atan(1.0D0)
  write(6,*) ' 剰余：mod(pi,1) =',mod(pi,1.0D0)
  600 format(' 整数型 j =',i6,', 実数型 b =',e12.5)
  610 format(a24,f12.5)
end program f1_format
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_format.f90; ./a
整数型i =   2, 実数型a = 0.20000E+01
整数型j =   2, 実数型b =-0.20000E+01
絶対値: abs(b) =   2.00000
平方根: sqrt(a) =   1.41421
exp(1) =   2.71828
自然対数: log(e) =   1.00000
常用対数: log10(b) =   0.30103
sin(pi/2) = a =   1.00000
arcsin(a) =   1.57080
cos(0) = a =   1.00000
arccos(a) =   0.00000
3*pi/4 = a =   2.35619
tan(a) = b =   -1.00000
arctan(b) =   -0.78540
atan2:   2.35619
pi =   3.1415926535897931
3.1415926535897931
剰余: mod(pi,1) = 0.14159265358979312

7　条件分岐（IF），選択分岐（CASE）

7.1　関係演算子，論理演算子

関係演算子（relational operator）（表 3 参照）を用いて，次のように 2 つの式を比較したものを関係式（relational expression）とよび，真（T），偽（F）を返す．

expr-1　関係演算子　*expr-2*

- *expr-1*，*expr-1*：数値式，文字式（文字の大小は辞書式順序による）

表 3　関係演算子

関係演算子（互換性） FORTRAN 77～	Fortran 90～	説明，通常の代数演算	例（*expr-1*，*expr-2*：式）
.EQ.	==	等しい（equal to），$=$	*expr-1* == *expr-2*
.NE.	/=	等しくない（not equal to），$\neq$	*expr-1* /= *expr-2*
.GT.	>	大きい（greater than），$>$	*expr-1* > *expr-2*
.LT.	<	小さい（less than），$<$	*expr-1* < *expr-2*
.GE.	>=	大きいか等しい（greater than or equal to），$\geq$	*expr-1* >= *expr-2*
.LE.	<=	小さいか等しい（less than or equal to），$\leq$	*expr-1* <= *expr-2*

また，論理演算子（logical operator）（表 4 参照）を用いて，真（T），偽（F）を返すこともできる（表 5 参照）.

単項論理演算子 *logical-expr-1*
logical-expr-1 2 項論理演算子 *logical-expr-2*

- *logical-expr-1*，*logical-expr-2*：関係式（真（T），偽（F））

表 4　論理演算子

論理演算子	説明	優先順位	例	互換性
.NOT.	否定	1	.NOT. *logical-expr-1*	FORTRAN 77〜
.AND.	論理積（かつ）	2	*logical-expr-1* .AND. *logical-expr-2*	FORTRAN 77〜
.OR.	論理和（または）	3	*logical-expr-1* .OR. *logical-expr-2*	FORTRAN 77〜
.EQV.	真と真あるいは偽と偽ならば真	4	*logical-expr-1* .EQV. *logical-expr-2*	Fortran 90〜
.NEQV.	真と偽ならば真	4	*logical-expr-1* .NEQV. *logical-expr-2*	Fortran 90〜

表 5　真理値表

logical-expr-1	*logical-expr-2*	.AND.	.OR.	.EQV.	.NEQV.
T	T	T	T	F	T
T	F	F	T	T	F
F	T	F	T	T	F
F	F	F	F	F	T

7.2 IF 文

IF 文（statement）は，論理（logical）IF 文 [FORTRAN 77] に対応するもので，条件を満たす場合に一つの実行文を処理する.

■Syntax

```
IF (scalar-logical-expr) action-stmt
```

（互換性：FORTRAN 77〜）

- *scalar-logical-expr*：スカラ論理式 n（真のとき，*action-stmt* が実行される）
- *action-stmt*：単一の実行文

複数の実行文を分岐処理する場合は，GOTO 文（使う必要がないので省略）ではなく，後述する IF 構文（construct）を用いる．その他に算術 IF 文があるが，廃止予定であるので，ここでは省略する.

■IF 文を用いたプログラム例 1　$|x| > 1$ のとき，$x = 0$ とする（f1_if_rect.f90）.

```
program f1_if_rect
  implicit none
  real(8) x ! 倍精度実数型
  read *,x
  if(abs(x)>1.0d0) x = 0.0D0
  print *,'x = ',x
end program f1_if_rect
```

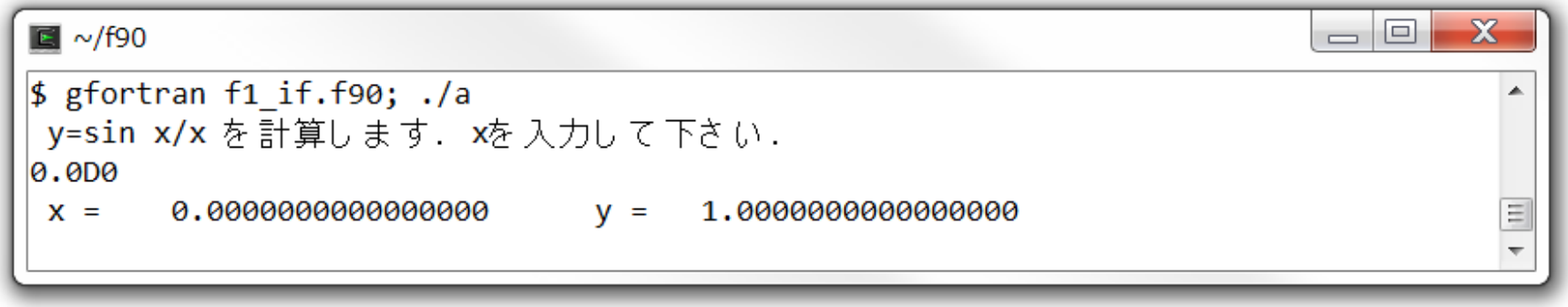
■IF 文を用いたプログラム例 2　$|x| \leq 0$ のとき，プログラムをストップさせる（f1_if_stop.f90）.

```
program f1_if_stop
  implicit none
  real(8) x ! 倍精度実数型
  read *,x
  if(x<=0.0d0) stop 'x<=0'
  print *,'x(>0) = ',x
end program f1_if_stop
```

■IF 文を用いたプログラム例 3　$y = \frac{\sin x}{x}$ の計算において，IF 文を用いてゼロ割を回避する（f1_if.f90）.

```
program f1_if
  implicit none
  real(8) x,y ! 倍精度実数型
  print *, 'y=sin x/x を計算します．x を入力して下さい．'
  read(5,*) x
! y = sin(x)/x の計算
  y = 1.0D0
  if(x/=0.0d0) y = sin(x)/x
  write(6,*) 'x = ',x,'y =',y
end program f1_if
```

ソースコード f1_if.f90 をコンパイル，リンク，実行すると，



7.3 IF 構文

7.3.1 構文名をもたない IF 構文

IF 構文（construct）は，ブロック IF 文 [FORTRAN 77] に対応するもので，多数の分岐ならびに複数の実行文（ブロック）を処理できる.

■Syntax　　[...] は省略可能の意.

```
IF (scalar-logical-expr-1) THEN
      block-1
[ELSE IF (scalar-logical-expr-2) THEN
      block-2]
[.....
     .....      ]
[ELSE
      block-N]
END IF
```

（互換性：FORTRAN 77〜）

- *scalar-logical-expr-n* (*n=1,2, ... ,N*)：スカラ論理式 n
- *block-n* (*n=1,2, ... ,N*)：ブロック n（単一あるいは複数の実行文）

■ブロック 1 のみ場合　論理 IF 文の処理と同様で次のようになる.

```
IF (scalar-logical-expr-1) THEN
    block-1
END IF
```

（互換性：FORTRAN 77〜）

これより，f1_if_rect.f90（$|x| > 1$ のとき，$x = 0$ とする）は，次のようになる (f1_endif_rect.f90).

```
program f1_endif_rect
  implicit none
  real(8) x ! 倍精度実数型
  read *,x
  if(abs(x)>1.0d0) then
     x = 0.0D0
  endif
  print *,'x = ',x
end program f1_endif_rect
```

■ブロック 1，N を処理する場合　ELSE を用いて（論理 IF 文も通常，このような処理を行う場合が多い），

```
IF (scalar-logical-expr-1) THEN
    block-1
ELSE
    block-N
END IF
```

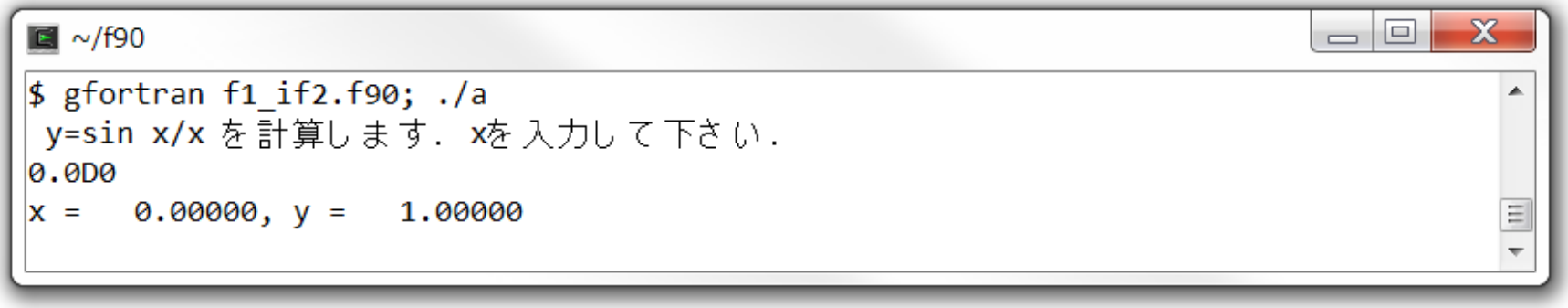
（互換性：FORTRAN 77～）

これより，f1_if.f90（ゼロ割回避）は，IF 構文を用いて次のようになる（f1_if2.f90）．ただし，書式付出力．

```
program f1_if2
  implicit none
  real(8) x,y ! 倍精度実数型
  print *, 'y=sin x/x を計算します．x を入力して下さい．'
  read(*,*) x
! y = sin(x)/x の計算
  if(x==0.0D0) then
     y = 1.0D0
  else
```

```
      y = sin(x)/x
   endif
   write(*,'("x =",f10.5,", y =",f10.5)') x,y
end program f1_if2
```

コンパイル，リンク，実行すると，



7.3.2　構文名をもつ IF 構文

if 構文名をつけた IF 構文では，先頭の IF THEN 文の構文名と，対応する END IF 文の構文名の両方が必須である.

■Syntax

```
[name:] IF(scalar-logical-expr-1) THEN
      block-1
[ELSE IF(scalar-logical-expr-2) THEN [[name]]
      block-2]
[.....
     .....      ]
[ELSE [[name]]
      block-N]
END IF [name]
```

（互換性：Fortran 90〜）

- *name*：if 構文名（省略可で上位互換）
- *scalar-logical-expr-n* (*n=1,2, ... ,N*)：スカラ論理式 n
- *block-n* (*n=1,2, ... ,N*)：ブロック n（単一あるいは複数の実行文）

■入れ子構造　IF の入れ子構造に対して，IF 構文名を用いた例を示すと次のようになる.

```
[name-1:] IF(scalar-logical-expr-1-1) THEN
      block-1-1
      [name-2:] IF(scalar-logical-expr-2-1) THEN
            block-2-1
      ELSE IF(scalar-logical-expr-2-2) THEN
            block-2-2
     .....
          .....
      END IF [name-2]
ELSE IF(scalar-logical-expr-1-2) THEN
      block-1-2
.....
     .....
ELSE
      block-1-N
END IF [name-1]
```

（互換性：Fortran 90～）

7.4 CASE 構文

CASE 構文（construct）は，多数の多段階分岐を場合式を用いて処理する．

■Syntax

```
[name:] SELECT CASE (case-expr)
CASE (case-selector-1) [[name]]
      block-1
[CASE (case-selector-2) [[name]]
      block-2]
[.....
     .....      ]
[CASE DEFAULT [[name]]
      block-n]
END SELECT [name]
```

（互換性：Fortran 90〜）

- *case-expr*：場合式（スカラ整数式，スカラ文字式，スカラ論理式）
- *case-selector-n*：場合値範囲並び
- *name*：case 構文名

SELECT CASE 文における場合式の値と CASE 文の場合値範囲並びが一致したとき，その CASE 文に続くブロックが実行される．場合式がどの場合値範囲並びにも一致しないとき，CASE DEFAULT のブロック N が実行される．ただし，case 構文名は省略してもよい．

■場合式としてスカラ整数式を用いたプログラム例 （f1_case_lecture.f90）．

```
program f1_case_lecture
  implicit none
  integer i ! 整数型
  read(5,*) i
  select case(i)
  case(1)
     print *,'9:00-10:30'
  case(2)
     print *,'10:45-12:15'
  case(3)
     print *,'13:10-14:40'
  case(4)
     print *,'14:55-16:25'
  case(5)
     print *,'16:40-18:10'
  case default
     print *,' その他'
```

```
  end select
end program f1_case_lecture
```

整数式のとき，場合値範囲並びは，（整数）のほかに，次のように（整数，整数，...），（下限:上限）等でもよい（f1_case_gpa.f90）.

```
program f1_case_gpa
  implicit none
  integer ip ! 整数型
  read(5,*) ip
  gpa: select case(ip)
  case(90:99,100)
     print *,'A'
  case(80:89)
     print *,'B'
  case(70:79)
     print *,'C'
  case(60:69)
     print *,'D'
  case(0,1:59)
     print *,'F'
  case default
     print *,'error'
```

```
  end select gpa
end program f1_case_gpa
```

■場合式としてスカラ文字式を用いたプログラム例 （f1_case_color.f90).

```
program f1_case_color
  implicit none
  character(len=6) c ! 文字型
  read(5,*) c
  color: select case(c)
  case('red')
     print *,'Stop now.'
  case('yellow')
     print *,'Prepare to stop.'
  case('green')
     print *,'Proceed through intersection.'
  case default
     print *,'Illegal color encountered.'
  end select color
end program f1_case_color
```

8　繰り返し計算（DO）

8.1　DO ループ構文（do 変数）

DO ループ構文は，計算回数を予め与えて繰り返し処理を行う．

8.1.1　文番号を用いない END DO 文

■Syntax

```
[name:] DO index = istart, iend [, incr]
    block
END DO [name]
```

（互換性：Fortran 90～）

- *index*：do 変数あるいはカウンター（整数型のスカラ変数）
- *istart, iend, incr*：初期値，終値，増分（増分を省略すると 1）
- *block*：ブロック（単一あるいは複数の実行文）
- *name*：do 構文名（省略可）

DO ループをぬけた後，do 変数はさらに増分が加わった値となる．

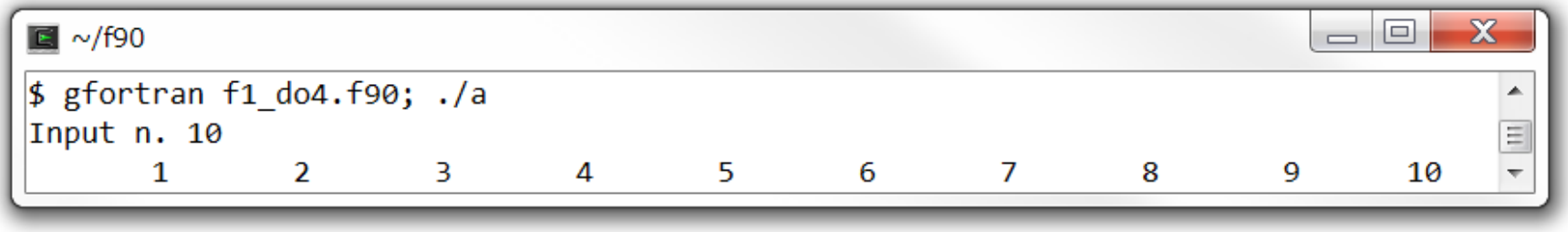
■END DO 文を用いたプログラム例　do 構文名を使用しない場合（f1_do3.f90),

```
program f1_do3
  implicit none
  integer i,n ! 整数型
  write(*,'(a)',advance='no') 'Input n. '
  read(*,*) n
  do i=1,n
     write(*,'(i8,$)') i ! 改行抑止
  enddo
end program f1_do3
```

do 構文名を使用した場合（f1_do4.f90),

```
program f1_do4
  implicit none
  integer i,n ! 整数型
  write(*,'(a)',advance='no') 'Input n. '
  read(*,*) n
  loop: do i=1,n
     write(*,'(i8,$)') i ! 改行抑止
  enddo loop
end program f1_do4
```

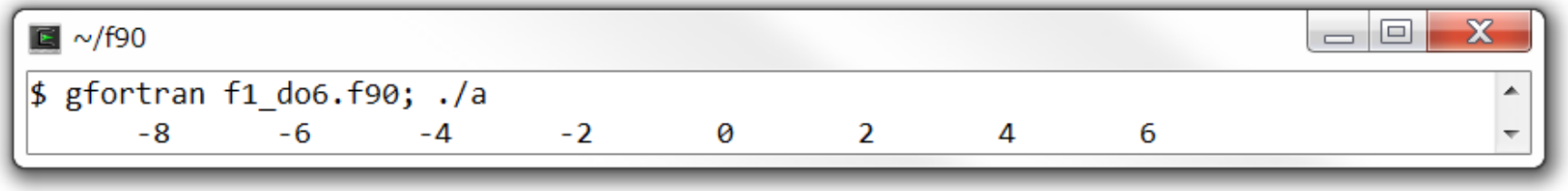
「暗黙の型宣言」を明示的に使わない宣言，書式指定でも文番号を使用しないスタイルとした．両者とも同じ結果となるもので，後者をコンパイル，リンク，実行すると，



■do 変数の初期値，終値，増分を設定したプログラム例 （f1_do6.f90）

```
program f1_do6
  implicit none
  integer i,n1,n2,n3 ! 整数型
  n1 = -8 ! 初期値
  n2 = 7 ! 終値
  n3 = 2 ! 増分
  do i=n1,n2,n3
     write(*,'(i8,$)') i ! 改行抑止
  enddo
end program f1_do6
```

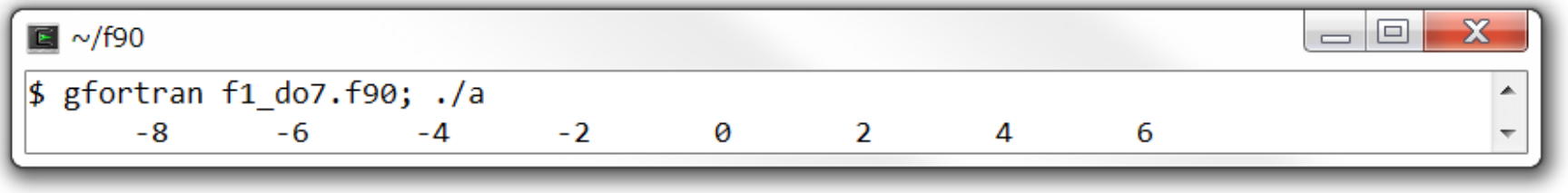
コンパイル，リンク，実行すると，



■do 変数をループカウンターとして用いたプログラム例 （f1_do7.f90）

```
program f1_do7
  implicit none
  integer i,n1,n3,n ! 整数型
  n1 = -8 ! 初期値
  n3 = 2 ! 増分
  n = 8 ! DO ループの繰り返し回数
  do i=1,n
     write(*,'(i8,$)') n1+n3*(i-1) ! 改行抑止
  enddo
end program f1_do7
```

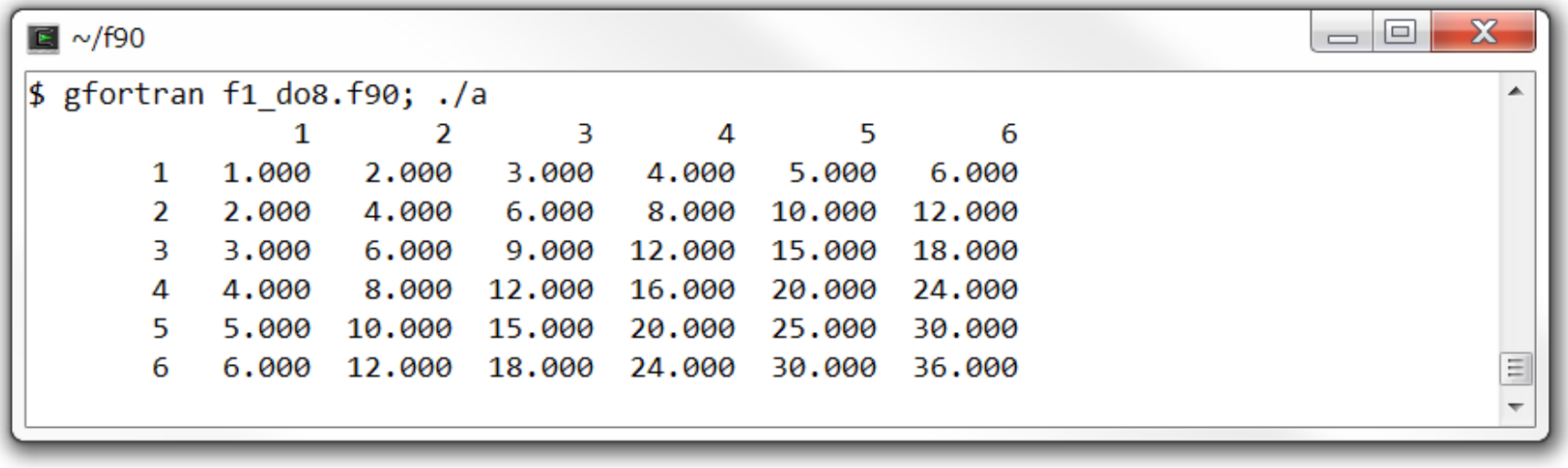
コンパイル，リンク，実行すると，



■2 重 DO ループを用いたプログラム例 （f1_do8.f90）

```
program f1_do8
  implicit none
  integer i,j,n ! 整数型
  real(8) a,a0,da,b,b0,db ! 倍精度実数型
  a0 = 1.0D0; da = 1.0D0
  b0 = 1.0D0; db = 1.0D0
  n = 6 ! DO ループの繰り返し回数
  write(*,'(8x,100i8)') (j,j=1,n)
  loop1: do i=1,n
     a = a0+da*(i-1)
     write(*,'(i8,$)') i ! 改行抑止
     loop2: do j=1,n
        b = b0+db*(j-1)
        write(*,'(f8.3,$)') a*b ! 改行抑止
     enddo loop2
     write(*,*)
  enddo loop1
end program f1_do8
```

コンパイル，リンク，実行すると，



8.1.2　文番号 DO 文

■Syntax　文番号 DO 文も，計算回数を予め与えて繰り返し処理を行うもので，CONTINUE 文を用いる場合，次のようになる．

```
DO stmt-label  index = istart, iend [, incr]
      block
stmt-label  CONTINUE
```

（互換性：FORTRAN 77～）

CONTINUE 文を用いない場合，

```
DO stmt-label  index = istart, iend [, incr]
      block
stmt-label  action-stmt
```

（互換性：FORTRAN 77～）

- *stmt-label*：文番号（1～99999 のスカラ整数値）
- *index*：do 変数（整数型のスカラ変数，ただし，FORTRAN 77 では実数型も可）
- *istart, iend, incr*：初期値，終値，増分（増分を省略すると 1）
- *block*：ブロック（単一あるいは複数の実行文）
- *action-stmt*：単一の実行文

また，END DO 文を用いる場合，

```
DO stmt-label  index = istart, iend [, incr]
      block
stmt-label END DO
```

（互換性：Fortran 90〜）

■文番号をつけたプログラム例　CONTINUE 文を用いた場合（f1_do.f90），

```
program f1_do\verb|
  write(6,*) 'Input n ? '
  read(5,*) n
  do 10 i=1,n
     write(6,600) i
10   continue
600  format(i8,$)
end program f1_do
```

実行文に文番号をつけた場合（f1_do2.f90），

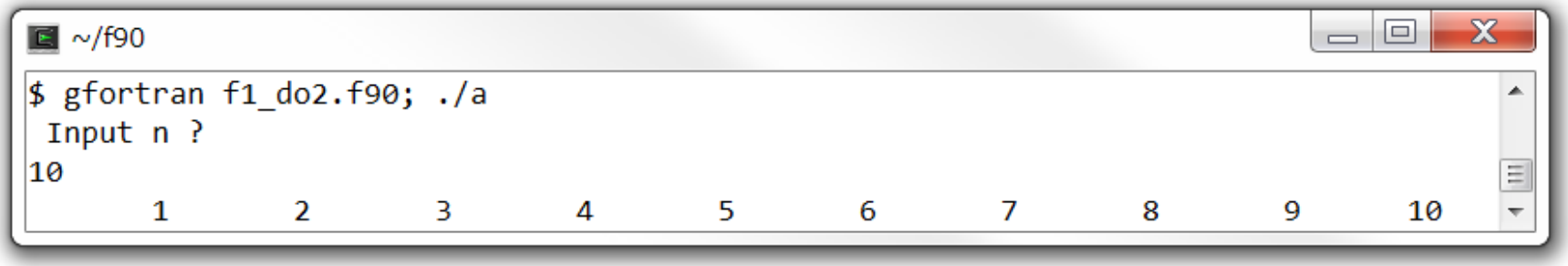
```
program f1_do2
  write(6,*) 'Input n ? '
```

```
  read(5,*) n
  do 10 i=1,n
10    write(6,600) i
600   format(i8,$)
end program f1_do2
```

END DO 文を用いた場合（f1_do1.f90），

```
program f1_do1
  write(6,*) 'Input n ? '
  read(5,*) n
  do 10 i=1,n
     write(6,600) i
10    enddo
600   format(i8,$)
end program f1_do1
```

いずれも同じ結果となり，f1_do2.f90 をコンパイル，リンク，実行すると，



8.2 DO 型並び

■Syntax　　単一ループの場合,

```
..... (expr,  index = istart, iend [, incr])
```

（互換性：FORTRAN 77～）

二重ループの場合,

```
..... ( (expr,  index-1 = istart-1, iend-1 [, incr-1])  index-2 = istart-2, iend-2 [, incr-2])
```

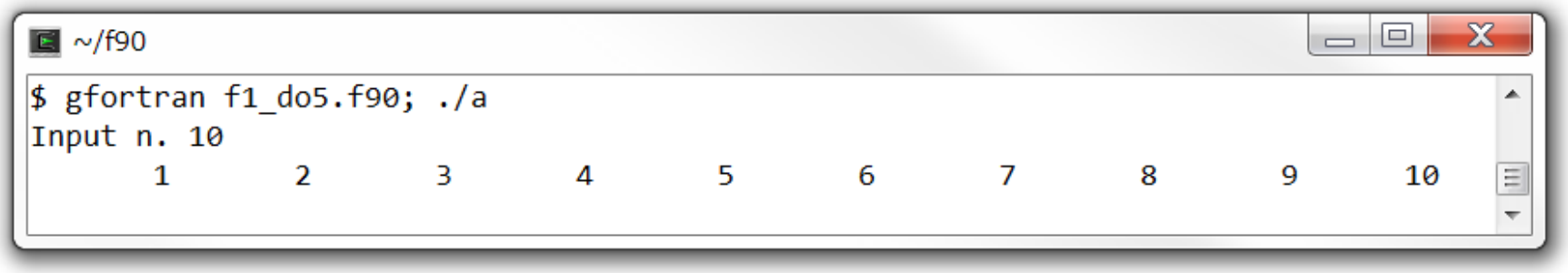
（互換性：FORTRAN 77～）

- *expr*：スカラ変数あるいはスカラ式
- *index*：do 変数あるいはカウンター（整数型のスカラ変数）
- *istart, iend, incr*：初期値，終値，増分（増分を省略すると 1）
- *istart-n, iend-n, incr-n (n=1,2)*：初期値-n，終値-n，増分-n（増分-n を省略すると 1）

■DO 型並びを用いたプログラム例　単一ループの場合（f1_do5.f90),

```
program f1_do5
  implicit none
  integer i,n ! 整数型
  write(*,'(a)',advance='no') 'Input n. '
  read(*,*) n
  write(*,'(100i8)') (i,i=1,n) ! DO 型並び
end program f1_do5
```

コンパイル，リンク，実行すると，



二重ループの場合（f1_do5.f90），

```
program f1_do5b
  implicit none
  integer i,j,n ! 整数型
  write(*,'(a)',advance='no') 'Input n. '
  read(*,*) n
  write(*,'(100i8)') ((i*j,i=1,n),j=1,n) ! DO 型並び
```

```
end program f1_do5b
```

8.3 不定回数の DO ループ

8.3.1 DO WHILE 文

DO WHILE 文は，不定回数のループを行う．

■Syntax　まず，スカラ論理式が計算され，この式が真のとき，実行文が処理され，その後，再度，スカラ論理式が評価される．この式が真である間，ループ処理が繰り返され，偽になると終了する（ループをぬける）．この式が最初から偽であれば，実行文を一度も処理しない．実行文の中でループの継続条件判定のため，スカラ論理式の値を再設定することが多い．

```
[name:] DO WHILE (scalar-logical-expr)
    block
END DO [name]
```

（互換性：Fortran 90～）

- *name*：do while 構文名（省略可）
- *scalar-logical-expr*：スカラ論理式
- *block*：ブロック（単一あるいは複数の実行文）

次のように DO WHILE 文によって DO 構文と同じような計算ができる.

```
index = istart
[name:] DO WHILE (index .LE. iend)
      block
      index = index + incr
END DO [name]
```

（互換性：Fortran 90〜）

■DO WHILE 文を用いたプログラム例 （f1_do_while.f90）

```
program f1_do_while
  implicit none
  integer i,n ! 整数型
  write(*,'(a)',advance='no') 'Input n. '
  read(*,*) n
  i = 1
  do while(i<=n)
     write(*,'(i4,$)') i
     i = i+1
  enddo
end program f1_do_while
```

8.3.2 do 変数を用いない DO 文

do 変数を用いない DO 文は，不定回数のループとなり，ループの継続・終了の判定処理を行う文を付け加えて使用される.

■Syntax

```
[name:] DO
      block
END DO [name]
```

（互換性：Fortran 90〜）

8.4 EXIT 文

DO ループを途中でぬける場合，EXIT 文を使用する.

8.4.1 単一 DO ループで用いる EXIT 文

繰り返し回数が既定されている DO ループ構文において，条件分岐によって EXIT 文を実行してループをぬけることができる.

■Syntax

```
[name :] DO index = istart, iend [, incr]
     .....
      IF(scalar-logical-expr) EXIT [name]
     .....
END DO [name]
..... ! EXIT [name] の次は，この行が実行される.
```

（互換性：Fortran 90〜）

■EXIT 文を用いたプログラム例１　（f1_exit.f90）

```
program f1_exit
  implicit none
  integer i,n ! 整数型
  n = 9 ! DO ループの繰り返し回数
  write(*,'(100i4)') (i,i=1,n)
  do i=1,n
     if(i>3 .AND. i<7) then
        write(*,'(a4,$)') '' ! 改行抑止
        exit
     endif
     write(*,'(i4,$)') i ! 改行抑止
  enddo
end program f1_exit
```

コンパイル，リンク，実行すると，

8.4.2 DO WHILE 文との比較

■Syntax　　EXIT 文により，不定回数の DO ループをぬけることができる.

```
[name :] DO
     .....
      IF(scalar-logical-expr) EXIT [name]
     .....
END DO [name]
..... ! EXIT [name] の次は，この行が実行される.
```

（互換性：Fortran 90〜）

特に，次のような場合，DO WHILE 文と同じような計算ができる.

```
[name :] DO
      IF(scalar-logical-expr) EXIT [name]
      block
END DO [name]
..... ! EXIT [name] の次は，この行が実行される.
```

（互換性：Fortran 90〜）

したがって，DO WHILE 文の説明で示したように，do 変数による制御を次のようにして

行うことができる.

index = istart
[name :] DO
　　IF(*index .GT. iend*) EXIT *[name]*
　　block
　　index = index + incr
END DO *[name]*
..... ! EXIT *[name]* の次は，この行が実行される.

（互換性：Fortran 90〜）

■EXIT 文を用いたプログラム例 2 （f1_do_while_2.f90）

```
program f1_do_while_2
  implicit none
  integer i,n ! 整数型
  write(*,'(a)',advance='no') 'Input n. '
  read(*,*) n
  i = 1
  do
     if(i>n) exit
     write(*,'(i4,$)') i
     i = i+1
  enddo
```

```
end program f1_do_while_2
```

8.4.3　多重 DO ループで用いる EXIT 文

次のようにぬける DO ループを do 構文名で指定できる.

■EXIT 文で外側ループの do 構文名を指定した場合

```
name-1 : DO index-1 = istart-1, iend-1 [, incr-1]
    [name-2 :] DO index-2 = istart-2, iend-2 [, incr-2]
        .....
        IF(scalar-logical-expr) EXIT name-1
        .....
    END DO [name-2]
END DO name-1
..... ! EXIT name-1 の次は，この行が実行される.
```

（互換性：Fortran 90〜）

■EXIT 文で do 構文名を指定したプログラム例 1　（f1_exit2.f90）

```
program f1_exit2
  implicit none
  integer i,j,n,k ! 整数型
  n = 9 ! DO ループの繰り返し回数
  loop1: do i=1,n
     write(*,*) ' '
```

```
    loop2: do j=1,n
       k = i*j
       if(k>16 .AND. k<32) then
          write(*,'(a4,$)') '' ! 改行抑止
          exit loop1
       endif
       write(*,'(i4,$)') k ! 改行抑止
    enddo loop2
    write(*,'(" ...",$)') ! 改行抑止
  enddo loop1
end program f1_exit2
```

■EXIT 文で内側ループの do 構文名を指定した場合

```
[name-1 :] DO index-1 = istart-1, iend-1 [, incr-1]
     name-2 : DO index-2 = istart-2, iend-2 [, incr-2]
         .....
          IF(scalar-logical-expr) EXIT name-2
         .....
     END DO name-2
    ..... ! EXIT name-2 の次は，この行が実行される.
END DO [name-1]
```

（互換性：Fortran 90〜）

■EXIT 文で do 構文名を指定したプログラム例 2 （f1_exit3.f90）

```
program f1_exit3
  implicit none
  integer i,j,n,k ! 整数型
  n = 9 ! DO ループの繰り返し回数
  loop1: do i=1,n
     write(*,*) ' '
     loop2: do j=1,n
        k = i*j
```

```
        if(k>16 .AND. k<32) then
           write(*,'(a4,$)') '' ! 改行抑止
           exit loop2
        endif
        write(*,'(i4,$)') k ! 改行抑止
     enddo loop2
     write(*,'(" ...",$)') ! 改行抑止
  enddo loop1
end program f1_exit3
```

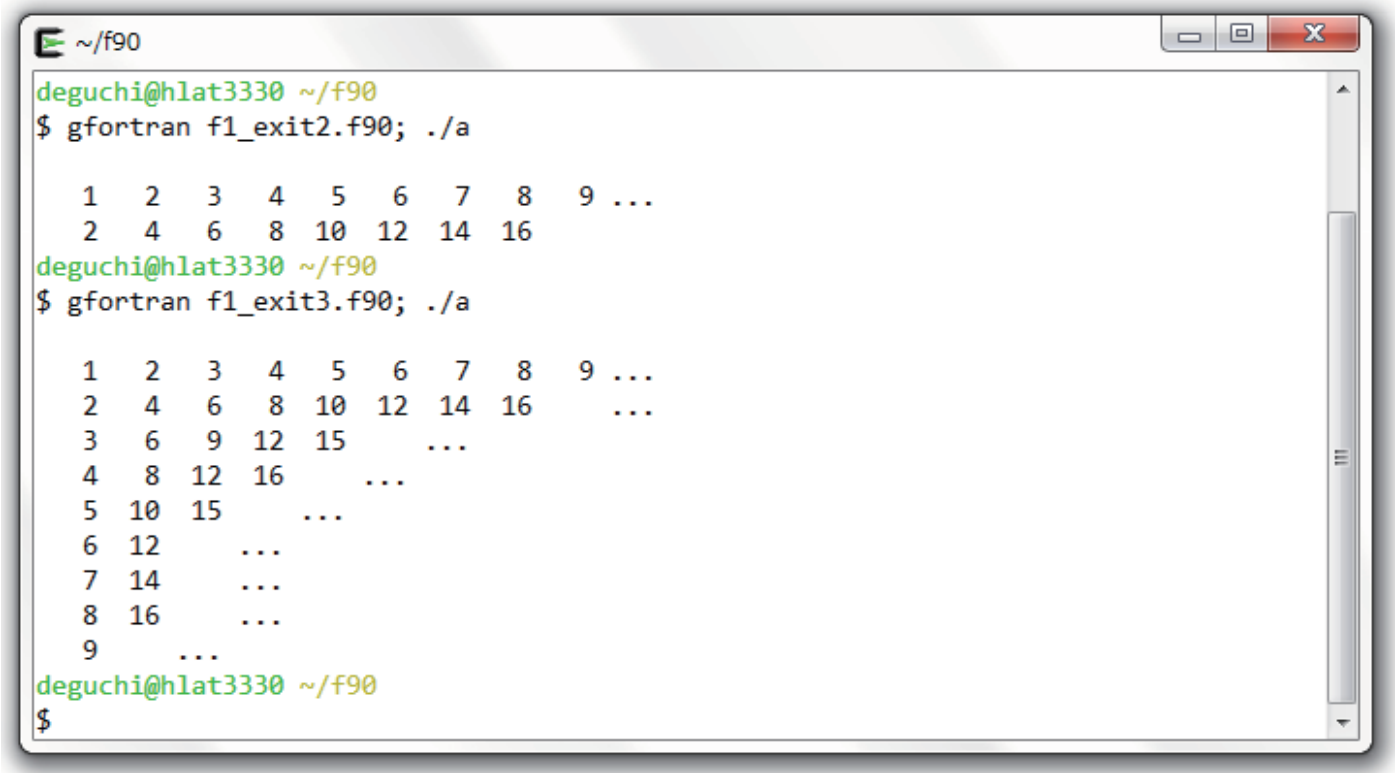
プログラム f1_exit2.f90 では，

```
if(...)  exit loop1
```

として外側のループの do 構文名を指定している．一方，f1_exit3.f90 では，

```
if(...)  exit loop2
```

として内側のループの do 構文名を指定している．両者の実行結果は次のとおり．



8.5 CYCLE 文

DO ループの最初にもどってループを継続する場合，CYCLE 文を使用する.

8.5.1 単一 DO ループで用いる CYCLE 文

繰り返し回数が既定されている DO ループ構文において，条件分岐によって CYCLE 文を実行してループの最初にもどることができる.

■Syntax

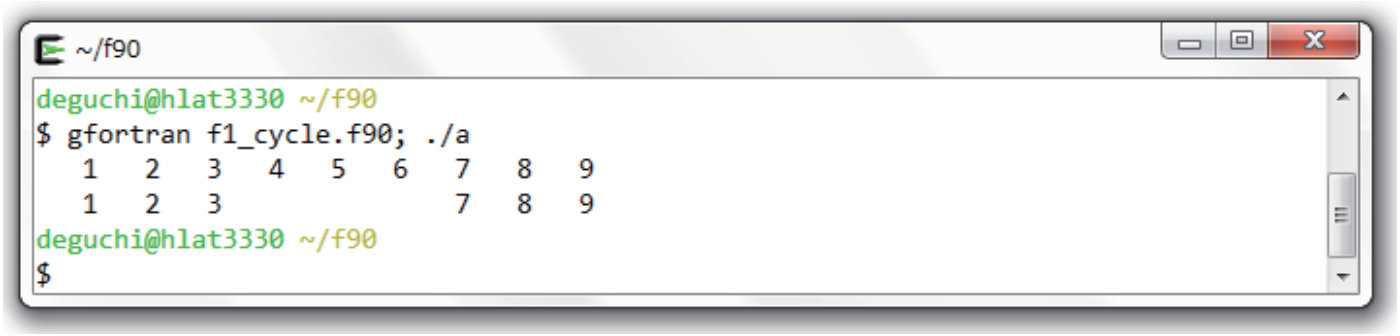
```
[name :] DO index = istart, iend [, incr]
    ..... ! CYCLE [name] の次は，ループの最初（この行）にもどる.
    .....
     IF(scalar-logical-expr) CYCLE [name]
    .....
END DO [name]
```

（互換性：Fortran 95〜）

■CYCLE 文を用いたプログラム例　（f1_cycle.f90）

```
program f1_cycle
  implicit none
  integer i,n ! 整数型
  n = 9 ! DO ループの繰り返し回数
  write(*,'(100i4)') (i,i=1,n)
  do i=1,n
     if(i>3 .AND. i<7) then
        write(*,'(a4,$)') '' ! 改行抑止
        cycle
     endif
     write(*,'(i4,$)') i ! 改行抑止
  enddo
end program f1_cycle
```

コンパイル，リンク，実行すると，



8.5.2　多重 DO ループで用いる CYCLE 文

次のようにぬける DO ループを do 構文名で指定できる.

■CYCLE 文で外側ループの do 構文名を指定した場合

name-1 : DO *index-1 = istart-1, iend-1 [, incr-1]* ! CYCLE *name-1* の次は外側ループの最初にもどる.
　　[name-2 :] DO *index-2 = istart-2, iend-2 [, incr-2]*
　　.....
　　IF(*scalar-logical-expr*) CYCLE *name-1*
　　.....
　　END DO *[name-2]*
END DO *name-1*

（互換性：Fortran 95～）

■CYCLE 文で do 構文名を指定したプログラム例 1　（f1_cycle2.f90）

```
program f1_cycle2
  implicit none
  integer i,j,n,k ! 整数型
  n = 9 ! DOループの繰り返し回数
  loop1: do i=1,n
     write(*,*) ' '
     loop2: do j=1,n
```

```
        k = i*j
        if(k>16 .AND. k<32) then
           write(*,'(a4,$)') '' ! 改行抑止
           cycle loop1
        endif
        write(*,'(i4,$)') k ! 改行抑止
     enddo loop2
     write(*,'(" ...",$)') ! 改行抑止
  enddo loop1
end program f1_cycle2
```

■CYCLE 文で内側ループの do 構文名を指定した場合

[name-1 :] DO *index-1 = istart-1, iend-1 [, incr-1]*
name-2 : DO *index-2 = istart-2, iend-2 [, incr-2]* ! CYCLE *name-2* の次は内側ループの最初にもどる.

```
        .....
        IF(scalar-logical-expr) CYCLE name-2
        .....
     END DO name-2
  END DO [name-1]
```

（互換性：Fortran 95～）

■内側ループの do 構文名を用いたプログラム例 （f1_cycle3.f90）

```
program f1_cycle3
  implicit none
  integer i,j,n,k ! 整数型
  n = 9 ! DO ループの繰り返し回数
  loop1: do i=1,n
     write(*,*) ' '
     loop2: do j=1,n
        k = i*j
        if(k>16 .AND. k<32) then
           write(*,'(a4,$)') '' ! 改行抑止
           cycle loop2
        endif
        write(*,'(i4,$)') k ! 改行抑止
     enddo loop2
     write(*,'(" ...",$)') ! 改行抑止
  enddo loop1
end program f1_cycle3
```

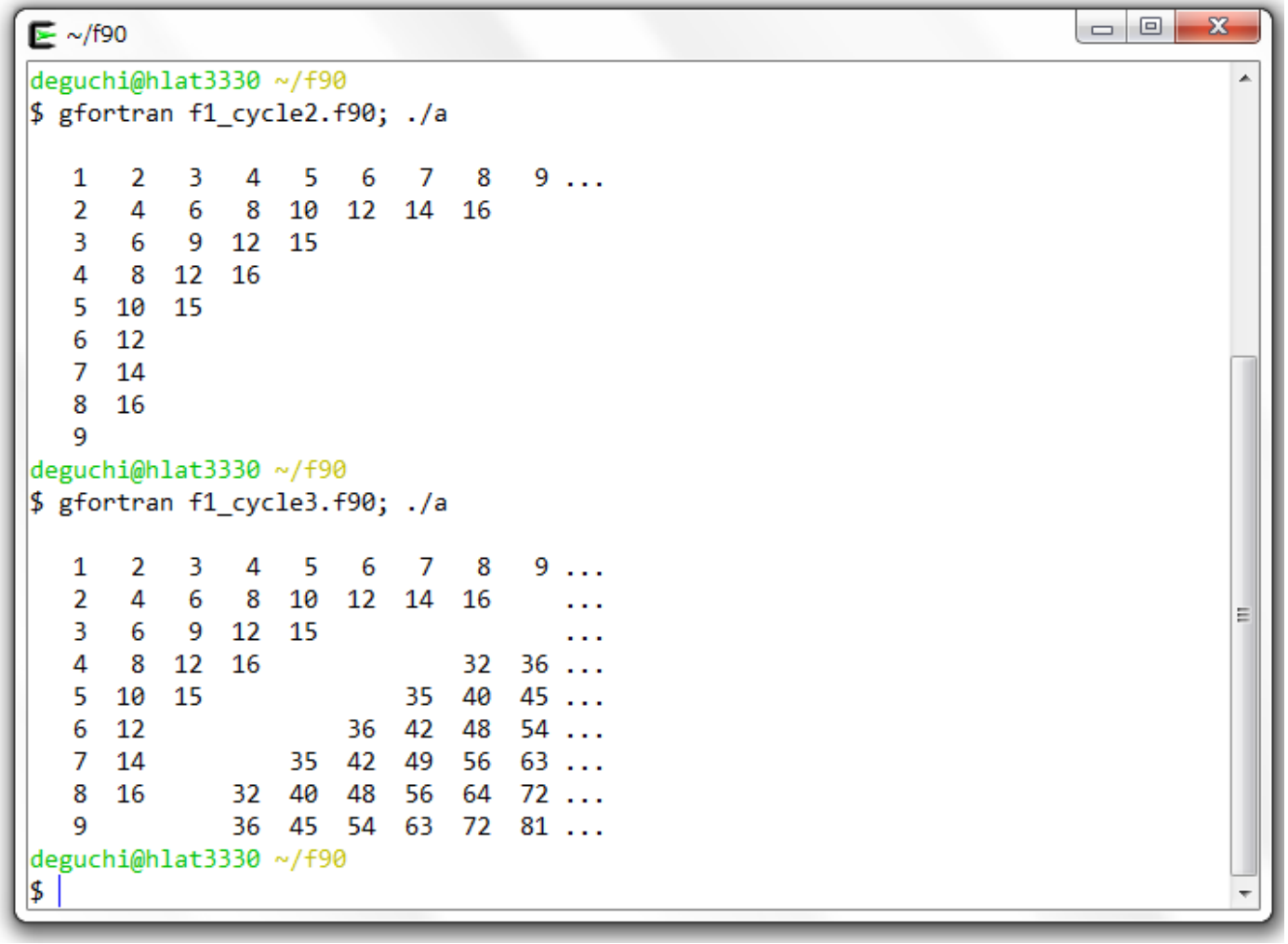
プログラム f1_cycle2.f90 では，

```
if(...)  cycle loop1
```

として外側のループの do 構文名を指定している．一方，f1_cycle3.f90 では，

```
if(...)  cycle loop2
```

として内側のループの do 構文名を指定している．両者の実行結果は次のとおり．



8.5.3 CYCLE 文と EXIT 文

■do 変数を用いない例

[name :] DO ! CYCLE *[name]* の次は，ループの最初（この行）にもどる.

```
     .....
      IF (scalar-logical-expr) CYCLE [name]
     .....
      IF (scalar-logical-expr) EXIT [name]
END DO [name]
..... ! EXIT [name] の次は，この行が実行される.
```

（互換性：Fortran 95〜）

■do 変数を用いる例

[name :] DO *index = istart, iend [, incr]* ! CYCLE *[name]* の次は，ループの最初（この行）にもどる.

```
     .....
      IF(scalar-logical-expr) CYCLE [name]
     .....
      IF (scalar-logical-expr) EXIT [name]
END DO [name]
..... ! EXIT [name] の次は，この行が実行される.
```

（互換性：Fortran 95〜）

9　配列

配列に関しては，Fortran 90/95 では大幅に見直しが行われ，FORTRAN77 に比べて機能が増え強化されている.

9.1　配列の宣言

9.1.1　DIMENSION 文

■Syntax　型宣言とは別に，配列を宣言する場合，次のようになる（非実行文ゆえ，最初の実行文よりも前に書く）.

```
TYPE array, .....
DIMENSION array( array-spec ), .....
```

（互換性：FORTRAN 77～）

- *TYPE*：配列の型.
- *array*：配列名.
- *array-spec*：配列の形状（array specification）.

配列名に対する型宣言文がない場合，その配列の型は暗黙の型宣言に従う．あるいは，

```
TYPE  array ( array-spec ) ,  .....
```

（互換性：FORTRAN 77〜）

1 次元および 2 次元配列では（3〜7 次元配列も同様），

```
DIMENSION  array-1 ( [lower-1:] upper-1 ) ,  array-2 ( [lower-1:] upper-1 , [lower-2:] upper-2 )
```

（互換性：FORTRAN 77〜）

- *array-i*：配列名．
- *lower-j*：第 *j* 次元の配列添字の下限（lower dimension bound）．整数値．
- *upper-j*：第 *j* 次元の配列添字の上限（upper dimension bound）．整数値．

下限 *lower-j* を省略すると *lower-j*=1 となり，ほとんどこのようにして配列が宣言される．

```
DIMENSION  array-1 ( upper-1 ) ,  array-2 ( upper-1, upper-2 ) ,  . . . . .
DIMENSION  array-7 ( upper-1, upper-2, upper-3, upper-4, upper-5, upper-6, upper-7 )
```

（互換性：FORTRAN 77〜）

例えば，1 次元配列の場合，配列要素（array element）は，*array-1*(1)，*array-1*(2)，... ，*array-1* (*upper-1*) ．

■1 次元配列および 2 次元配列の例

```
dimension a(10),b(10)
dimension c(10,10),d(10,10)
```

■配列要素への代入，初期化　配列要素に一つずつ値を設定する場合，

array（ *subscript* ） ＝ *scalar-expr*

（互換性：FORTRAN 77〜）

- *array*：配列名.
- *subscript*：配列の添字.
- *array*（ *subscript* ）：配列の 1 要素.
- *scalar-expr*：スカラ値あるいはスカラ式.

9.1.2 PARAMETER 文

■Syntax　定数に英字名を付けるために用いる.

PARAMETER （ *constant-name* = *constant-expr* ）

（互換性：FORTRAN 77〜）

- *constant-name*：定数名
- *constant-expr*：定数式

配列の宣言に用いる場合，例えば,

```
PARAMETER (M=10,N=10)
DIMENSION A(N), B(N)
DIMENSION C(M,N), D(M,N)
```

9.1.3 DIMENSION 属性指定子

■Syntax　変数の属性指定（:: が必要）によって配列を宣言する場合，次のようになる．

```
TYPE, DIMENSION( array-spec) ::  array-i, ....
```

（互換性：Fortran 90〜）

1 次元および 2 次元配列では（3〜7 次元配列も同様），

```
TYPE, DIMENSION( [lower-1:] upper-1) ::  array-1, array-1', .....
TYPE, DIMENSION( [lower-1:] upper-1 , [lower-2:] upper-2) ::  array-2, array-2', .....
```

（互換性：Fortran 90〜）

- *TYPE*：配列の型．
- *array-spec*：配列の形状（array shape specification）．
- *array-i*：配列名．
- *lower-j*：第 *j* 次元の配列添字の下限（整数値）．下限を省略すると *lower-j*=1．
- *upper-j*：第 *j* 次元の配列添字の上限（整数値）

このように形状を数値によって指定した配列を，形状明示配列（explicit-shape array）という．あるいは，次のように記述することもできる（この場合，上位互換で :: は省略可）．

TYPE :: *array-1*（*upper-1*），*vector*（*upper-1'*）
TYPE :: *array-2*（*upper-1*，*upper-2*），*matrix*（*upper-1'*，*upper-2'*）

（互換性：Fortran 90〜）

9.1.4 PARAMETER 属性指定子

■Syntax　定数に英字名を付けるために用いる.

TYPE, PARAMETER :: *constant-name* = *constant-expr*

（互換性：Fortran 90〜）

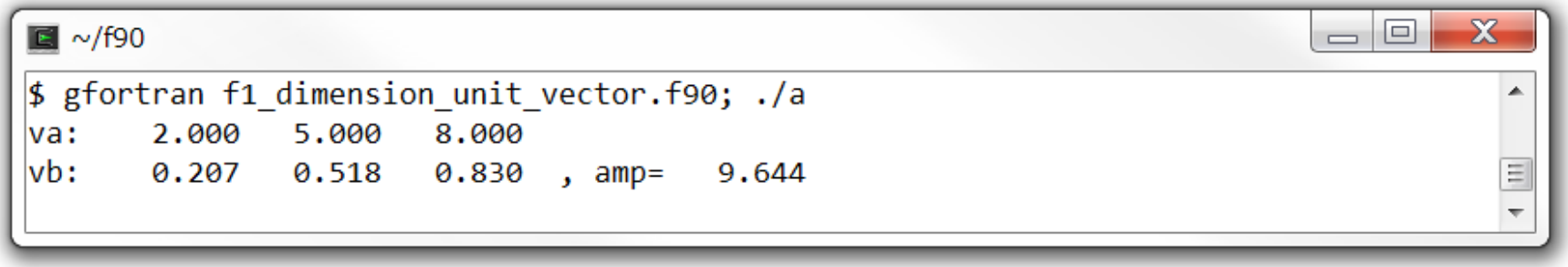
配列の宣言に用いる場合，例えば,

```
integer, parameter :: m=10, n=10
real, dimension :: a(m), b(n)
real, dimension(m,n) :: c, d
real, dimension :: e(m,m), f(n,n,n), g(m,m,n,n,n)
```

■1 次元配列に関するプログラム例　要素数 3 の 1 次元配列を用い，3 次元空間ベクトルに対する単位ベクトルおよび振幅を計算する（f1_dimension_unit_vector.f90）.

```
program f1_dimenion_unit_vector
  implicit none
  integer, parameter :: nn=3
  integer i
  real(8) va(nn),vb(nn) ! 1 次元配列の宣言（倍精度実数型）
  real(8) amp
! ベクトルの振幅および単位ベクトルの計算
  va(1) = 2.0D0; va(2) = 5.0D0; va(3) = 8.0D0
  write(*,'(a4,3f8.3)') 'va: ',(va(i),i=1,3)
  amp = va(1)**2+va(2)**2+va(3)**2
  amp = sqrt(amp)
  do i=1,3
     vb(i) = va(i)/amp
  enddo
  write(*,'(a4,3f8.3,a8,f8.3)')&
       'vb: ',(vb(i),i=1,3),', amp=',amp
end program f1_dimenion_unit_vector
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_dimension_unit_vector.f90; ./a
va: 2.000   5.000   8.000
vb: 0.207   0.518   0.830 , amp=   9.644

9.2　配列の動的割り付け

割り付け配列（allocatable array）は，配列の大きさを適宜変更できる.

9.2.1　ALLOCATABLE 属性

■Syntax　配列範囲を明示的に指定せず，次元数のみを指定した配列を形状無指定配列（deferred-shape array）とよび，ALLOCATABLE 属性（attribute）を用いて型宣言できる.

```
TYPE, ALLOCATABLE, DIMENSION( :,:, ... ) :: array, .....
```

（互換性：Fortran 90〜）

あるいは,

```
TYPE, ALLOCATABLE :: array( :,:, ... ), .....
```

（互換性：Fortran 90〜）

- *TYPE*：配列の型.
- *array*：配列名.

9.2.2　割り付け配列

■ALLOCATE 文　　ALLOCATABLE 属性を与えた配列に対して配列寸法を指定し，配列の記憶場所を割付ける.

```
ALLOCATE( array( array-spec ), .....   [,stat])
```

（互換性：Fortran 90〜）

- *array*：配列名.
- *array-spec*：配列の形状（array shape specification）.
- stat：オプション（省略可）．スカラ整数変数．すべての配列に対して割付けが成功した場合，ゼロに設定され，一つでも失敗した場合にはエラーコードが設定される.

■DEALLOCATE 文　　割付けの解除は次のようにして行う.

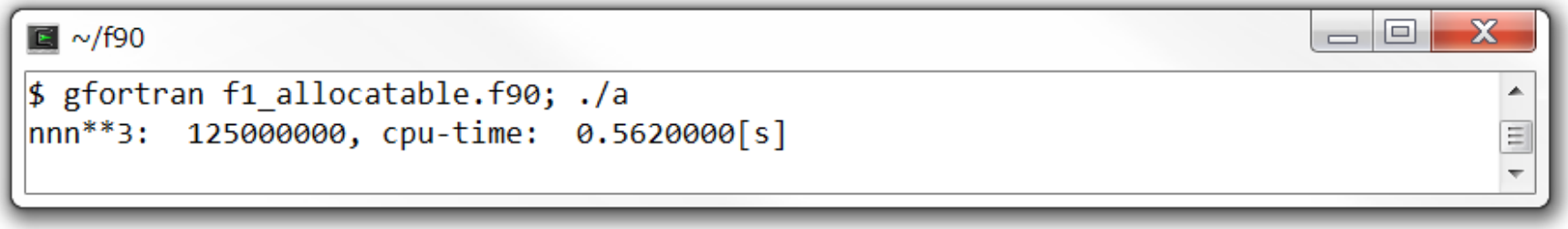
```
DEALLOCATE( array )
```

（互換性：Fortran 90〜）

■動的割り付けに関するプログラム例　　3 次元配列の場合，例えば次のようなる (f1_allocatable.f90),

```
program f1_allocatable
  implicit none
  integer nnn,is
  real(8), allocatable :: aaa(:,:,:)
  real(8) t1,t2

  nnn = 500
  call cpu_time(t1)
  allocate(aaa(nnn,nnn,nnn),stat=is)
  if(is.ne.0) stop 'connot allocate'
  aaa(:,:,:) = 1.0D0
  deallocate(aaa)
  call cpu_time(t2)
  write(*,'("nnn**3: ",i10,", cpu-time: ",f10.7,"[s]")') nnn**3,t2-t1

end program f1_allocatable
```

コンパイル，リンク，実行すると,


~/f90
$ gfortran f1_allocatable.f90; ./a
nnn**3: 125000000, cpu-time: 0.5620000[s]

9.3　部分配列と全体配列

■Syntax　配列のサブセット（array subsets）のことを部分配列（array section）という.

array(*sect-subscript-list*)

（互換性：Fortran 90〜）

例えば，1 次元および 2 次元配列では（3〜7 次元配列も同様），

array-1(*first-bound-1*:*last-bound-1* *[:stride-1]*)
array-2(*first-bound-1*:*last-bound-1* *[:stride-1]*, *first-bound-2*:*last-bound-2* *[:stride-2]*)

（互換性：Fortran 90〜）

- *array*：配列名（1〜7 次元配列）.
- *array-i*：*i* 次元配列名.
- *sect-subscript-list*：部分配列の添字範囲.
- *first-bound-j*, *last-bound-j*：第 *j* 次元の配列添字の開始値，終値（整数値）.
- *stride-j*：第 *j* 次元の配列添字の増分（整数値）．省略すると *stride-j*=1.

部分配列の添字範囲を指定する添字 3 つ組（subscript triplets）は，上で示したものも含

め，全てあげると次のようになる（2〜7 次元配列も同様）.

```
array( first : last : stride )
array( first : last ) !  stride=1
array( first :) !  last=upper, stride=1
array( first :: stride ) !  last=upper
array(: last ) !  first=lower, stride=1
array(: last : stride ) !  first=lower
array(:: stride ) !  first=lower, last=upper
```

（互換性：Fortran 90〜）

さらに，配列要素を全て指定すると，

```
array( lower : upper ) !  stride=1
array(:) !  first=lower, last=upper, stride=1
```

（互換性：Fortran 90〜）

このように配列全体（whole array）の場合には，次のように配列名だけでもよい.

```
array !  first=lower, last=upper, stride=1
```

（互換性：Fortran 90〜）

■（部分）配列を用いた式　右辺と左辺の配列の形状が一致している場合，次のような部分配列や配列全体による代入が行える．

```
array = array-a
array = array-b( first : last : stride )
array( first : last : stride ) = array-c
array( first : last : stride ) = array-d( first : last : stride )
```

（互換性：Fortran 90〜）

配列全体を指定する場合，次のように明示的に配列であることを示すのが望ましい．

```
array(:)   = array-a(:)
array(:)   = array-b( first : last : stride )
array( first : last : stride ) = array-c(:)
```

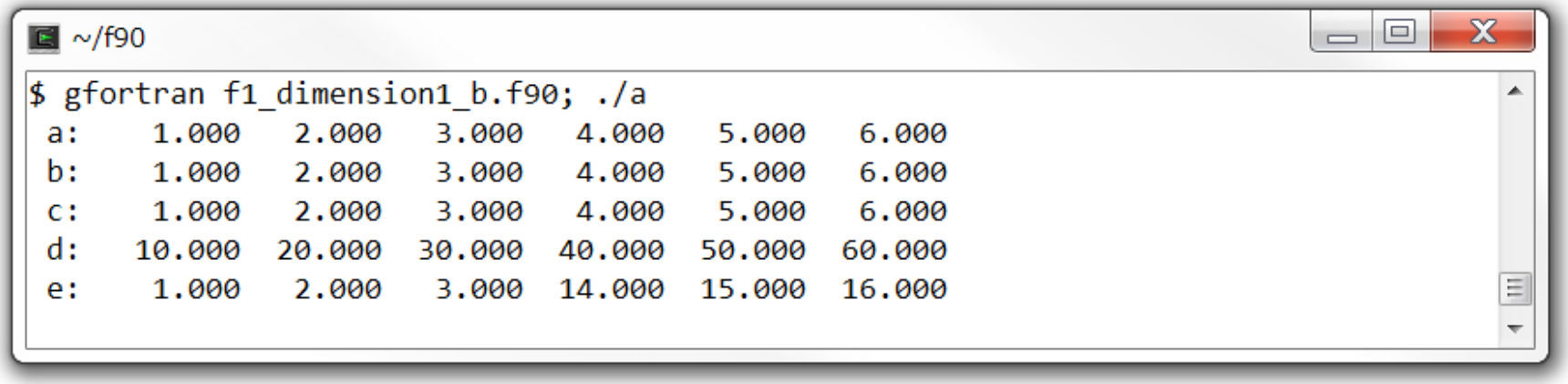
（互換性：Fortran 90〜）

■部分配列に関するプログラム例　（ f1_dimension1_b.f90）．

```
program f1_dimenion1_b
  implicit none
  real, dimension(6) :: a,b,c,d,e,f
  integer i
```

```
  a(:) = (/ (1.0*i, i=1,6) /)
  b(:) = a(1:6)
  c(:) = a(:)
  d(:) = a(:)*10.0
  e(1:3) = a(1:3)
  e(4:6) = a(4:6)+10.0

  write(*,'(a4,6f8.3)')'a: ',(a(i),i=1,6)
  write(*,'(a4,6f8.3)')'b: ',b(1:6)
  write(*,'(a4,6f8.3)')'c: ',c(:)
  write(*,'(a4,6f8.3)')'d: ',d(:)
  write(*,'(a4,6f8.3)')'e: ',e(:)
end program f1_dimenion1_b
```

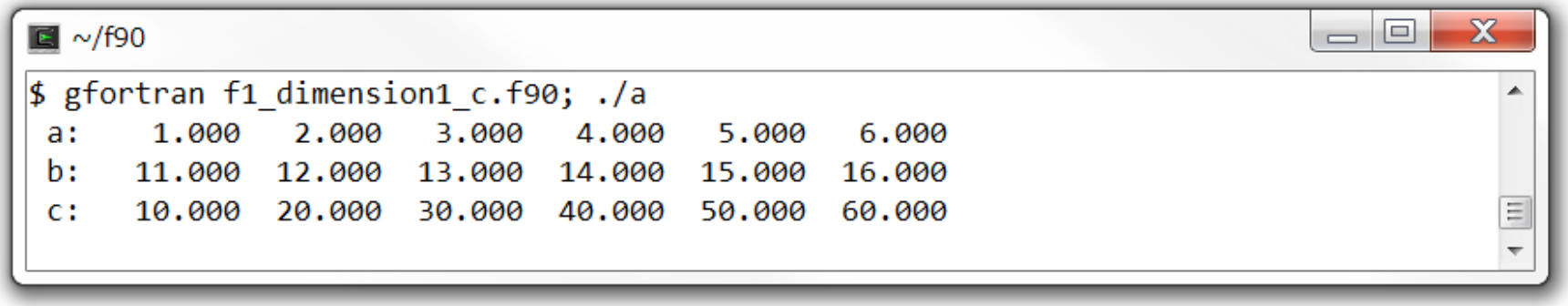
コンパイル，リンク，実行すると，



次のように（:）は省略できる.

```
program f1_dimenion1_c
  implicit none
  real, dimension(6) :: a,b,c
  integer i

  a = (/ (1.0*i, i=1,6) /)
  b = a+10.0
  c = a*10.0

  write(*,'(a4,6f8.3)')'a: ',(a(i),i=1,6)
  write(*,'(a4,6f8.3)')'b: ',b
  write(*,'(a4,6f8.3)')'c: ',c
end program f1_dimenion1_c
```

コンパイル，リンク，実行すると，



9.4　定数の代入

要素すべてを一つの値に設定することもできる.

array = *scalar-expr*

（互換性：Fortran 90〜）

あるいは，1 次元配列の場合,

array(:) = *scalar-expr*

（互換性：Fortran 90〜）

9.5　配列構成子

■配列構成子による 1 次元配列の代入　宣言された 1 次元配列に対して，配列構成子（array constructors）（/ および /）を用いると配列要素に値を代入できる.

```
array = (/ spec /)
```

（互換性：Fortran 90〜）

配列構成要素並び *spec* は，例えば,

```
array = (/ scalar-1, scalar-2, ..... /)
array = (/ (expr, index = istart, iend [, incr]) /)
array = (/ array-1( sect-subscript-list ) /)
```

（互換性：Fortran 90〜）

- *array*：配列名.
- *spec*：配列構成要素並び.
- *scalar-i (i=1,2, ...)*：スカラ値.
- *(expr, index = istart, iend [, incr])*：DO 型並び.
- *array-1(sect-subscript-list)*：部分配列.

また，次のように並べた指定でもよい．

```
array = (/ scalar-1, scalar-2, (expr, index = istart, iend [, incr]), array-1( sect-subscript-list ) .....
/)
```

（互換性：Fortran 90〜）

次のように明示的に 1 次元配列であることを示すこともできる．

```
array(:)  = (/ scalar-1, scalar-2, .....  /)
array(:)  = (/ (expr, index = istart, iend [, incr]) /)
array(:)  = (/ array-1( sect-subscript-list ) /)
```

（互換性：Fortran 90〜）

■配列構成子による配列の初期化　宣言時に配列を初期化する場合，

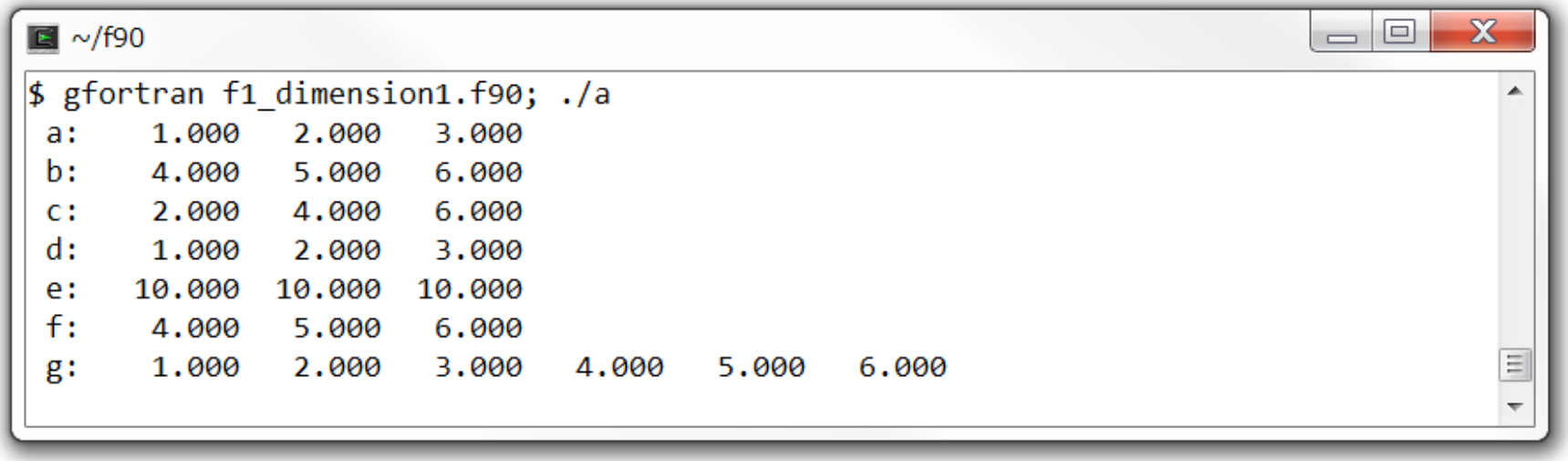
```
TYPE, DIMENSION( array-spec) ::  array = (/ spec /)
TYPE ::  array( array-spec) = (/ spec /)
```

（互換性：Fortran 90〜）

■1 次元配列によるプログラム例　代入，初期化を行う（ f1_dimension1.f90）.

```
program f1_dimenion1
  implicit none
  real(8), dimension(3) :: a = (/ 1.0, 2.0, 3.0 /), b,c,d,e
  real(8) f(6)
  integer i

  b(:) = (/ 4.0, 5.0, 6.0 /)
  c(:) = (/ (2.0*i, i=1,3) /)
  d(:) = (/ a(1),a(2),a(3) /)
  e(:) = (/ b(1:3) /)
  f(:) = (/ a(1:3), b(1:3) /)

  write(*,'(a4,3f8.3)')'a: ',(a(i),i=1,3)
  write(*,'(a4,3f8.3)')'b: ',(b(i),i=1,3)
  write(*,'(a4,3f8.3)')'c: ',(c(i),i=1,3)
  write(*,'(a4,3f8.3)')'d: ',(d(i),i=1,3)
  write(*,'(a4,3f8.3)')'e: ',(e(i),i=1,3)
  write(*,'(a4,6f8.3)')'e: ',(f(i),i=1,6)
end program f1_dimenion1
```

コンパイル，リンク，実行すると，

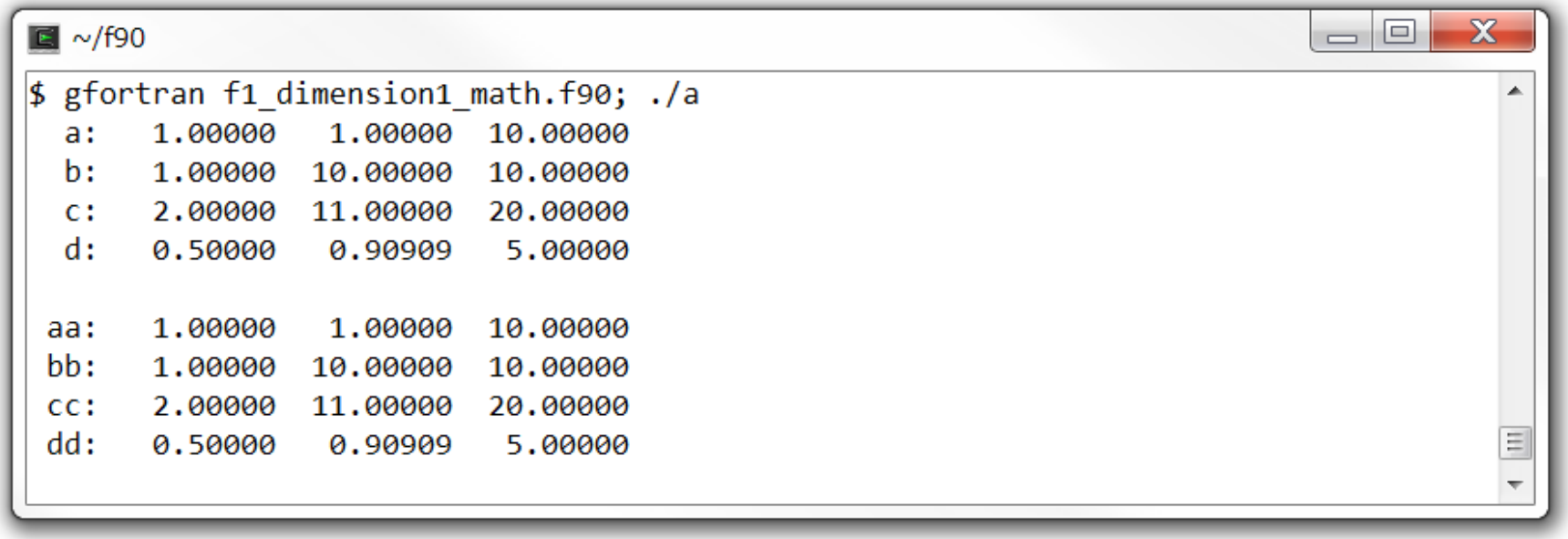

~/f90
$ gfortran f1_dimension1.f90; ./a
a: 1.000 2.000 3.000
b: 4.000 5.000 6.000
c: 2.000 4.000 6.000
d: 1.000 2.000 3.000
e: 10.000 10.000 10.000
f: 4.000 5.000 6.000
g: 1.000 2.000 3.000 4.000 5.000 6.000

■配列に対する四則演算に関するプログラム例　配列あるいは部分配列を用いて配列要素に対して四則演算が行える（ f1_dimension1_math.f90）.

```
program f1_dimenion1_math
  implicit none
  integer i
  real(8) a(3),b(3),c(3),d(3), aa(3),bb(3),cc(3),dd(3)
! FORTRAN77
  a(1) = 1.0D0
  a(2) = 1.0D0
  a(3) = 1.0D1
  b(1) = 1.0D0
  b(2) = 1.0D1
  b(3) = 1.0D1
  do i=1,3
     c(i) = a(i)+b(i)
     d(i) = 1.0D0/(1.0D0/a(i)+1.0D0/b(i))
  enddo
  write(6,600) 'a:',(a(i),i=1,3)
  write(6,600) 'b:',(b(i),i=1,3)
  write(6,600) 'c:',(c(i),i=1,3)
  write(6,600) 'd:',(d(i),i=1,3)
```

```
  write(*,*)
600 format(a4,3f10.5)
! Fortran90/95
  aa(:) = a(:)
  bb(:) = (/ 1.0D0, 1.0D01, 1.0D1 /)
  cc(:) = aa(:)+bb(:)
  dd(:) = 1.0D0/(1.0D0/aa(:)+1.0D0/bb(:))
  write(*,'(a4,3f10.5)')'aa:',aa(:)
  write(*,'(a4,3f10.5)')'bb:',bb(:)
  write(*,'(a4,3f10.5)')'cc:',cc(:)
  write(*,'(a4,3f10.5)')'dd:',dd(:)
end program f1_dimenion1_math
```

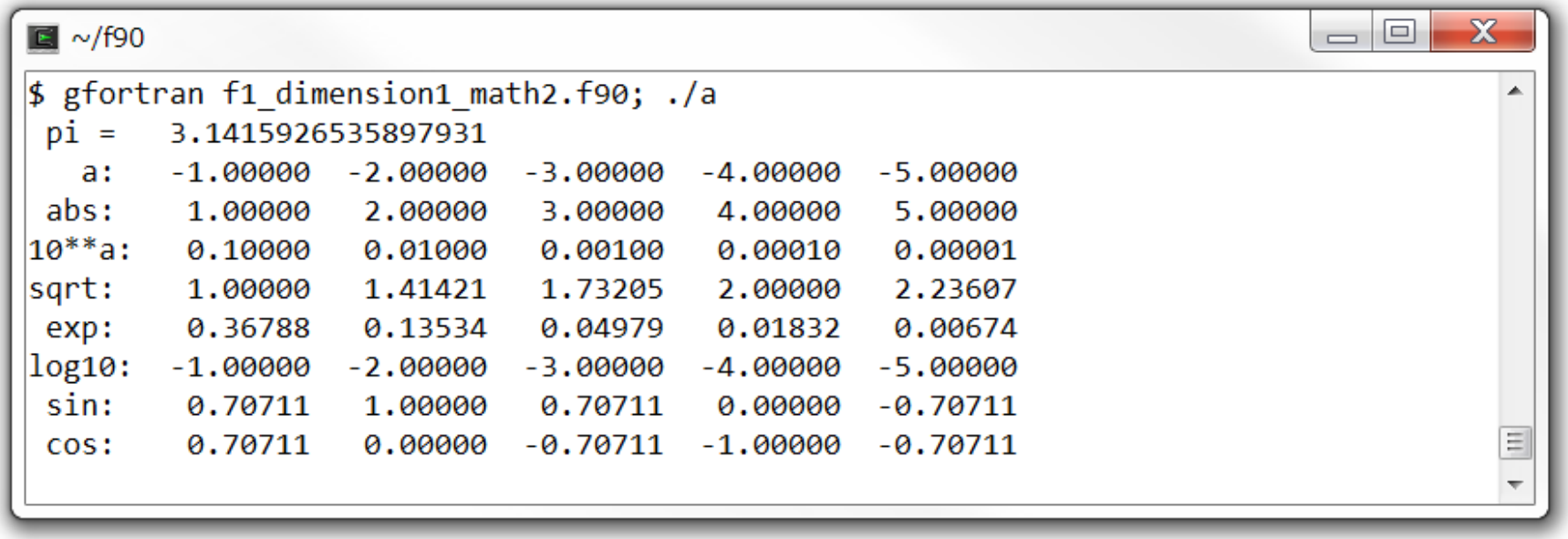
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_dimension1_math.f90; ./a
a: 1.00000 1.00000 10.00000
b: 1.00000 10.00000 10.00000
c: 2.00000 11.00000 20.00000
d: 0.50000 0.90909 5.00000
aa: 1.00000 1.00000 10.00000
bb: 1.00000 10.00000 10.00000
cc: 2.00000 11.00000 20.00000
dd: 0.50000 0.90909 5.00000

■配列に対する組込み演算に関するプログラム例　絶対値，平方根，べき乗などについても，配列要素に対して演算が行える（ f1_dimension1_math2.f90）.

```
program f1_dimenion1_math2
  implicit none
  integer i
  real(8) a(5),b(5),c(5)
  real(8) pi
  pi = acos(0.0D0)*2.0D0
  write(*,*) 'pi =',pi
  a(:) = (/ (-1.0*i, i=1,5) /)
  b(:) = abs(a)
  c(:) = 10.0D0**a(:)
  write(*,'(a6,5f10.5)')'a: ',a(:)
  write(*,'(a6,5f10.5)')'abs: ',b(:)
  write(*,'(a6,5f10.5)')'10**a: ',c(:)
  write(*,'(a6,5f10.5)')'sqrt: ',sqrt(b(:))
  write(*,'(a6,5f10.5)')'exp: ',exp(-b(:))
  write(*,'(a6,5f10.5)')'log10: ',log10(c(:))
  write(*,'(a6,5f10.5)')'sin: ',sin(b(:)*pi*0.25D0)
  write(*,'(a6,5f10.5)')'cos: ',cos(b(:)*pi*0.25D0)
end program f1_dimenion1_math2
```

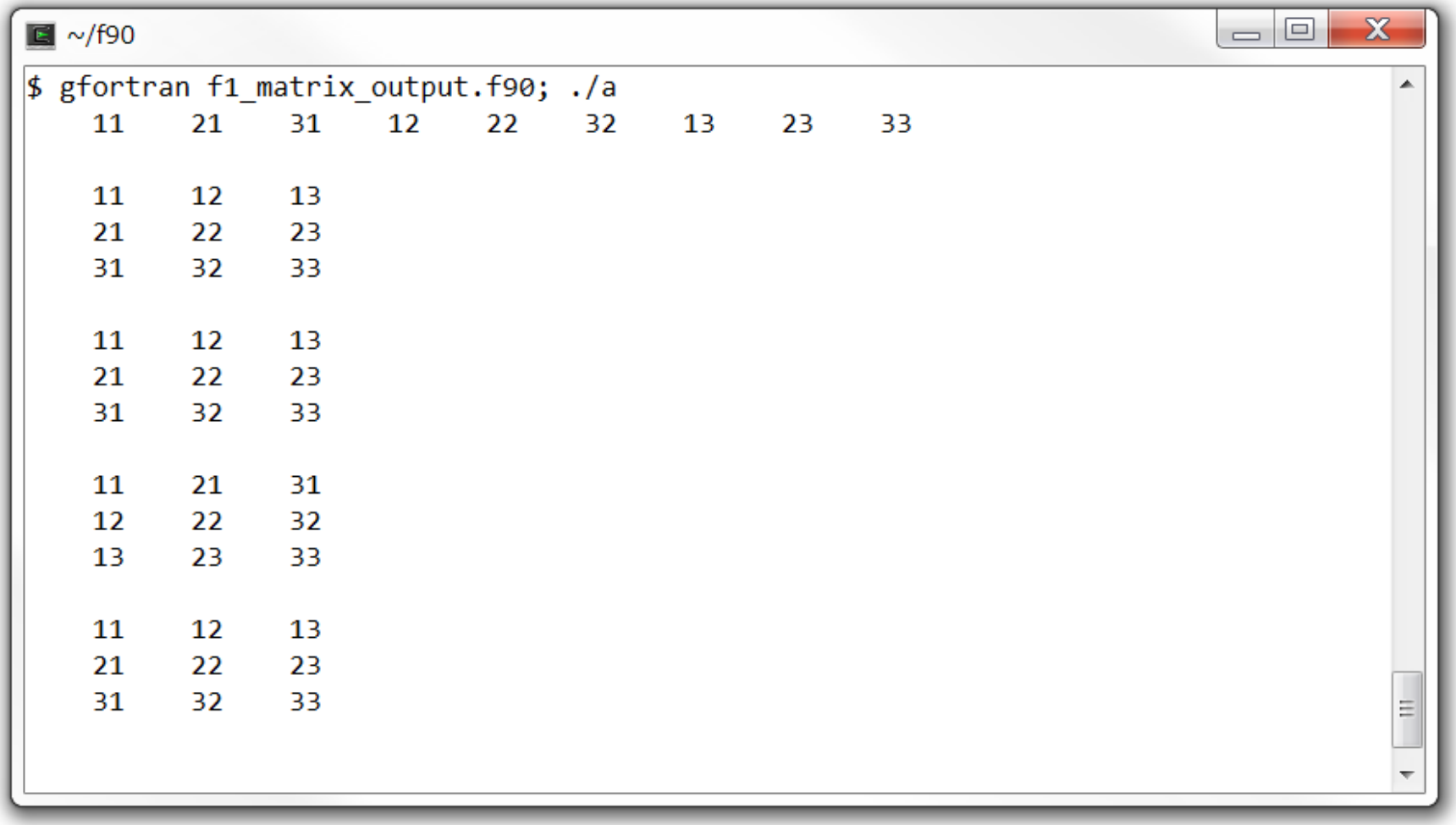
コンパイル，リンク，実行すると，



■多次元配列に関するプログラム例　2 次元配列を行列として用いる場合，1 次元のデータ並びの順序は，列順となっている．次のプログラムでは 2 次元配列を出力してその並びを確認し，行列に適した出力例を示す（ f1_matrix_output.f90）．

```
program f1_matrix_output
  implicit none
  integer, parameter :: nn = 3
  integer k(nn,nn)
  integer i,j
  k(1,:) = (/ 11, 12, 13 /)
  k(2,:) = (/ 21, 22, 23 /)
  k(3,:) = (/ 31, 32, 33 /)
  write(*,'(100i6)') k(:,:)
  write(*,'(/3(3i6/))') ((k(i,j),j=1,3),i=1,3)
  write(*,'(3(3i6/))') (k(i,:),i=1,3)
  write(*,'(3(3i6/))') k(:,:)
  write(*,'(3(3i6/))') transpose(k)
end program f1_matrix_output
```

コンパイル，リンク，実行すると，

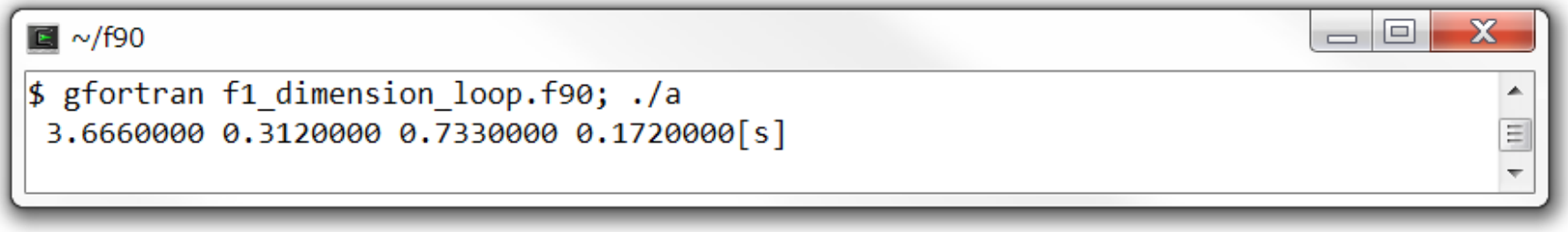

~/f90
$ gfortran f1_matrix_output.f90; ./a
11 21 31 12 22 32 13 23 33
11 12 13
21 22 23
31 32 33
11 12 13
21 22 23
31 32 33
11 21 31
12 22 32
13 23 33
11 12 13
21 22 23
31 32 33

■多次元配列要素のアクセスに関するプログラム例　多次元配列を用いて計算を行う場合，配列要素へのアクセスの順序によって計算時間が異なる場合がある．これもデータが列順に並んでいることによるものである（f1_dimension_loop.f90）．

```
program f1_dimension_loop
  implicit none
  integer, parameter :: nn=700
  integer :: kk(nn,nn,nn)
  integer i,j,k,n
  real(8) t1,t2,t3,t4,t5

  n = 500
  call cpu_time(t1)
  do i=1,n ! slow
     do j=1,n
        do k=1,n
           kk(i,j,k) = 1
        enddo
     enddo
  enddo
  call cpu_time(t2)
  do k=1,n
```

```
      do j=1,n
         do i=1,n
            kk(i,j,k) = 1
         enddo
      enddo
   enddo
   call cpu_time(t3)
   kk(:,:,:) = 1 ! 配列代入文
   call cpu_time(t4)
   kk(1:n,1:n,1:n) = 1 ! 配列代入文（部分配列）
   call cpu_time(t5)
   write(*,'(4f10.7,"[s]")') t2-t1, t3-t2, t4-t3, t5-t4
end program f1_dimension_loop
```

コンパイル，リンク，実行すると，



10　配列選別代入（WHERE，FORALL）

条件式を満たした配列要素に対して代入文を実行する.

10.1　WHERE

10.1.1　（単純）WHERE 文（WHERE statement）

論理配列式が真の場合に，配列代入文を実行する.

■Syntax

```
WHERE (mask-expr)  assign-stmt
```

（互換性：Fortran 90～）

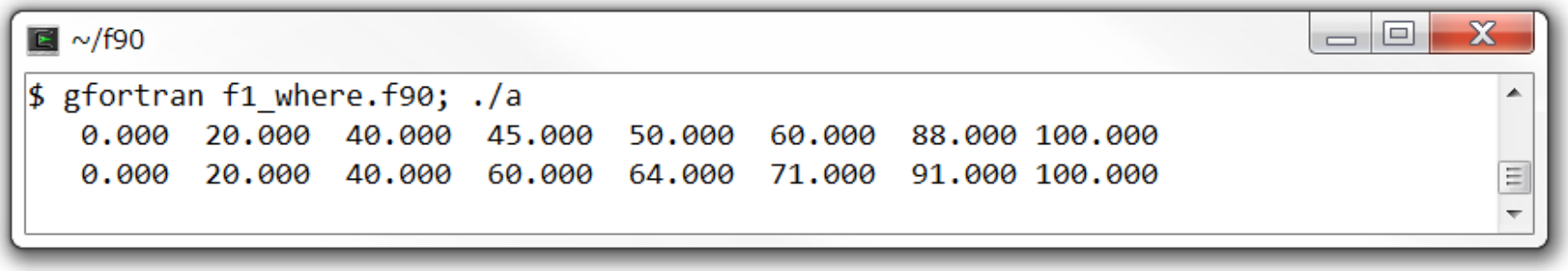
- *mask-expr*：配列選別式（配列要素に関係演算子を適用した論理式を要素とする論理配列式）．マスク式ともいう.
- *assign-stmt*：配列代入文

論理配列式と配列代入文の配列名は異なってもよいが，同じ配列形状でなければならない.

■WHERE 文に関するプログラム例　（f1_where.f90）

```
program f1_where
  implicit none
  real :: x(8),a=45.0,b
  x(:) = (/ 0.0, 20.0, 40.0, 45.0, 50.0, 60.0, 88.0, 100.0 /)
  write(*,'(8f8.3)') x(:)
  b = (100.0-60.0)/(100.0-a)
  where(x>=a) x = nint(b*(x-a)+60.0)
  write(*,'(8f8.3)') x(:)
end program f1_where
```

コンパイル，リンク，実行すると，



10.1.2　WHERE 構文（WHERE construct）

WHERE 構文は論理配列式の制御で，対応する配列選別式の成分が真の場合のみ代入される．

■Syntax

```
[name:] WHERE (mask-expr-1)
      assign-block-1
[ELSEWHERE (mask-expr-2) [[name]]
      assign-block-2]
[.....
     .....      ]
[ELSEWHERE [[name]]
      assign-block-N]
END WHERE [name]
```

（互換性：Fortran 90～）

- *mask-expr-i (i=1,2, ... ,N)*：配列選別式（マスク式）．
- *assign-block-i (i=1,2, ..., N)*：単一あるいは複数の配列代入文．

ただし，構文名は省略できる．

■WHERE 構文に関するプログラム例 （f1_where_2.f90）

```
program f1_where_2
  implicit none
  real :: x(8),a=45.0,b,c
  x(:) = (/ 0.0, 20.0, 40.0, 45.0, 50.0, 60.0, 88.0, 100.0 /)
  write(*,'(8f8.3)') x(:)
  b = (100.0-60.0)/(100.0-a)
  c = 60.0/a
  where(x>=a)
     x = nint(b*(x-a)+60.0)
  elsewhere
     x = nint(c*(x-a)+60.0)
  end where
  write(*,'(8f8.3)') x(:)
end program f1_where_2
```

10.2 FORALL

forall は，where を含む，一般化したものである．

10.2.1 （単純）FORALL 文（FORALL statement）

配列要素の範囲を指定できる．

■Syntax　1 次元配列に対しては，

FORALL (*index = lower : upper [: stride] [, mask-expr]*) *assign-stmt*

（互換性：Fortran 95〜）

- *index*：指標変数．
- *lower, upper, stride*：指標変数の添字 3 つ組．開始値，終値，刻み値（整数）．刻み値を省略すると 1 となる．
- *mask-expr*：配列選別式（マスク式）．省略可．
- *assign-stmt*：配列代入文．

■1 次元配列を用いた FORALL 文に関するプログラム例 （f1_forall.f90）

```
program f1_forall
  implicit none
  integer :: i
  real :: a(6) = 0.0
  a(2:5) = (/ -3.0, -1.0, 1.0, 3.0 /)
  write(*,'(6f8.3)') a(:)
  forall(i=2:5,a(i)<0.0) a(i) = abs(a(i))
  write(*,'(6f8.3)') a(:)
  forall(i=2:6) a(i) = a(i-1) ! 同時的代入
  write(*,'(6f8.3)') a(:)
end program f1_forall
```

■2 次元配列を用いた FORALL 文に関するプログラム例 （f1_forall_2.f90）

```
program f1_forall_2
  implicit none
  real :: a(3,3) = 0.0
  integer i,j
  forall(i=1:2,j=2:3) a(i,j) = i*j ! 論理配列式省略の例
  write(*,'(3(3f8.3/))') transpose(a)
```

```
end program f1_forall_2
```

10.2.2 FORALL 構文（FORALL construct）

複数の配列代入文を指定できる.

■Syntax

[name:] FORALL (*index = lower : upper [: stride] [, mask-expr]*)
    assign-block
END FORALL *[name]*

（互換性：Fortran 95〜）

- *index*：指標変数.
- *lower, upper, stride*：指標変数の添字 3 つ組. 開始値, 終値, 刻み値（整数）. 刻み値を省略すると 1 となる.
- *mask-expr*：配列選別式（マスク式）. 省略可.
- *assign-block*：単一あるいは複数の配列代入文.

ただし, 構文名は省略できる.

■FORALL 構文に関するプログラム例 （f1_forall_3.f90）

```
program f1_forall_3
  implicit none
  integer :: i
  real :: a(6) = 0.0
  a(2:5) = (/ -3.0, -1.0, 1.0, 3.0 /)
  write(*,'(6f8.3)') a(:)
  forall(i=2:6)
     a(i) = a(i-1) ! 同時的代入
     a(i) = abs(a(i))
  end forall
  write(*,'(6f8.3)') a(:)
end program f1_forall_3
```

11　配列組込み関数

11.1　ベクトル・行列演算

表 6 にベクトル演算および行列演算を行う配列組込み関数を示す.

表 6　ベクトル・行列演算の組込み関数

関数	説明（戻り値）	通常の代数式
DOT_PRODUCT(*vector_a*, *vector_b*)	ベクトルの内積（スカラ）	$\mathbf{a}^* \cdot \mathbf{b}$
MATMUL(*matrix_c*, *matrix_d*)	行列の積（配列）	$[c][d]$
TRANSPOSE(*matrix_c*)	行列の転置（配列）	$[c]^t$

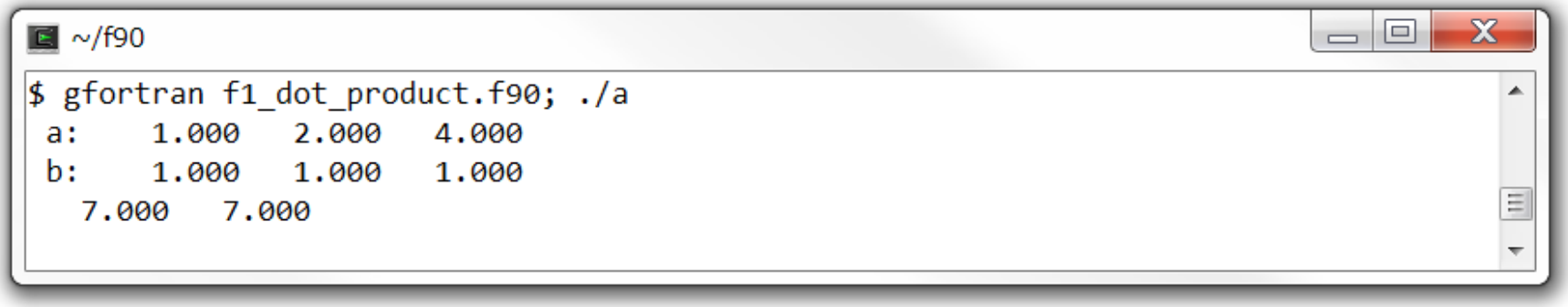
ただし, $\mathbf{a}^*$ は $\mathbf{a}$ の複素共役, $[c]^t$ は $[c]$ の転置行列を示す.

（互換性：Fortran 90〜）

■実数ベクトルの内積に関するプログラム例　引数が実数（f1_dot_product.f90），

```
program f1_dot_product
  implicit none
  integer i
  real(8) a(3),b(3),sp
  a(:) = (/ 1.0D0, 2.0D0, 4.0D0 /)
  b(:) = (/ 1.0D0, 1.0D0, 1.0D0 /)
  sp = 0.0D0
  do i=1,3
     sp = sp+a(i)*b(i)
  enddo
  write(*,'(a4,3f8.3)')'a: ',a(:)
  write(*,'(a4,3f8.3)')'b: ',b(:)
  write(*,'(2f8.3)') sp,dot_product(a,b)
end program f1_dot_product
```

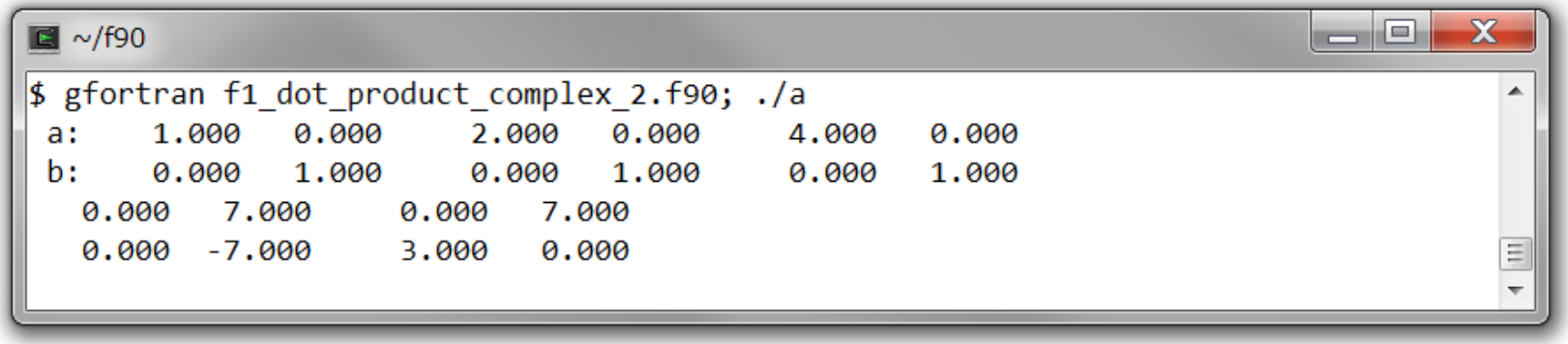
コンパイル，リンク，実行すると，



■複素ベクトルの内積に関するプログラム例　引数が複素数の場合，vector_a の複素共役をとって内積の計算が行われる（f1_dot_product_complex_2.f90）.

```
program f1_dot_product_complex_2
  implicit none
  integer i
  complex(8) :: zr = (1.0D0,0.0D0), zj = (0.0D0,1.0D0)
  complex(8) za(3),zb(3),zc(3),zsp
  za(:) = (/ 1.0D0, 2.0D0, 4.0D0 /)*zr
  zb(:) = (/ 1.0D0, 1.0D0, 1.0D0 /)*zj
  zsp = 0.0D0
  do i=1,3
     zsp = zsp+za(i)*zb(i)
  enddo
  write(*,'(a4,3(2f8.3,2x))')'a: ',za(:)
  write(*,'(a4,3(2f8.3,2x))')'b: ',zb(:)
  write(*,'(2(2f8.3,2x))') zsp,dot_product(za,zb)
  write(*,'(2(2f8.3,2x))') dot_product(zb,za),dot_product(zb,zb)
end program f1_dot_product_complex_2
```

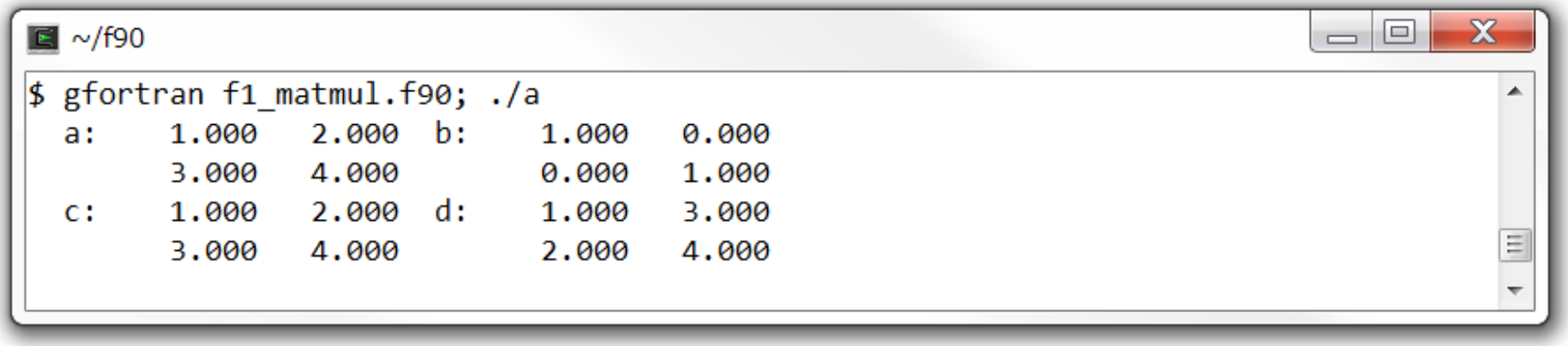
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_dot_product_complex_2.f90; ./a
a: 1.000 0.000 2.000 0.000 4.000 0.000
b: 0.000 1.000 0.000 1.000 0.000 1.000
0.000 7.000 0.000 7.000
0.000 -7.000 3.000 0.000

■実数行列の演算に関するプログラム例　引数の配列が実数の場合（f1_matmul.f90),

```
program f1_matmul
  implicit none
  integer, parameter :: nn = 2
  real(8) a(nn,nn),b(nn,nn),c(nn,nn),d(nn,nn)
  a(1,:) = (/ 1.0D0, 2.0D0 /);   b(1,:) = (/ 1.0D0, 0.0D0 /)
  a(2,:) = (/ 3.0D0, 4.0D0 /);   b(2,:) = (/ 0.0D0, 1.0D0 /)
  c = matmul(a,b)
  d = transpose(a)
  write(*,'(2(2x,a3,2f8.3))') 'a: ',a(1,:),'b: ',b(1,:)
  write(*,'(2(2x,a3,2f8.3))') '   ',a(2,:),'   ',b(2,:)
  write(*,'(2(2x,a3,2f8.3))') 'c: ',c(1,:),'d: ',d(1,:)
  write(*,'(2(2x,a3,2f8.3))') '   ',c(2,:),'   ',d(2,:)
end program f1_matmul
```

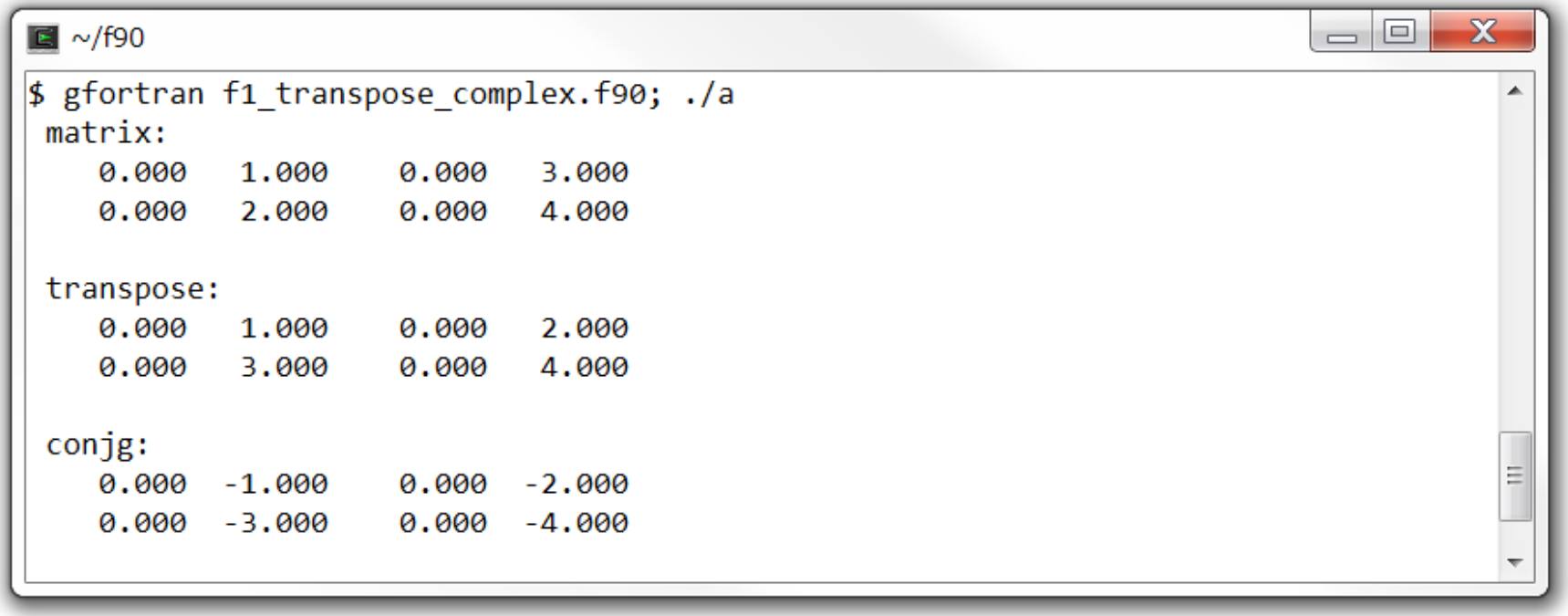
コンパイル，リンク，実行すると，



■複素行列の演算（転置，共役）に関するプログラム例　引数の配列が複素数の場合（f1_transpose_complex.f90），

```
program f1_transpose_compex
  implicit none
  integer, parameter :: nn = 2
  complex(8) za(nn,nn),zb(nn,nn),zc(nn,nn)
  complex(8) :: zj=(0.0D0,1.0D0)

  za(1,:) = (/ 1.0D0, 2.0D0 /)*zj
  za(2,:) = (/ 3.0D0, 4.0D0 /)*zj
  zb = transpose(za)
  zc = conjg(zb)

  write(*,*) 'matrix:'
  write(*,'(2(2(1x,2f8.3)/))') za(:,:)
  write(*,*) 'transpose:'
  write(*,'(2(2(1x,2f8.3)/))') zb(:,:)
  write(*,*) 'conjg:'
  write(*,'(2(2(1x,2f8.3)/))') zc(:,:)
end program f1_transpose_compex
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_transpose_complex.f90; ./a
matrix:
0.000 1.000 0.000 3.000
0.000 2.000 0.000 4.000
transpose:
0.000 1.000 0.000 2.000
0.000 3.000 0.000 4.000
conjg:
0.000 -1.000 0.000 -2.000
0.000 -3.000 0.000 -4.000

11.2　配列要素の演算（集計）

表 7 に配列要素の演算（集計）を行う配列組込み関数を示す.

表 7　配列要素の演算を行う組込み関数

関数	説明，戻り値
SUM（*array [, dim] [, mask]* ） SUM（*array [, mask]* ）	第 *dim* 次元方向に沿って *mask* 要素が真となる位置の要素の和を求め，これらを要素とする配列を返す．配列が 1 次元，または *dim* が省略された場合（全配列要素），スカラー値を返す.
PRODUCT（*array [, dim] [, mask]* ） PRODUCT（*array [, mask]* ）	引数の配列について指定した要素の積を計算する（SUM 関数と同様）. 全配列要素の積を計算する（SUM 関数と同様）.
MAXVAL（*array [, dim] [, mask]* ）	指定した配列要素の中での最大値を返す（配列あるいはスカラ）.
MINVAL（*array [, dim] [, mask]* ）	指定した配列要素の中での最小値を返す（配列あるいはスカラ）.
ALL（*mask [, dim]* ）	*mask* の（第 *dim* 次元方向の）要素がすべて真のとき T（真），それ以外は F（偽）として論理配列を返す.
ANY（*mask [, dim]* ）	*mask* の（第 *dim* 次元方向の）要素が一つでも真のとき T（真），それ以外は F（偽）として論理配列を返す.
COUNT（*mask [, dim]* ）	*mask* の（第 *dim* 次元方向の）要素のうち，真の要素数を配列あるいはスカラで返す.

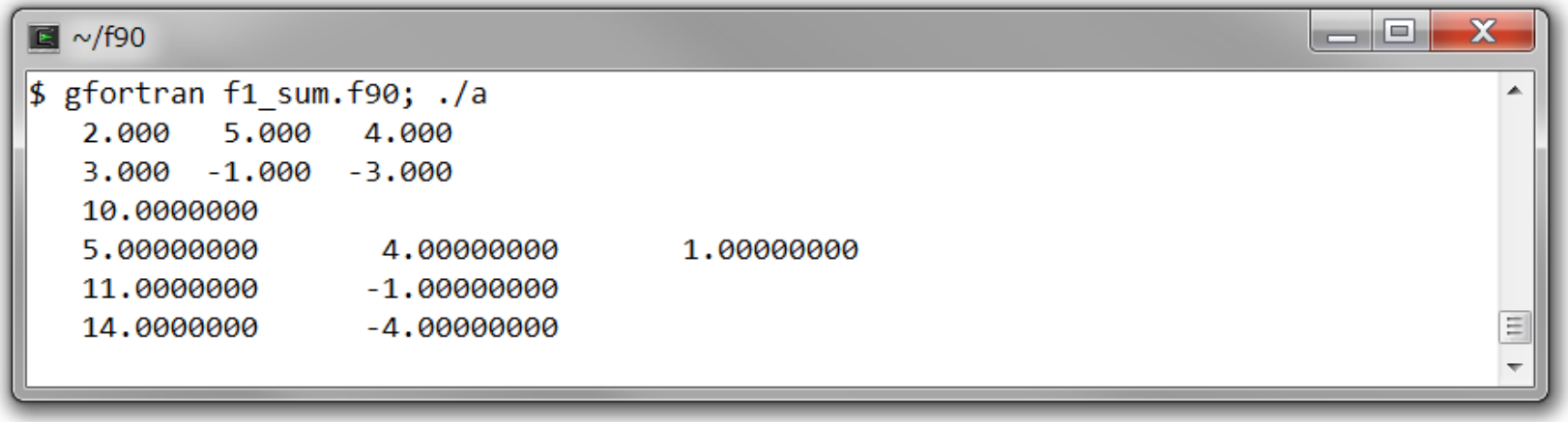
（互換性：Fortran 90～）

ただし，*array* は配列名，*dim* は指定する配列の次元，*mask* は配列要素を選択するための論理配列式.

■SUM 関数に関するプログラム例　（f1_sum.f90）

```
program f1_sum
  implicit none
  real a(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '(3f8.3/3f8.3)', transpose(a)
  print *, sum(a) ! 配列要素全ての和
  print *, sum(a,dim=1) ! 第 1 次元の方向の和
  print *, sum(a,2) ! 第 2 次元の方向の和
  print *, sum(a,mask=a>0.0)& ! 正の要素のみの和
           ,sum(a,a<0.0) ! 負の要素のみの和
end program f1_sum
```

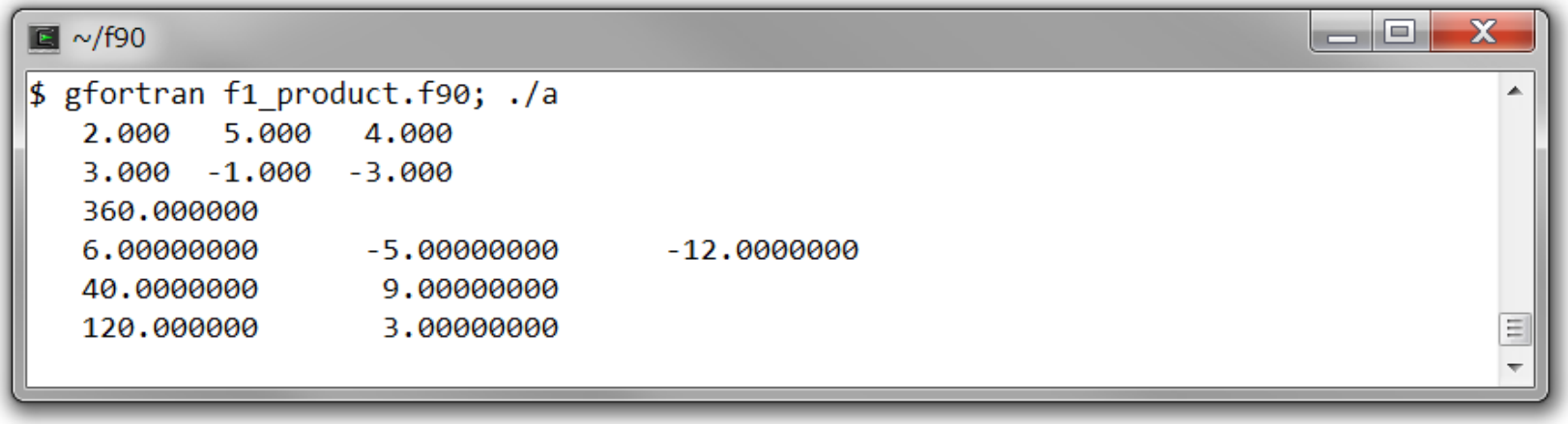
コンパイル，リンク，実行すると，



■PRODUCT 関数に関するプログラム例 （f1_product.f90）

```
program f1_product
  implicit none
  real a(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '(3f8.3/3f8.3)', transpose(a)
  print *, product(a) ! 配列要素全ての積
  print *, product(a,dim=1) ! 第 1 次元の方向の積
  print *, product(a,2) ! 第 2 次元の方向の積
  print *, product(a,mask=a>0.0)& ! 正の要素のみの積
          ,product(a,a<0.0) ! 負の要素のみの積
end program f1_product
```

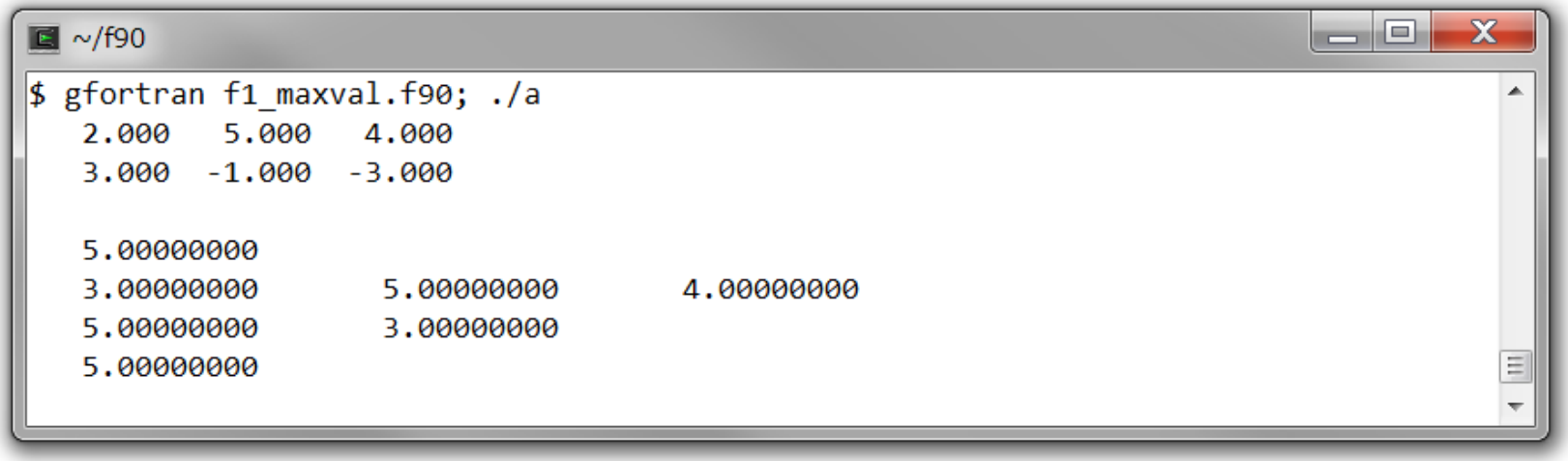
コンパイル，リンク，実行すると，



■MAXVAL 関数に関するプログラム例 （f1_maxval.f90）

```
program f1_maxval
  implicit none
  real a(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '(2(3f8.3/))', transpose(a)
  print *, maxval(a) ! 配列要素全ての中での最大値
  print *, maxval(a,dim=1) ! 第 1 次元の方向の要素の中での最大値
  print *, maxval(a,2) ! 第 2 次元の方向の要素の中での最大値
  print *, maxval(a,mask=a>0.0) ! 正の要素の中での最大値
end program f1_maxval
```

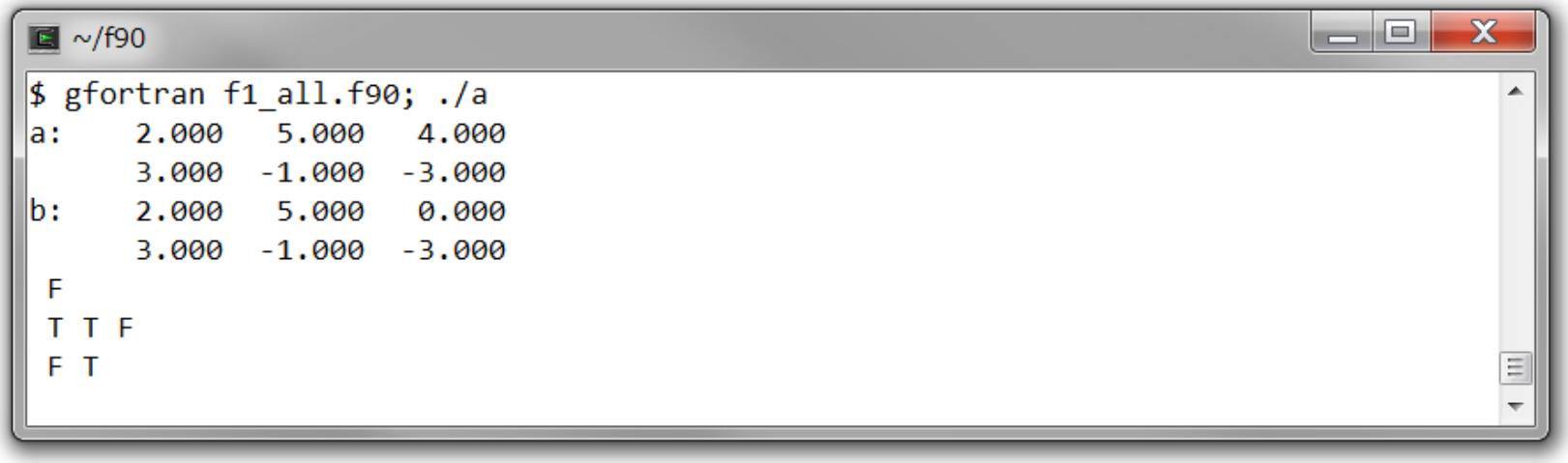
コンパイル，リンク，実行すると，



■ALL 関数に関するプログラム例 （f1_all.f90）

```
program f1_all
  implicit none
  real a(2,3),b(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '("a: ",3f8.3/"   ",3f8.3)', transpose(a)
  b(:,:) = a(:,:); b(1,3) = 0.0
  print '("b: ",3f8.3/"   ",3f8.3)', transpose(b)
  print *, all(mask=a==b)
  print *, all(mask=a==b,dim=1)
  print *, all(a==b,2)
end program f1_all
```

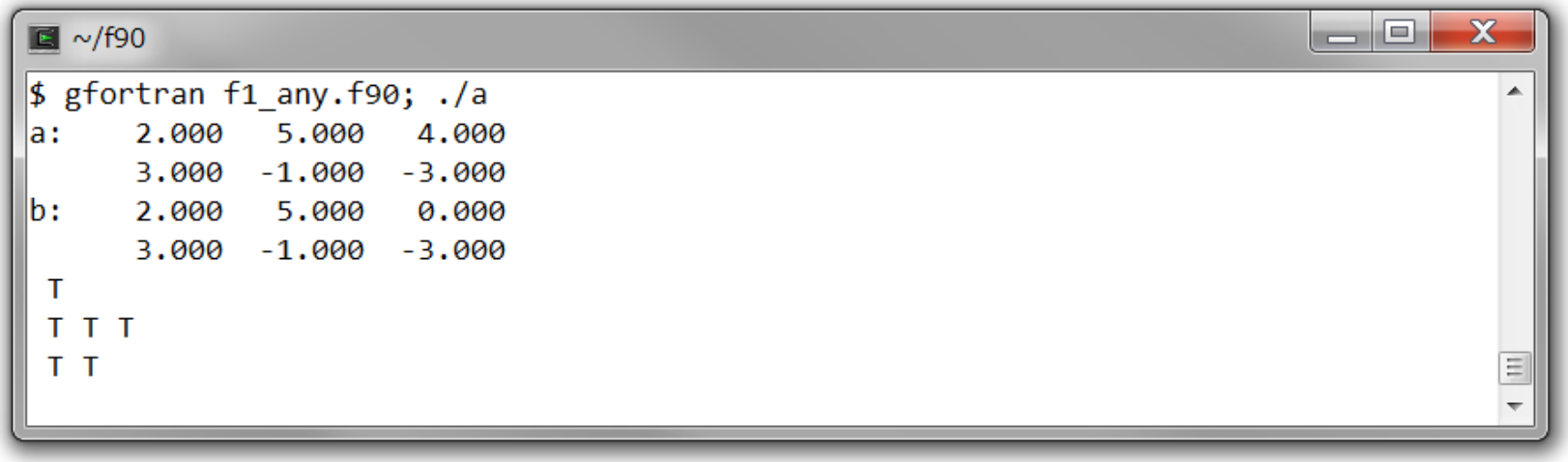
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_all.f90; ./a
a: 2.000 5.000 4.000
    3.000 -1.000 -3.000
b: 2.000 5.000 0.000
    3.000 -1.000 -3.000
 F
 T T F
 F T

■ANY 関数に関するプログラム例 （f1_any.f90）

```
program f1_any
  implicit none
  real a(2,3),b(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '("a: ",3f8.3/"   ",3f8.3)', transpose(a)
  b(:,:) = a(:,:); b(1,3) = 0.0
  print '("b: ",3f8.3/"   ",3f8.3)', transpose(b)
  print *, any(mask=a==b)
  print *, any(mask=a==b,dim=1)
  print *, any(a==b,2)
end program f1_any
```

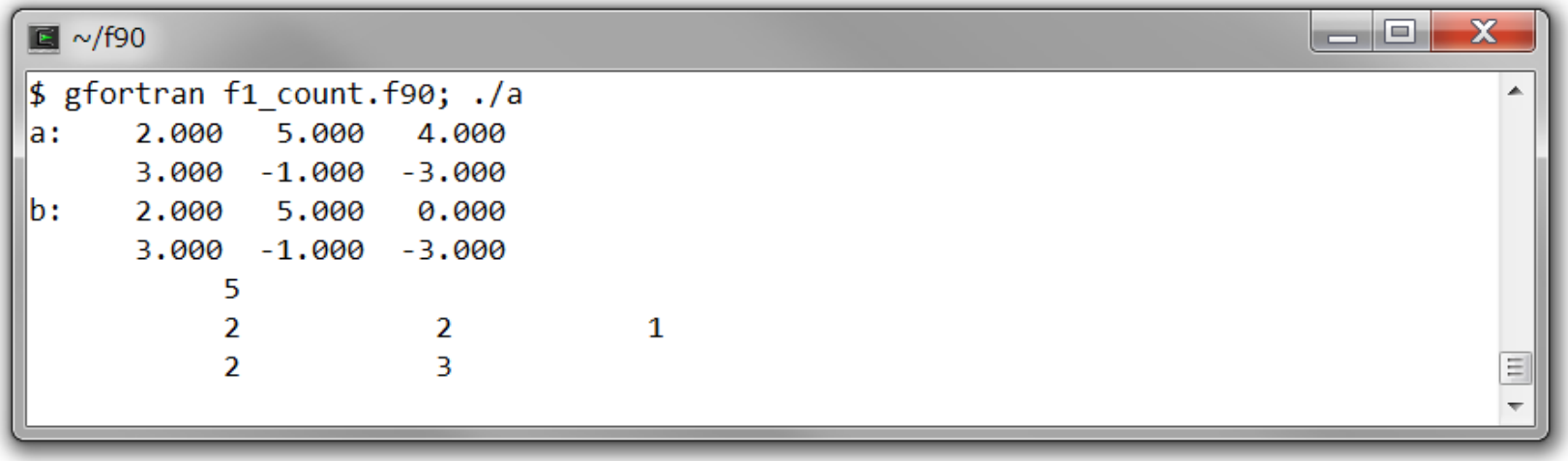
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_any.f90; ./a
a: 2.000 5.000 4.000
    3.000 -1.000 -3.000
b: 2.000 5.000 0.000
    3.000 -1.000 -3.000
 T
 T T T
 T T

■COUNT 関数に関するプログラム例 （f1_count.f90）

```
program f1_count
  implicit none
  real a(2,3),b(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '("a: ",3f8.3/"   ",3f8.3)', transpose(a)
  b(:,:) = a(:,:); b(1,3) = 0.0
  print '("b: ",3f8.3/"   ",3f8.3)', transpose(b)
  print *, count(mask=a==b)
  print *, count(mask=a==b,dim=1)
  print *, count(a==b,2)
end program f1_count
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_count.f90; ./a
a: 2.000 5.000 4.000
    3.000 -1.000 -3.000
b: 2.000 5.000 0.000
    3.000 -1.000 -3.000
          5
          2   2   1
          2   3

11.3　配列に関する問い合わせ関数

表 8 に配列に関する問い合わせ関数を示す.

表 8　配列に関する問い合わせ関数

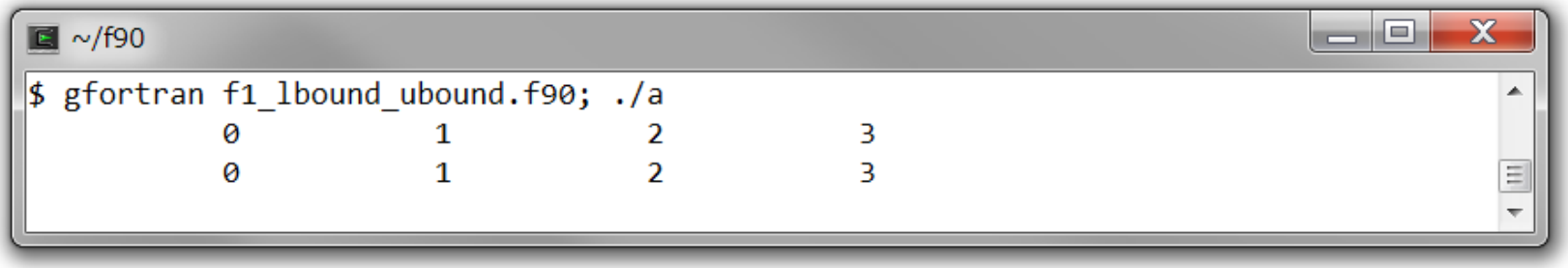
関数	説明，戻り値
LBOUND（*array [, dim]* ）	配列 *array* について，指定した次元の添字の下限値を返す.
UBOUND（*array [, dim]* ）	配列 *array* について，指定した次元の添字の上限値を返す.
SIZE（*array [, dim]* ）	配列 *array* について，指定した次元の要素数を返す.
SHAPE（*array* ）	配列 *array* の形状（配列の各次元を要素とする 1 次元整数配列）を返す.
MAXLOC（*array [, dim] [, mask]* ） MAXLOC（*array [, dim]* ）	配列 *array* について，指定した要素の中での最大値の位置を示す添字を，配列あるいはスカラで返す.
MINLOC（*array [, dim] [, mask]* ） MINLOC（*array [, dim]* ）	配列 *array* について，指定した要素の中での最小値の位置を示す添字を，配列あるいはスカラで返す.

ただし， *array* は配列名， *dim* は指定する配列の次元， *mask* は配列要素を選択するための論理配列式.

■LBOUND 関数，UBOUND 関数に関するプログラム例 （f1_lbound_ubound.f90）

```
program f1_lbound_ubound
  implicit none
  real a(0:2,3)
  integer n1,n2,nn(2)
  n1 = lbound(a,dim=1)
  n2 = lbound(a,2)
  print *,n1,n2,ubound(a,dim=1),ubound(a,2)
  nn(:) = lbound(a)
  print *,nn,ubound(a)
end program f1_lbound_ubound
```

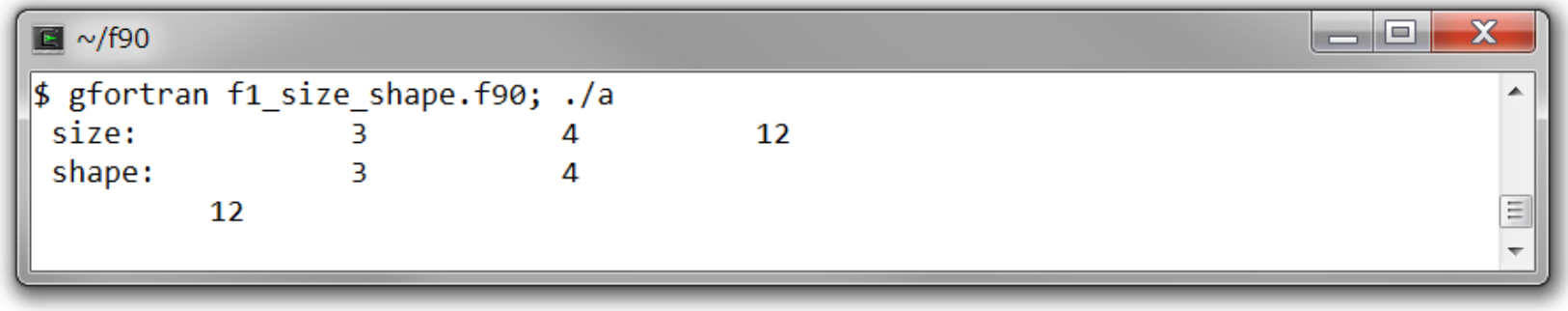
コンパイル，リンク，実行すると，



■SIZE 関数，SHAPE 関数に関するプログラム例 （f1_size_shape.f90）

```
program f1_size_shape
  implicit none
  real a(0:2,4)
  integer n1,n2,nn(2)
  n1 = size(a,dim=1)
  n2 = size(a,2)
  print *,'size: ',n1,n2,size(a)
  nn(:) = shape(a)
  print *,'shape:',nn(:)
  print *,product(nn)
end program f1_size_shape
```

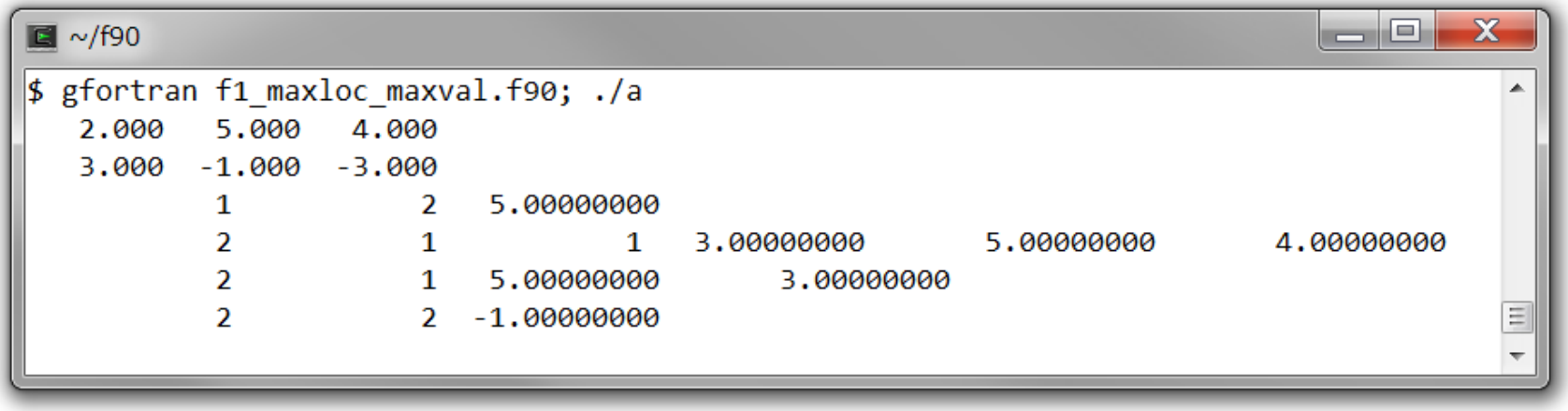
コンパイル，リンク，実行すると，



■MAXLOC 関数，MAXVAL 関数に関するプログラム例 （f1_maxloc_maxval.f90）

```
program f1_maxloc_maxval
  implicit none
  real a(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '(3f8.3)', transpose(a)
  print *, maxloc(a),maxval(a) ! 配列要素全ての中での最大
  print *, maxloc(a,dim=1),maxval(a,dim=1) ! 第 1 次元の方向の要素の中での最大
  print *, maxloc(a,2),maxval(a,2) ! 第 2 次元の方向の要素の中での最大
  print *, maxloc(a,mask=a<0.0),maxval(a,mask=a<0.0) ! 負の要素の中での最大
end program f1_maxloc_maxval
```

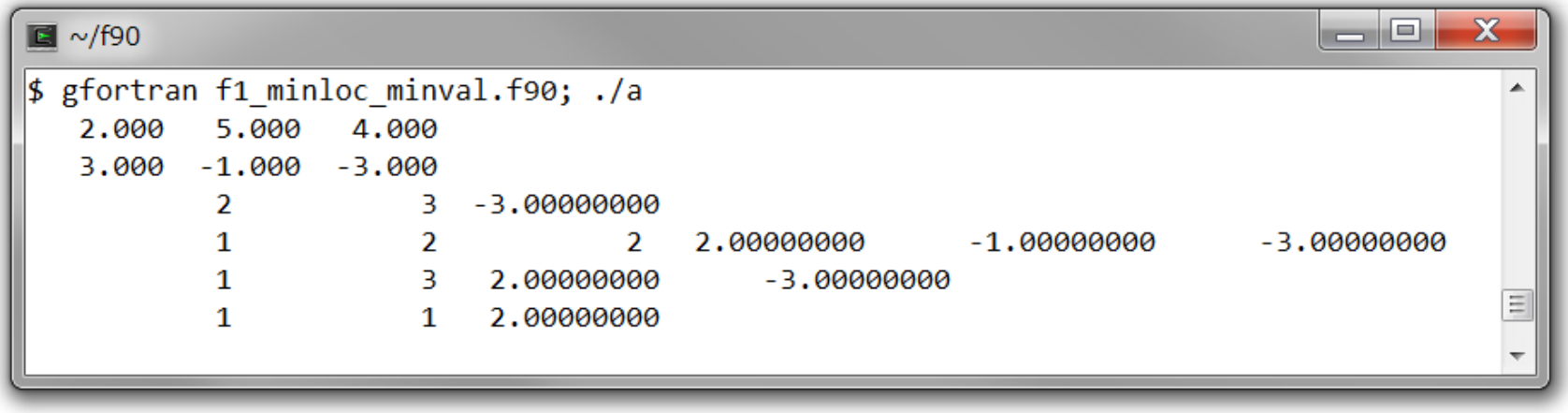
コンパイル，リンク，実行すると，



■MINLOC 関数，MINVAL 関数に関するプログラム例 （f1_minloc_minval.f90）

```
program f1_minloc_minval
  implicit none
  real a(2,3)
  a(1,:) = (/ 2.0,  5.0,  4.0 /)
  a(2,:) = (/ 3.0, -1.0, -3.0 /)
  print '(3f8.3)', transpose(a)
  print *, minloc(a),minval(a) ! 配列要素全ての中での最小
  print *, minloc(a,dim=1),minval(a,dim=1) ! 第 1 次元の方向の要素の中での最小
  print *, minloc(a,2),minval(a,2) ! 第 2 次元の方向の要素の中での最小
  print *, minloc(a,mask=a>0.0),minval(a,mask=a>0.0) ! 正の要素の中での最小
end program f1_minloc_minval
```

コンパイル，リンク，実行すると，



11.4　配列構成・操作関数

表 9 に配列構成関数，表 10 に配列操作関数を示す．

表 9　配列構成関数

関数	説明，戻り値
MERGE（*tsource, fsource, mask*）	*mask* の要素が真なら *tsource* の要素，偽なら *fsource* の要素を選択して構成した配列を返す（*tsource*，*fsource* は同じ型・形状）.
PACK（*array, mask [, vector]*）	配列 *array* の配列要素を選別式に応じて 1 次元配列に詰め込む. *vector* の要素数が戻り値より大きい場合，*vector* の要素が返される？.
UNPACK（*vector, mask, field*）	*mask* で指定した選別式が真のとき，*field* で指定した配列に 1 次元配列の値を順次代入する.
SPREAD（*array, dim, ncopies*）	指定した配列を指定した回数 *ncopies* だけコピーして次元を拡張する.

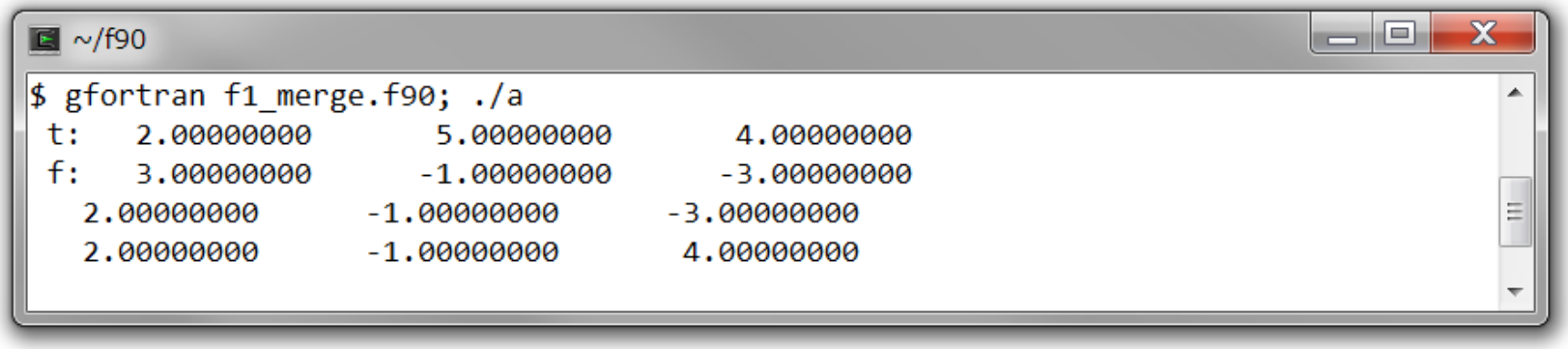
表 10　配列操作関数

関数	説明，戻り値
RESHAPE（*array, shape*）	*array* の要素を 1 次元整数配列 *shape* で定義される形状の配列に組み替える.
CSHIFT（*array, shift [, dim]*）	配列要素の値を *shift* の数だけ *dim* 方向に循環移動する.
EOSHIFT（*array, shift [, boundary, dim]*）	配列要素の値を *shift* の数だけ *dim* 方向に移動し，空いた部分を 0 とする（または *boundary* で指定）.

■MERGE 関数に関するプログラム例 （f1_merge.f90）

```
program f1_merge
  implicit none
  real :: t(3),f(3)
  logical :: mtf(3) = [.TRUE.,.FALSE.,.TRUE.]
  t(:) = [ 2.0,  5.0,  4.0 ]
  f(:) = [ 3.0, -1.0, -3.0 ]
  print *, 't:',t(:)
  print *, 'f:',f(:)
  print *, merge(t,f,mask=f>0)
  print *, merge(t,f,mtf)
end program f1_merge
```

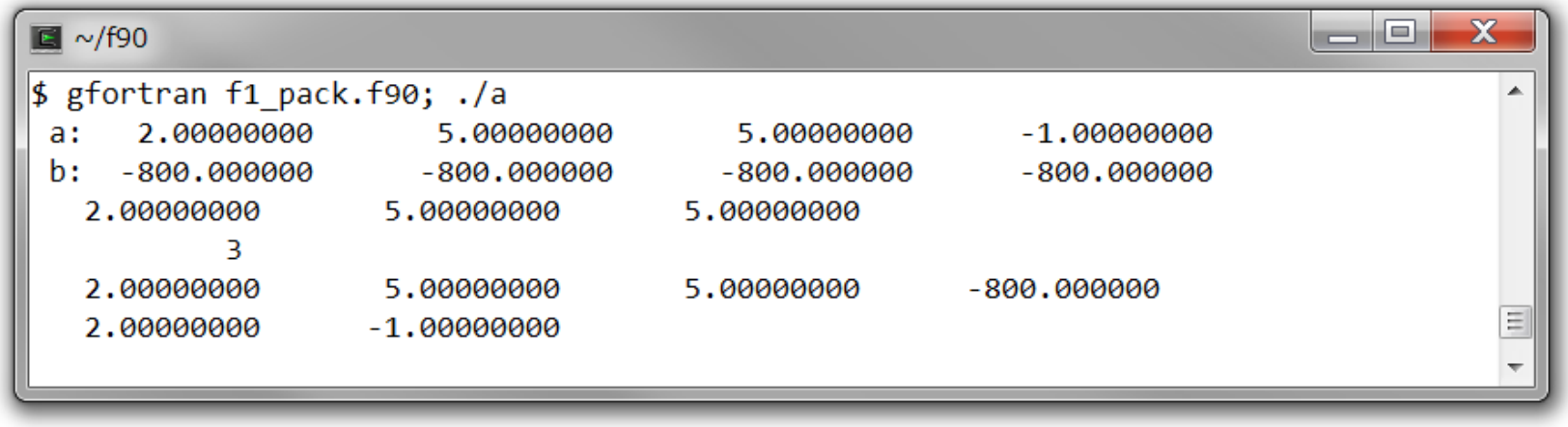
コンパイル，リンク，実行すると，



■PACK 関数に関するプログラム例　（f1_pack.f90）

```
program f1_pack
  implicit none
  real :: a(4),b(4)=-800.0
  a(:) = [ 2.0,  5.0,  5.0, -1.0 ]
  print *, 'a:',a(:)
  print *, 'b:',b(:)
  print *, pack(a,mask=a>0.0)
  print *, size(pack(a,mask=a>0.0))
  print *, pack(a,mask=a>0.0,vector=b)
  print *, pack(a,mask=a<maxval(a))
end program f1_pack
```

コンパイル，リンク，実行すると，



■UNPACK 関数に関するプログラム例　（f1_unpack.f90）

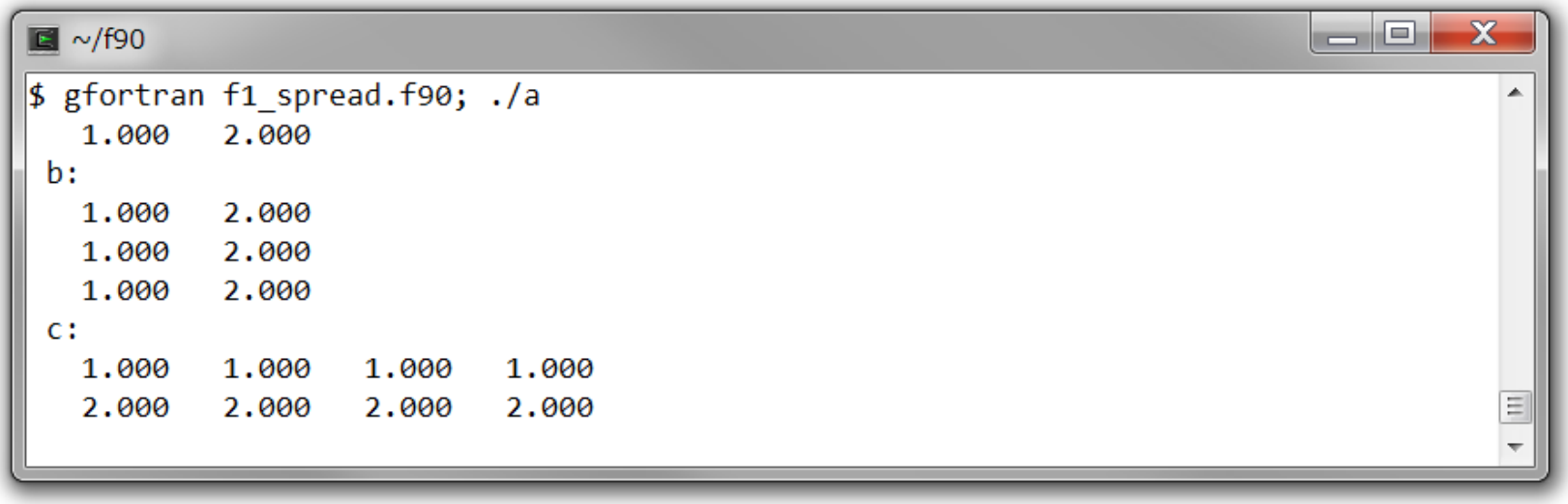
```
program f1_unpack
  implicit none
  real :: v(3),a(2,3),y(2,3)
  v(:) = [ 10.0, 20.0, 30.0 ]
  print '(3f8.3)',v(:)
  a(1,:) = [ -1.0, -3.0,  5.0 ]
  a(2,:) = [  2.0,  4.0, -6.0 ]
  print '(3f8.3)',transpose(a)
  y = unpack(v,a<=0,field=a)
  print '(3f8.3)',transpose(y)
end program f1_unpack
```

コンパイル，リンク，実行すると，

■SPREAD 関数に関するプログラム例　（f1_spread.f90）

```
program f1_spread
  implicit none
  real :: a(2) = [ 1.0, 2.0 ], b(3,2), c(2,4)
  print '(2f8.3)',a(:)
  b = spread(a, dim=1, ncopies=3)
  print *,'b:'
  print '(2f8.3)',transpose(b)
  c = spread(a, dim=2, ncopies=4)
  print *,'c:'
  print '(4f8.3)',transpose(c)
end program f1_spread
```

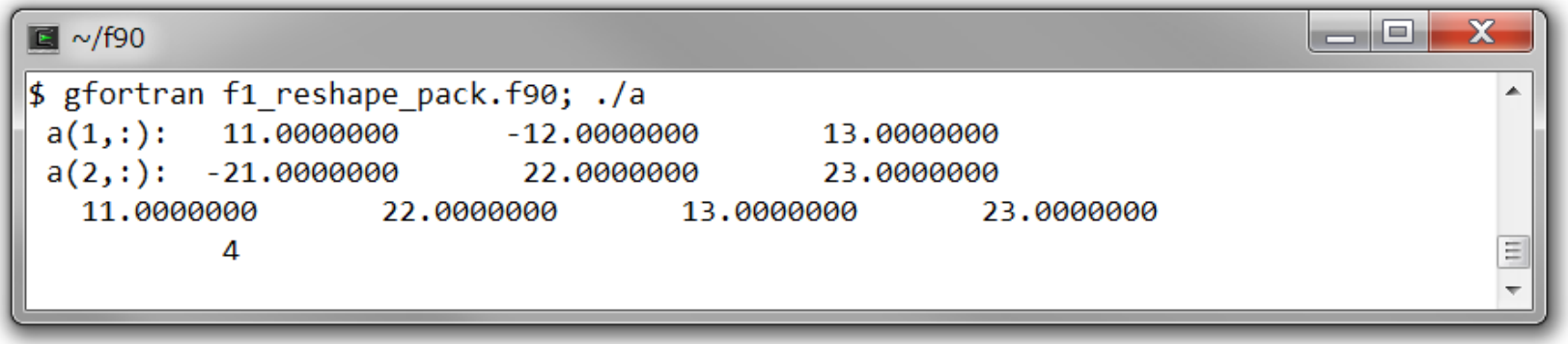
コンパイル，リンク，実行すると，



■RESHAPE 関数に関するプログラム例 （f1_reshape_pack.f90）

```
program f1_reshape_pack
  implicit none
  real :: a(2,3)
  a = reshape( [ 11.0, -21.0, -12.0, 22.0, 13.0, 23.0 ], [2, 3] )
  print *, 'a(1,:):',a(1,:)
  print *, 'a(2,:):',a(2,:)
  print *, pack(a,mask=a>0.0)
  print *, size(pack(a,mask=a>0.0))
end program f1_reshape_pack
```

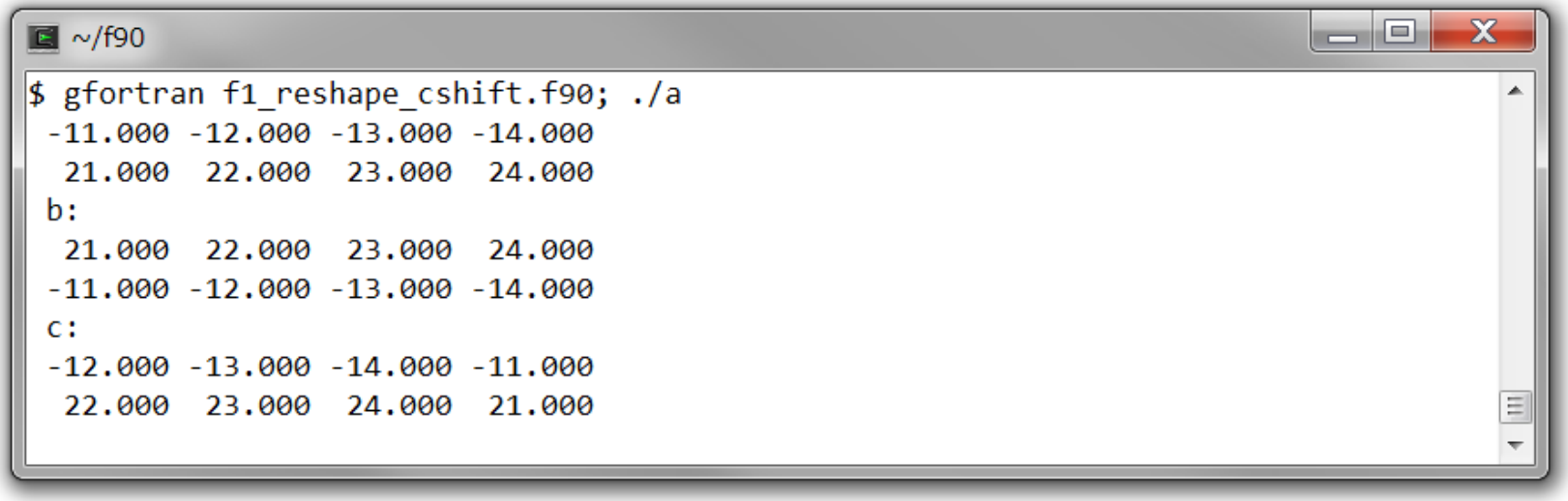
コンパイル，リンク，実行すると，



■CSHIFT 関数に関するプログラム例 （f1_reshape_cshift.f90）

```
program f1_reshape_cshift
  implicit none
  real, dimension(2,4) :: a,b,c
  a = reshape( [ -11., 21., -12., 22., -13., 23., -14., 24. ], [2, 4] )
  print '(4f8.3)',transpose(a)
  b = cshift(a,1)
  print *,'b:'; print '(4f8.3)',transpose(b)
  c = cshift(a,shift=1,dim=2)
  print *,'c:'; print '(4f8.3)',transpose(c)
end program f1_reshape_cshift
```

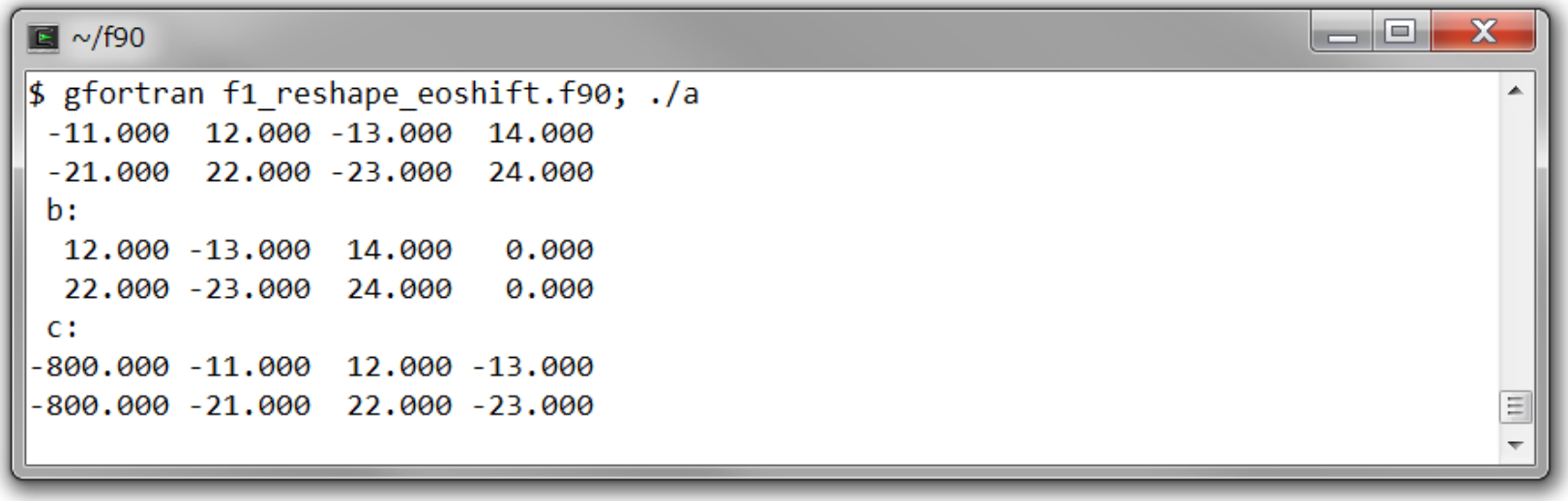
コンパイル，リンク，実行すると，



■EOSHIFT 関数に関するプログラム例 （f1_reshape_eoshift.f90）

```
program f1_reshape_eoshift
  implicit none
  real, dimension(2,4) :: a,b,c
  a = reshape( [ -11., -21., 12., 22., -13., -23., 14., 24. ], [2, 4] )
  print '(4f8.3)', transpose(a)
  b = eoshift(a,shift=1,dim=2)
  print *,'b:'; print '(4f8.3)', transpose(b)
  c = eoshift(a,shift=-1,boundary=[-800.0, -800.0],dim=2)
  print *,'c:'; print '(4f8.3)', transpose(c)
end program f1_reshape_eoshift
```

コンパイル，リンク，実行すると，



12　文関数

文関数（Statement function）はただ一つの文を用いて定義する関数である．定義しているプログラム単位内でのみ，組み込み関数と同様の使い方で計算できる．

■Syntax

stmt-func (*d-arg-1 [, d-arg-2, ...]*) = *scalar-expr*

（互換性：FORTRAN 77〜）

- *stmt-func*：文関数名．
- *d-arg-i　(i=1,2, ...)*：仮引数（dummy argument）．
- *scalar-expr*：スカラ式．

宣言文より後（*stme-func*，*d-arg-i* の宣言が必要），実行文より前に書く．文関数の実行は，次のようになる．

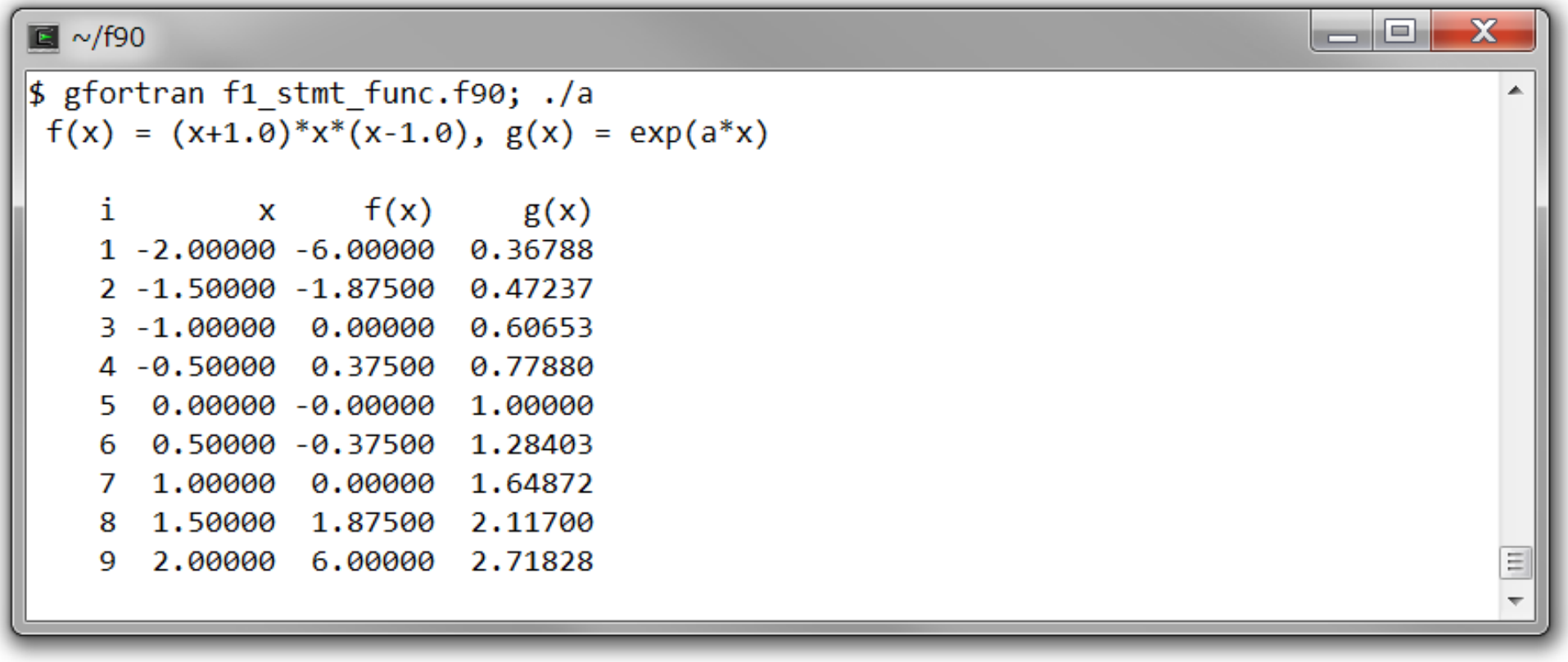
```
stmt-func(a-arg-1 [, a-arg-2, ... ])
```

ただし，`a-arg-1`，`a-arg-2` は実引数（actual argument）．

■文関数に関するプログラム例１　（f1_stmt_func.f90）

```
program f1_stmt_func
  implicit none
  integer, parameter :: nn=101
  real :: xx(nn),ff(nn),gg(nn), x,f,g,a
  real :: x0 = -2.0, dx = 0.5
  integer :: i,nx = 9
  f(x) = (x+1.0)*x*(x-1.0)
  g(x) = exp(a*x)
  a = 0.5
  do i=1,nx
     x = x0+dx*(i-1)
     xx(i) = x
     ff(i) = f(x)
     gg(i) = g(x)
  enddo
  write(6,*) 'f(x) = (x+1.0)*x*(x-1.0), g(x) = exp(a*x)'
  write(6,'(/1x,a4,3(1x,a8))') 'i','x','f(x)','g(x)'
  do i=1,nx
     write(6,'(1x,i4,3(1x,f8.5))') i,xx(i),ff(i),gg(i)
  enddo
  stop
end program f1_stmt_func
```

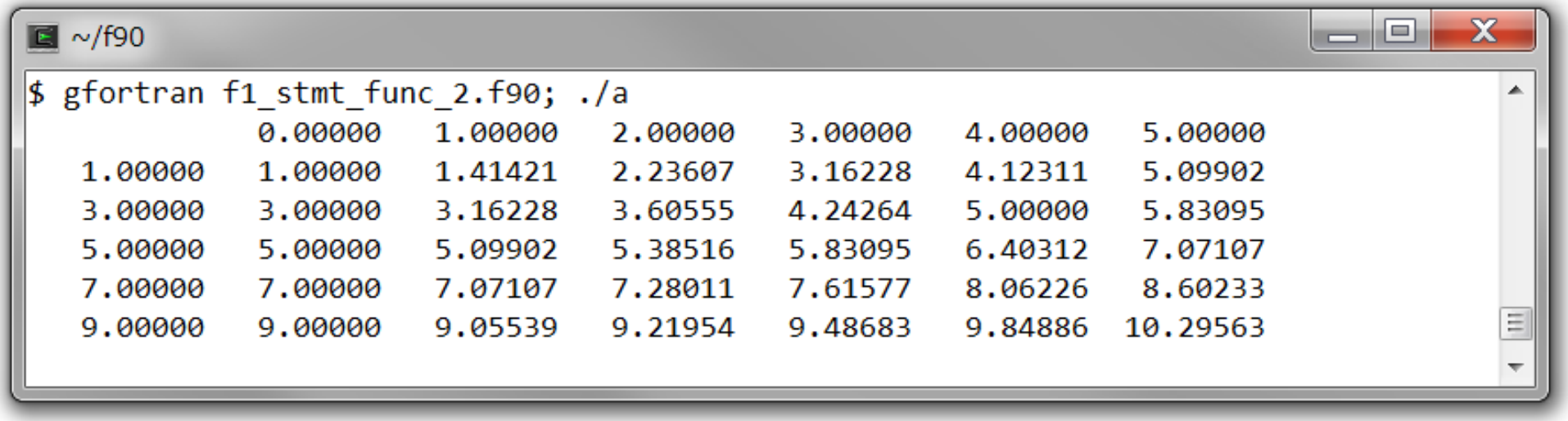
コンパイル，リンク，実行すると，



■文関数に関するプログラム例２ （f1_stmt_func_2.f90）

```
program f1_stmt_func_2
  implicit none
  integer, parameter :: nn=101
  real :: xx(nn),yy(nn),ff2(nn,nn),f2,x,y
  real :: x0 = 1.0, dx = 2.0, y0 = 0.0, dy = 1.0
  integer :: i,j, nx = 5, ny = 6
  f2(x,y) = sqrt(x**2+y**2)

  xx(1:nx) = [ (x0+dx*(i-1),i=1,nx) ]
  yy(1:ny) = [ (y0+dy*(j-1),j=1,ny) ]
  do j=1,ny
     do i=1,nx
        ff2(i,j) = f2(xx(i),yy(j))
     enddo
  enddo
  print '(10x,6f10.5)',yy(1:ny)
  do i=1,nx
     print '(7f10.5)',xx(i),(ff2(i,j),j=1,ny)
  enddo
  stop
end program f1_stmt_func_2
```

コンパイル，リンク，実行すると，



13　外部関数副プログラム

13.1　FUNCTION 文

組み込み関数や文関数と同様の使い方で計算できる.

13.1.1　FUNCTION 文で型宣言を行う場合

■Syntax

```
TYPE-f FUNCTION func (d-arg-1 [, d-arg-2, ... ] )
      TYPE-a  d-arg-1 [, d-arg-2, ... ]
     ...
      func = stmt
     ...
      RETURN
END [FUNCTION [func] ]
```

（互換性：FORTRAN 77～）

- *TYPE-f, TYPE-a*：関数の型，仮引数の型.
- *func*：関数名.

- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）.
- *expr*：スカラ式.

関数副プログラムの実行は，次のようになる.

```
func(a-arg-1 [, a-arg-2, ... ])
```

ただし，`a-arg-1`，`a-arg-2` は実引数（actual argument）．例えば,

```
real function func1(x,a)
  implicit none
  real x,a
  func1 = exp(a*x)
  return
end function func1
```

13.1.2 FUNCTION 文で型宣言を行わない場合

■Syntax

```
FUNCTION func (d-arg-1 [, d-arg-2, ... ] )
      TYPE-f  func
      TYPE-a  d-arg-1 [, d-arg-2, ... ]
     ...
      func = stmt
     ...
      RETURN
END [FUNCTION [func] ]
```

（互換性：FORTRAN 77～）

例えば,

```
function func2(x,a)
  implicit none
  real func2,x,a
  func2 = exp(a*x)
end function func2
```

RETURN 文は省略できる.

13.1.3　外部関数の呼び出し

主プログラムとは別の独立したプログラム単位として扱った関数副プログラムは，外部副プログラム（external subprogram）という．例えば，

```
PROGRAM prog-name ! 主プログラム
    ...
     func (a-arg-1 [, a-arg-2, ... ] ) ! 関数の呼び出し
    ...
END [PROGRAM [prog-name] ]

TYPE-f FUNCTION func (d-arg-1 [, d-arg-2, ... ] ) ! 外部（関数）副プログラム
    ...
END [FUNCTION [func] ]
```

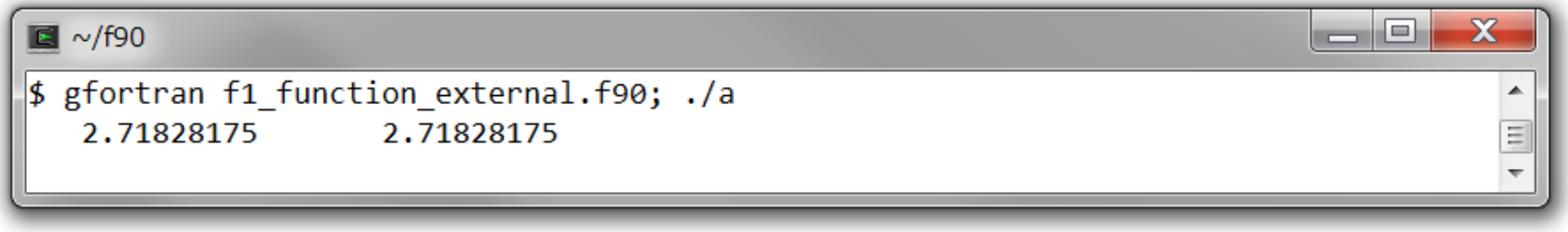
（互換性：FORTRAN 77～）

関数の呼び出し側で，関数の型宣言が必要となる．定義した変数は，各々独立である．

■外部関数副プログラムに関するプログラム例 （f1_function_external.f90）

```
program f1_function_external ! 主プログラム
  implicit none
  real :: func1,func2, a=2.0
  write(*,*) func1(0.5,a),func2(0.5,a) ! 関数の呼び出し
  stop
end program f1_function_external

real function func1(x,a) ! 外部関数副プログラム
  implicit none
  real x,a
  func1 = exp(a*x)
  return
end function func1

function func2(x,a) ! 外部関数副プログラム
  implicit none
  real func2,x,a
  func2 = exp(a*x)
end function func2
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_function_external.f90; ./a
2.71828175 2.71828175

13.1.4 RESULT 句

戻り値として，スカラだけでなく配列を設定できる．ただし，戻り値を配列にする場合，明示的引用仕様（後述する内部副プログラムあるいはモジュール等）でなければならない．

■Syntax

```
FUNCTION func (d-arg-1 [, d-arg-2, ... ] ) RESULT (r-name)
     TYPE-f  r-name
     TYPE-a  d-arg-1 [, d-arg-2, ... ]
    ...
     r-name = ...
    ...
RETURN
END [FUNCTION [func] ]
```

（互換性：Fortran 90～）

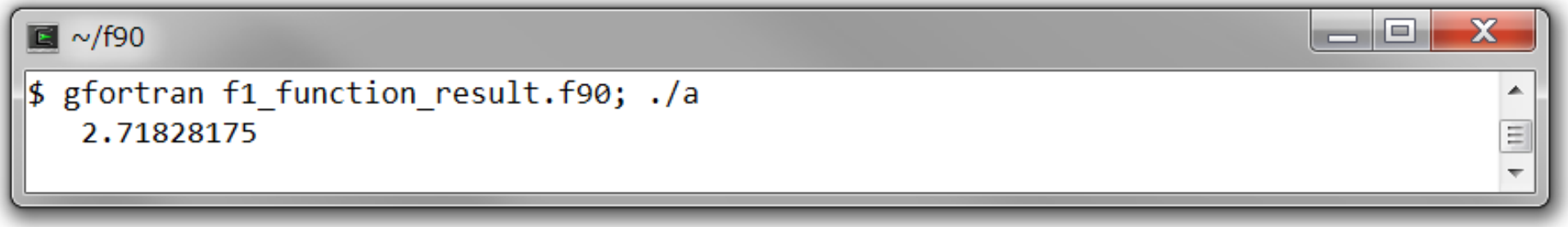
- *TYPE-f, TYPE-a*：関数の型，仮引数の型．
- *func*：関数名．
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）．
- *r-name*：戻り値．

■RESULT 句を用いた外部関数副プログラム例（戻り値がスカラ式）

（f1_function_result.f90）

```
program f1_function_result ! 主プログラム
  implicit none
  real :: a=2.0
  real, external :: func3
  Print *, func3(0.5,a) ! 関数の呼び出し
  stop
end program f1_function_result
function func3(x,a) result(y) ! 関数副プログラム
  implicit none
  real x,a,y
  y = exp(a*x)
  return
end function func3
```

コンパイル，リンク，実行すると，



13.2 COMMON 文

２つ以上のプログラム単位の間で，使用する変数または配列の対応するものどうしに同じ記憶場所を割り当てる.

■Syntax

```
COMMON [/common-block-name/] var-list
```

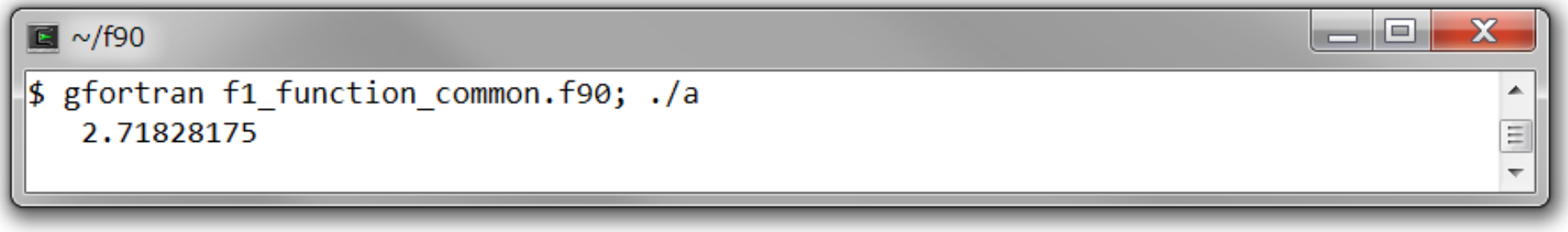
（互換性：FORTRAN 77〜）

- *common-block-name*：コモンブロック名.
- *var-list*：変数名，配列名の並び（異なるプログラム単位で，並びが一致していれば，名称が異なっていてもよい）.

■COMMON 文に関するプログラム例１ （f1_function_common.f90）

```
program f1_function_common ! 主プログラム
  implicit none
  real a,bb,funcc
  common /cb/ a,bb ! コモン文
  a = 2.0; bb = 1.0
  write(*,*) funcc(0.5) ! 関数の呼び出し
  stop
end program f1_function_common
real function funcc(x) ! 外部関数副プログラム
  implicit none
  real x,a,b
  common /cb/ a,b ! コモン文
  funcc = exp(a*x)/b
  return
end function funcc
```

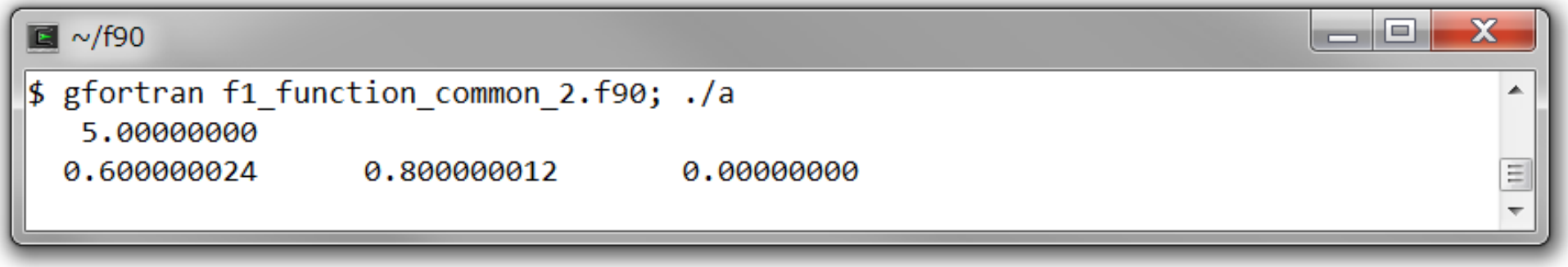
コンパイル，リンク，実行すると，



■COMMON 文に関するプログラム例２ （f1_function_common_2.f90）

```
program f1_function_common_2 ! 主プログラム
  implicit none
  real uvec(3),funcr
  common /cuvec/ uvec ! コモン文
  print *, funcr(3.0, 4.0, 0.0)
  print *, uvec(:)
  stop
end program f1_function_common_2
real function funcr(x,y,z) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: x,y,z
  real r,uvec(3)
  common /cuvec/ uvec ! コモン文
  r = sqrt(x**2+y**2+z**2)
  if(r.ne.0.0) then
     uvec(1) = x/r; uvec(2) = y/r; uvec(3) = z/r
  else
     uvec(1) = 0.0; uvec(2) = 0.0; uvec(3) = 1.0
  endif
  funcr = r
  return
end function funcr
```

コンパイル，リンク，実行すると，



13.3　属性

13.3.1　SAVE

副プログラムにおいて，ある呼び出しから次の呼び出しまで値を保持する.

■Syntax

```
SAVE var-list
SAVE /common-block-name/, ...
```

（互換性：FORTRAN 77〜）

- *var-list*：変数名，配列名の並び.
- *common-block-name*：コモンブロック名.

また，属性指定では，

```
TYPE, SAVE :: var-list
```

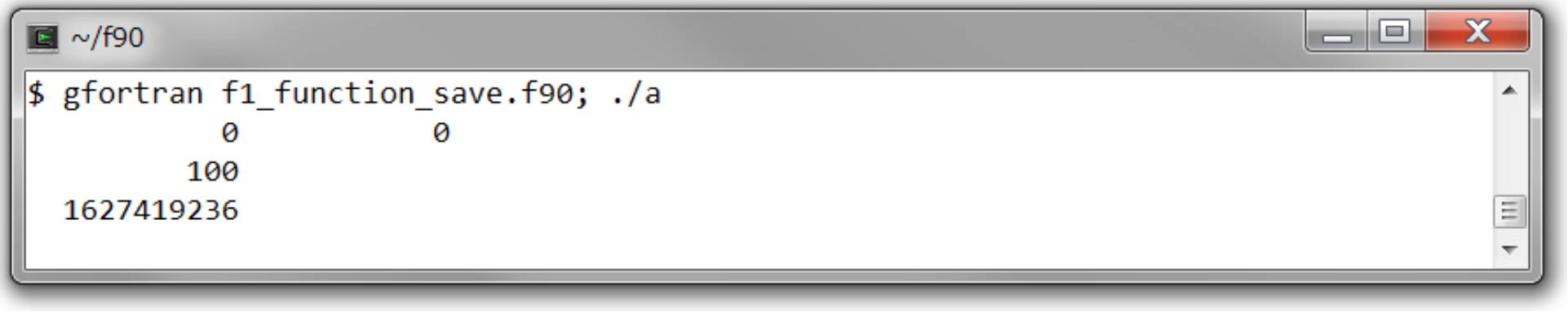
（互換性：Fortran 90〜）

ただし，初期値を設定した場合は，自動的に save 属性をもつ.

■SAVE 文に関するプログラム例 1 （f1_function_save.f90）

```
program f1_function_save ! 主プログラム
  implicit none
  integer jfunc,kfunc
  write(*,*) jfunc(0),kfunc(0)
  write(*,*) jfunc(100)
  write(*,*) kfunc(100)
end program f1_function_save
integer function jfunc(i) ! 関数副プログラム
  implicit none
  integer i,j
  save j ! save 文
  if(i==0) j = 0
  j = j+i; jfunc = j
end function jfunc
integer function kfunc(i) ! 関数副プログラム
  implicit none
  integer i,k
  if(i==0) k = 0
  k = k+i; kfunc = k
end function kfunc
```

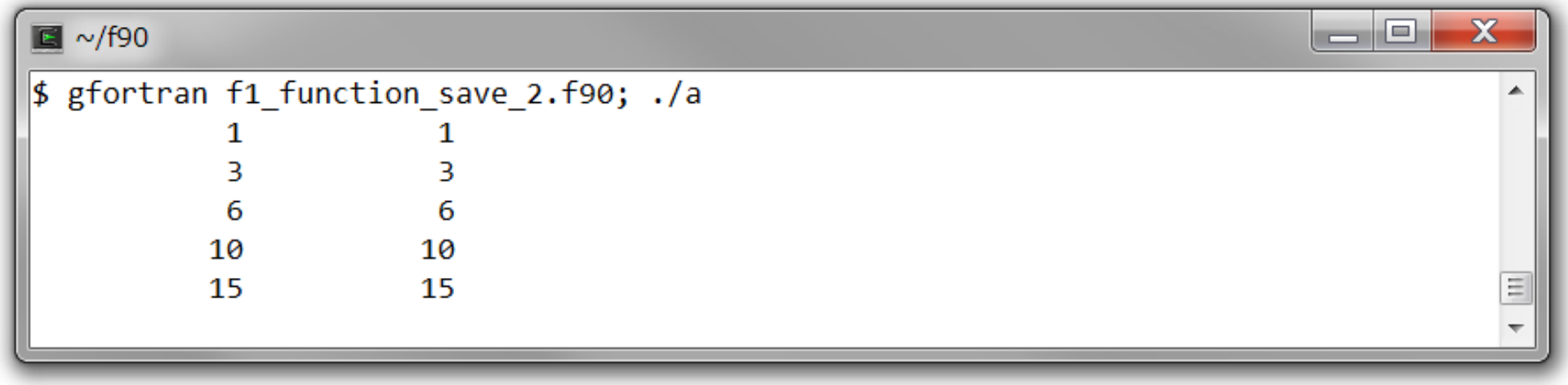
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_function_save.f90; ./a
0 0
100
1627419236

■SAVE 文に関するプログラム例 2 （f1_function_save_2.f90）

```
program f1_function_save_2 ! 主プログラム
  implicit none
  integer :: i
  integer, external :: jfunc,kfunc
  do i=1,5
    print *,jfunc(i),kfunc(i)
  enddo
  stop
end program f1_function_save_2
integer function jfunc(i) ! 関数副プログラム
  integer, save :: j=0 ! save 属性
  integer :: i
  j = j+i
  jfunc = j
  return
end function jfunc
integer function kfunc(i) ! 関数副プログラム
  integer :: k=0 ! 初期値を設定したとき，save 属性を自動的にもつ。
  k = k+i
  kfunc = k
  return
end function kfunc
```

コンパイル，リンク，実行すると，



13.3.2 INTENT

仮引数に，入力・出力・入出力の 3 種類の属性を設定し（推奨），後述する新機能や文法上の改善などがなされている.

■Syntax

TYPE, INTENT(*intent-spec*) :: *d-arg-1 [,d-arg-2, ...]*

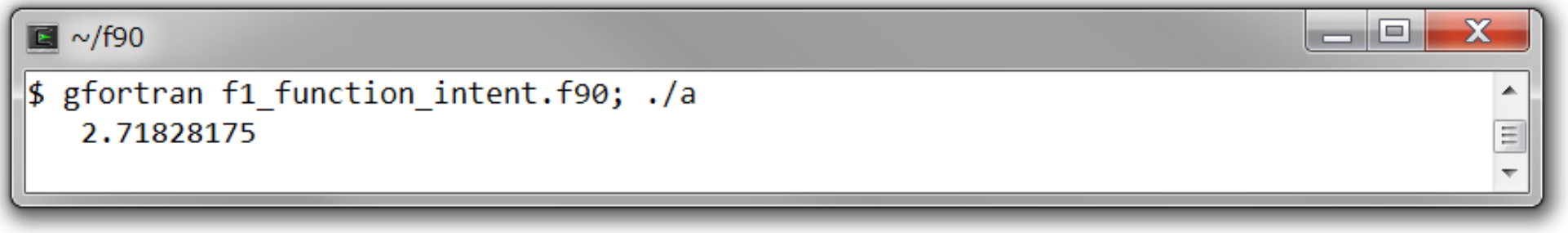
（互換性：Fortran 90〜）

- *TYPE*：仮引数の型.
- *intent-spec*：IN（変更できない），OUT（入力時は未定義），INOUT（変更・出力可）のいずれか.
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）.

■INTENT 属性を用いた外部関数副プログラム例 （f1_function_intent.f90）

```
program f1_function_intent ! 主プログラム
  implicit none
  real :: a = 2.0
  real, external :: func2b
  print *, func2b(0.5,a) ! 関数の呼び出し
  stop
end program f1_function_intent

function func2b(x,a) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: x,a
  real :: func2b
  func2b = exp(a*x)
end function func2b
```

コンパイル，リンク，実行すると，



13.4 INTERFACE 文

13.4.1 インターフェースブロック

外部関数副プログラムの引用仕様（interface）を，INTERFACE 文を用いたインターフェースブロック（interface block）によって行う.

```
INTERFACE
    TYPE-f FUNCTION func (d-arg-1 [, d-arg-2, ... ] )
        TYPE-a, INTENT( intent-spec ) :: d-arg-1 [,d-arg-2, ... ]
    END [FUNCTION [func] ]
END INTERFACE
```

（互換性：Fortran 90〜）

- *TYPE-f, TYPE-a*：関数の型，仮引数の型.
- *func*：関数名.
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）.
- *intent-spec*：IN, OUT, INOUT のいずれか.

一つのインターフェースブロック内に，複数の内部関数副プログラムを記述できる．また，IMPLICIT NONE は不要である.

また，RESULT 句を用いた場合，

```
INTERFACE
    FUNCTION func-r (d-arg-1 [, d-arg-2, ... ] ) RESULT (r-name)
        TYPE-f  r-name
        TYPE-a, INTENT( intent-spec ) :: d-arg-1 [,d-arg-2, ... ]
    END [FUNCTION [func-r] ]
END INTERFACE
```

（互換性：Fortran 90〜）

- *TYPE-f, TYPE-a*：関数の型，仮引数の型.
- *func-r*：関数名.
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）.
- *intent-spec*：IN, OUT, INOUT のいずれか.
- *r-name*：戻り値.

13.4.2　主プログラム内での引用仕様宣言

主プログラムの宣言文の次に，インターフェースブロックをおけば，外部関数副プログラムの引用仕様宣言ができる．例えば．主プログラムについてのみ示すと次のようになる．

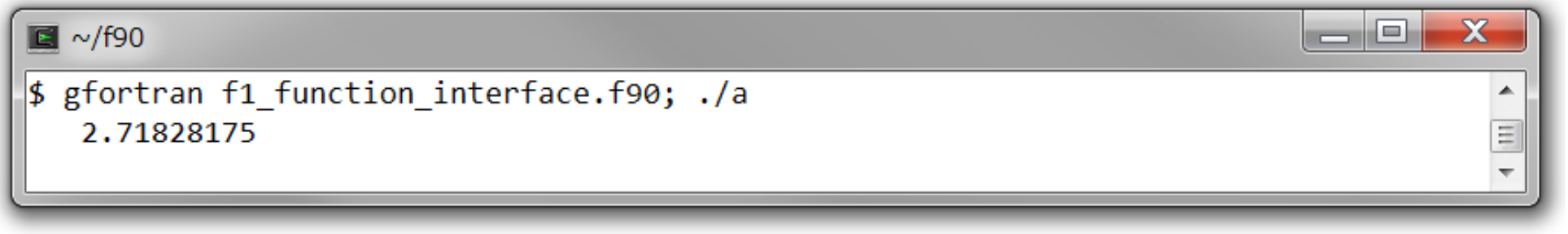
```
PROGRAM prog-name ! 主プログラム
    [specification-part]
    INTERFACE
        TYPE-f FUNCTION func (d-arg-1 [, d-arg-2, ... ] ) ! インターフェースブロック
            ...
        END [FUNCTION [func] ]
        FUNCTION func-r (d-arg-1 [, d-arg-2, ... ] ) RESULT (r-name) ! インターフェースブロック
            ...
        END [FUNCTION [func-r] ]
    END INTERFACE
    ...
    func (a-arg-1 [, a-arg-2, ... ] ) ! 関数の呼び出し
    func-r (b-arg-1 [, b-arg-2, ... ] ) ! 関数の呼び出し
    ...
END [PROGRAM [prog-name] ]
```

（互換性：Fortran 90～）

■インターフェースブロック（主プログラム）に関するプログラム例１　RESULT 句を用いない外部関数副プログラム（f1_function_interface.f90）.

```
program f1_function_interface ! 主プログラム
  implicit none

  interface ! インターフェースブロック
     real function func1(x,a)
       real, intent(IN) :: x,a
     end function func1
  end interface

  print *, func1(0.5,2.0) ! 関数の呼び出し
  stop
end program f1_function_interface

real function func1(x,a) ! 内部関数副プログラム
  implicit none
  real, intent(IN) :: x,a
  func1 = exp(a*x)
  return
end function func1
```

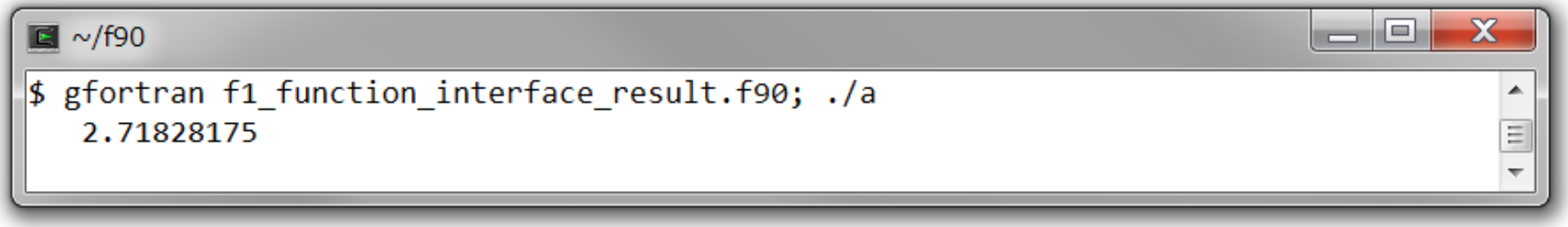
コンパイル，リンク，実行すると，



■インターフェースブロック（主プログラム）に関するプログラム例２　RESULT 句によってスカラを返す外部関数副プログラム（f1_function_interface_result.f90）.

```
program f1_function_interface_result ! 主プログラム
  implicit none
  real :: a = 2.0

  interface ! インターフェースブロック
     real function func3(x,a) result(y)
       real, intent(IN) :: x,a
     end function func3
  end interface

  print *, func3(0.5,a) ! 関数の呼び出し
  stop
end program f1_function_interface_result

real function func3(x,a) result(y) ! 外部関数副プログラム（戻り値がスカラ）
  implicit none
  real, intent(IN) :: x,a
  y = exp(a*x)
  return
end function func3
```

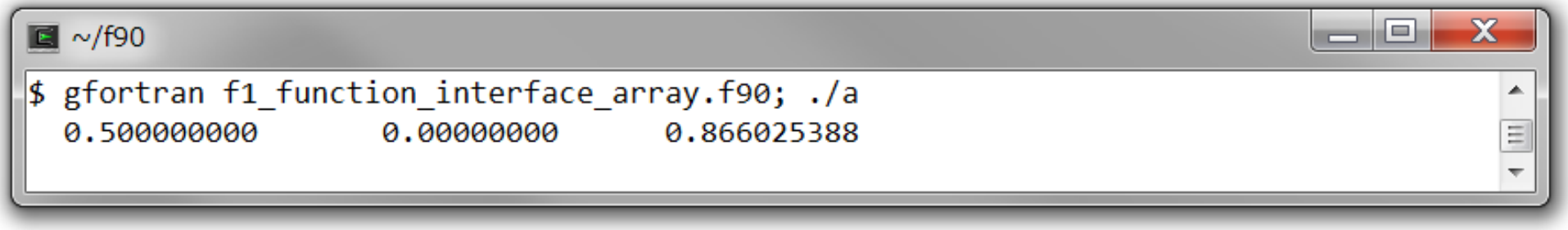
コンパイル，リンク，実行すると，



■インターフェースブロック（主プログラム）に関するプログラム例 3　RESULT 句によって配列を返す外部関数副プログラム（f1_function_interface_array.f90).

```
program f1_function_interface_array ! 主プログラム
  implicit none
  real :: q = 45.0/atan(1.0)
  interface ! インターフェースブロック
     function func4(th,phi) result(uvec)
       real, intent(IN) :: th,phi
       real :: uvec(3)
     end function func4
  end interface
  print *, func4(30.0/q, 0.0/q) ! degrees/q
  stop
end program f1_function_interface_array
function func4(th,phi) result(uvec) ! 外部関数副プログラム（戻り値が配列）
  implicit none
  real, intent(IN) :: th,phi
  real :: uvec(3),st,ct,sp,cp
  st = sin(th);  ct = cos(th)
  sp = sin(phi); cp = cos(phi)
  uvec(1) = st*cp; uvec(2) = st*sp; uvec(3) = ct
  return
end function func4
```

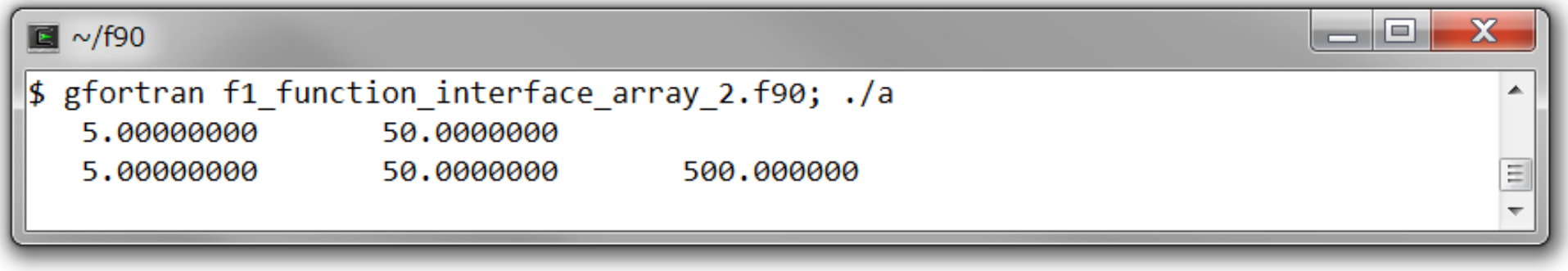
コンパイル，リンク，実行すると，



■インターフェースブロック（主プログラム）に関するプログラム例４　仮引数および戻り値がとも配列の外部関数副プログラム（f1_function_interface_array_2.f90）.

```
program f1_function_interface_array_2 ! 主プログラム
  implicit none
  interface ! インターフェースブロック
     function funca(x,y) result(z)
       real, intent(IN) :: x(:),y(:)
       real :: z(size(x))
     end function funca
  end interface
  print *, funca( (/3.0, 30.0/), [4.0, 40.0] )
  print *, funca( (/3.0, 30.0, 300.0/), [4.0, 40.0, 400.0] )
  stop
end program f1_function_interface_array_2

function funca(x,y) result(z) ! 外部関数副プログラム（仮引数，戻り値が配列）
  implicit none
  real, intent(IN) :: x(:),y(:)
  real :: z(size(x))
  z(:) = sqrt(x(:)**2 + y(:)**2)
  return
end function funca
```

コンパイル，リンク，実行すると，



13.5　引数の機能拡張

13.5.1　引数キーワード（argument keyword）

引用仕様（宣言）があれば，実引数を

dummy-argument-i = actual-argument-i　(i=1,2, ...)

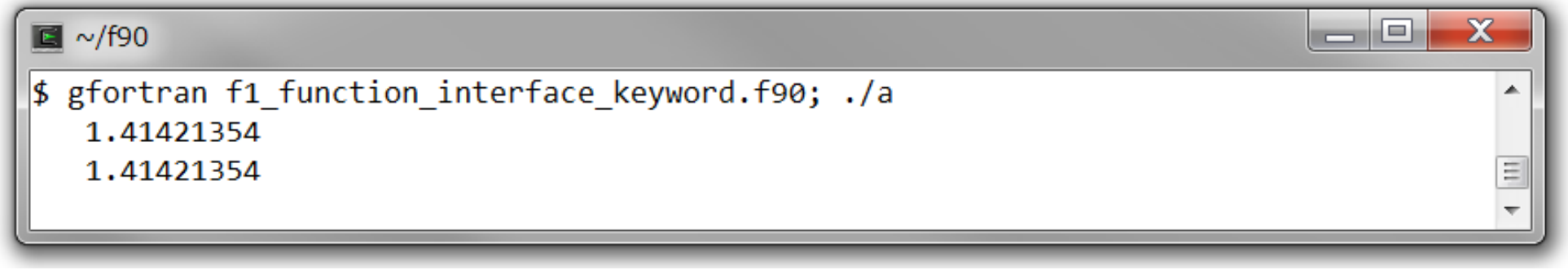
と指定することができ，引数の順番を変えて並べてもよい（引用仕様のない従来の外部関数副プログラムでは使用できない）.

■引数キーワードに関する外部関数副プログラム例 1　実引数の順番を変えない場合と変えた場合（f1_function_interface_keyword.f90）.

```
program f1_function_interface_keyword ! 主プログラム
  implicit none
  interface ! インターフェースブロック
     real function kfunc(x,y,a)
       real, intent(IN) :: x,y,a
     end function kfunc
  end interface
  print *, kfunc(x=1.0,y=1.0,a=0.5) ! 引数キーワード（順番を変えない場合）
  print *, kfunc(a=0.5,x=1.0,y=1.0) ! 引数キーワード（順番を変えた場合）
  stop
```

```
end program f1_function_interface_keyword

real function kfunc(x,y,a) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: x,y,a
  kfunc = (x**2+y**2)**a
  return
end function kfunc
```

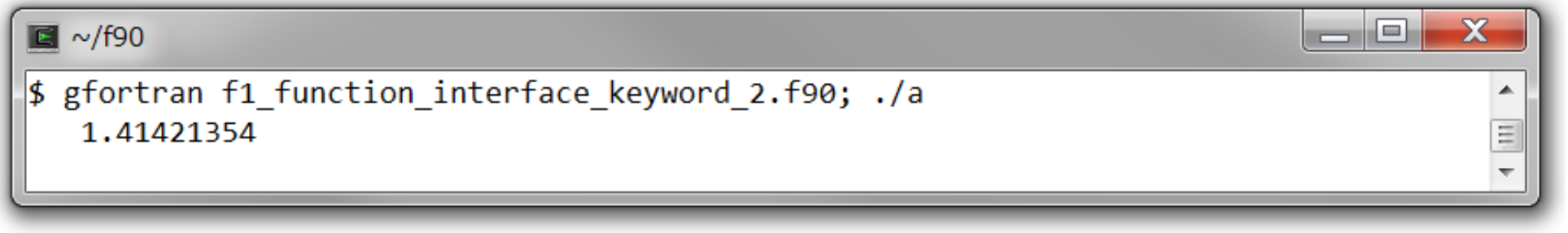
コンパイル，リンク，実行すると，



■引数キーワードに関する外部関数副プログラム例 2　あるいは，先行する実引数 *actual-argument-i　(i=1, 2, ...　,m)* は並び順として，そのあとの実引数で引数キーワードを *dummy-argument-i = actual-argument-i　(i=m+1, m+2, ...)* と指定し，引数の順番を変えて並べてもよい（f1_function_interface_keyword_2.f90）.

```
program f1_function_interface_keyword_2 ! 主プログラム
  implicit none
  interface ! インターフェースブロック
     real function kfunc(x,y,a)
       real, intent(IN) :: x,y,a
     end function kfunc
  end interface
  print *, kfunc(1.0,a=0.5,y=1.0) ! 引数キーワード
  stop
end program f1_function_interface_keyword_2

real function kfunc(x,y,a) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: x,y,a
  kfunc = (x**2+y**2)**a
  return
end function kfunc
```

コンパイル，リンク，実行すると，



13.5.2 OPTIONAL 属性

引用仕様（宣言）があれば，仮引数に OPTIONAL 属性をつけて，省略可能な仮引数を宣言できる．内部副プログラムおよび後述するモジュール副プログラムで使用できる（引用仕様のない従来の外部関数副プログラムでは使用できない）．

■Syntax

TYPE, OPTIONAL :: *d-arg-1 [,d-arg-2, ...]*

（互換性：Fortran 90〜）

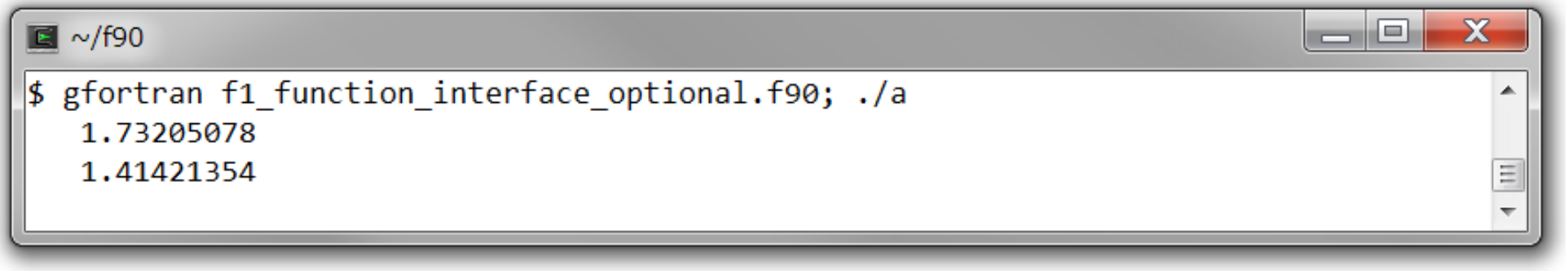
- *TYPE*：仮引数の型．
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）．

省略可能な仮引数に対応して，実引数があるかどうかを，PRESENT 組込関数（実引数がある場合は真，ない場合は偽）により検査し，その結果を基にして実行文を作成する．

■OPTIONAL 属性に関する外部関数副プログラム例　実引数の一部を省略した場合 (f1_function_interface_optional.f90)

```
program f1_function_interface_optinal ! 主プログラム
  implicit none
  interface ! インターフェースブロック
     real function ofunc(x,y,z)
       real, intent(IN) :: x,y
       real, intent(IN), optional :: z ! optional 属性
     end function ofunc
  end interface
  print *, ofunc(1.0,1.0,1.0) ! 実引数を省略していない場合
  print *, ofunc(1.0,1.0) ! 実引数を省略した場合
end program f1_function_interface_optinal
real function ofunc(x,y,z) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: x,y
  real, intent(IN), optional :: z ! optional 属性
  if(present(z)) then ! present 組込関数
     ofunc = sqrt(x**2+y**2+z**2)
  else
     ofunc = sqrt(x**2+y**2)
  endif
end function ofunc
```

コンパイル，リンク，実行すると，



13.6 FUNCTION PREFIX（接頭辞）

13.6.1 RECURSIVE FUNCTION（再帰関数）

自分自身を直接あるいは間接に呼び出すことができる（再帰コール）関数で，RESULT 句が必須（引用仕様のない従来の外部関数副プログラムでは使用できない）.

■Syntax

```
RECURSIVE FUNCTION rfunc (d-arg-1 [, d-arg-2, ... ] ) RESULT (r-name)
     TYPE-f  r-name
     TYPE-a ::  d-arg-1 [, d-arg-2, ... ]
    ...
    ...  =  ...  rfunc ( ...  ... )  ...
RETURN
END [FUNCTION [rfunc] ]
```

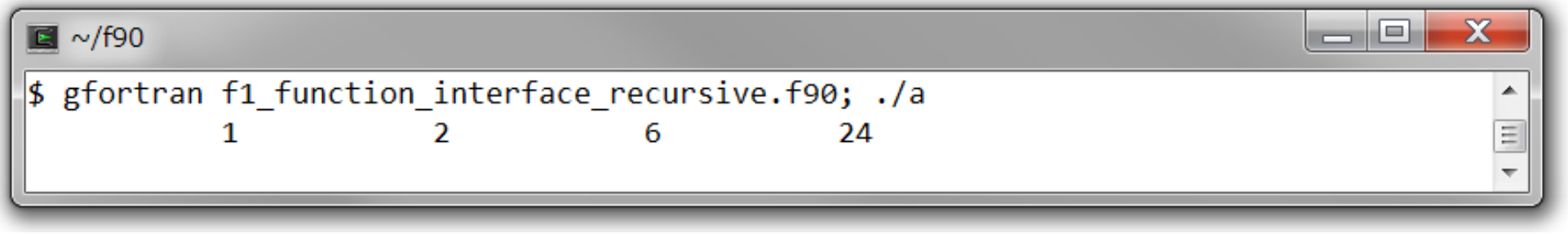
（互換性：Fortran 90〜）

- *TYPE-f, TYPE-a*：関数の型，仮引数の型.
- *rfunc*：関数名.
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）.
- *r-name*：戻り値.

■RECURSIVE FUNCTION に関するプログラム例 1　再帰コールによる階乗計算 (f1_function_interface_recursive.f90)

```
program f1_function_interface_recursive ! 主プログラム
  implicit none
  integer :: i
  interface ! インターフェースブロック
     recursive integer function rfunc(n) result(k) ! recursive 接頭辞
       integer, intent(IN) :: n
     end function rfunc
  end interface
  Print *, (rfunc(i),i=1,4)
  stop
end program f1_function_interface_recursive
recursive integer function rfunc(n) result(k) ! recursive 接頭辞
  implicit none
  integer, intent(IN) :: n
  if(n==1) then
     k = 1
  else
     k = n*rfunc(n-1)
  endif
  return
end function rfunc
```

コンパイル，リンク，実行すると，

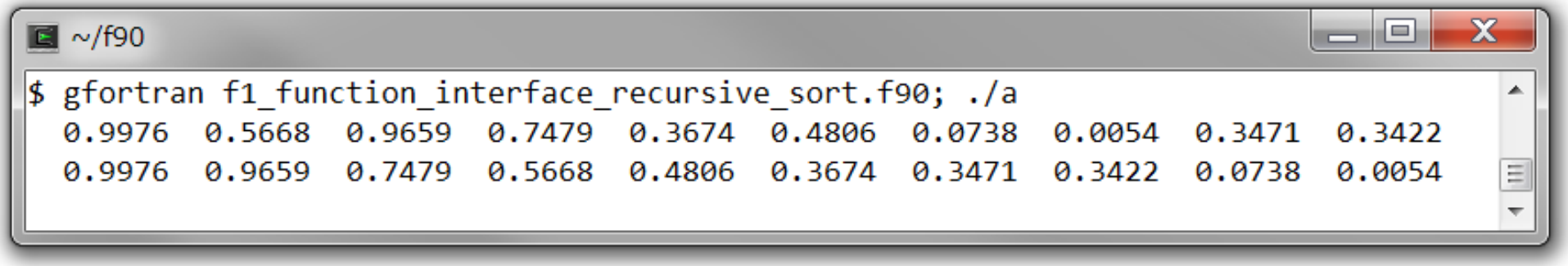


■RECURSIVE FUNCTION に関するプログラム例 2　再帰コールによるソート (f1_function_interface_recursive_sort.f90)

```
program f1_function_interface_recursive_sort
  implicit none
  integer, parameter :: nn = 10
  real, dimension(nn) :: data

  interface ! インターフェースブロック
     recursive function qsort(a) result(b)
       real, dimension(:), intent(IN) :: a
       real, dimension(1:size(a)) :: b
     end function qsort
  end interface

  call random_number(data)
  write(*,'(10f8.4)') data
  write(*,'(10f8.4)') qsort(data)
  stop
end program f1_function_interface_recursive_sort

recursive function qsort(a) result(b)
  real, dimension(:), intent(IN) :: a
  real, dimension(1:size(a)) :: b
```

```
  if(size(a)>1) then
     b = [ qsort(pack(a(2:),a(2:)>a(1))), a(1), &
           qsort(pack(a(2:),a(2:)<=a(1))) ]
  else
     b = a
  endif
  return
end function qsort
```

コンパイル，リンク，実行すると，



13.6.2 PURE FUNCTION（純粋関数）

仮引数のすべてが INTENT(IN) 属性で，入出力を行わない関数．組込関数に対応するユーザ定義関数である．

■Syntax

```
PURE  TYPE-f  pfunc (d-arg-1 [, d-arg-2, ... ] )  RESULT (r-name)
      TYPE-a,  INTENT(IN)  ::  d-arg-1 [, d-arg-2, ... ]
     ...
      r-name = ...
     ...
RETURN
END [FUNCTION [pfunc] ]
```

（互換性：Fortran 95〜）

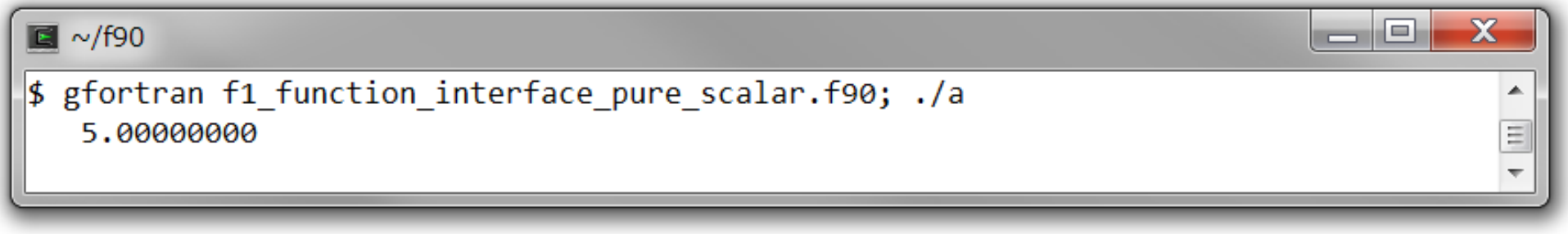
- *TYPE-f, TYPE-a*：関数の型，仮引数の型．
- *pfunc*：関数名．
- *d-arg-i (i=1,2, ...)*：INTENT(IN) 属性をもたせた仮引数（dummy argument）．
- *r-name*：戻り値．

ただし，局所変数の SAVE 属性宣言や初期化は行えない（複数の呼び出しが互いに独立）．

■PURE FUNCTION に関するプログラム例　戻り値がスカラ式の外部関数副プログラム (f1_function_interface_pure_scalar.f90)

```
program f1_function_interface_pure_scalar ! 主プログラム
  implicit none

  interface ! インターフェースブロック
     pure real function pfunc(x,y) result(z) ! pure 接頭辞
       real, intent(IN) :: x,y
     end function pfunc
  end interface

  print *, pfunc(3.0, 4.0)
  stop
end program f1_function_interface_pure_scalar

pure real function pfunc(x,y) result(z) ! pure 接頭辞
  real, intent(IN) :: x,y
  z = x**2 + y**2
  z = sqrt(z)
  return
end function pfunc
```

コンパイル，リンク，実行すると，



13.6.3 ELEMENTAL FUNCTION（要素別関数）

実引数がすべてスカラの場合，戻り値はスカラとなり，実引数が配列の場合，戻り値は配列となる（配列組込関数と同様）．ただし，PURE FUNCTION でなければならない．

■Syntax

```
ELEMENTAL FUNCTION efunc (d-arg-1 [, d-arg-2, ... ] ) RESULT (r-name)
     TYPE-f  r-name
     TYPE-a,  INTENT(IN)  ::  d-arg-1 [, d-arg-2, ... ]
    ...
     r-name = ...
    ...
RETURN
END [FUNCTION [efunc] ]
```

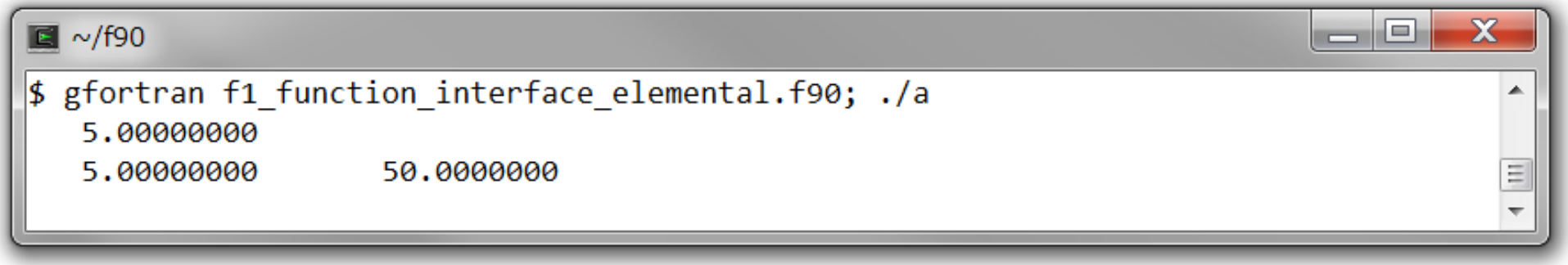
（互換性：Fortran 95〜）

- *TYPE-f, TYPE-a*：関数の型，仮引数の型．
- *efunc*：関数名．
- *d-arg-i (i=1,2, ...)*：INTENT 属性をもたせた仮引数（dummy argument）．
- *r-name*：戻り値．

■ELEMENTAL FUNCTION に関するプログラム例 1　実引数および戻り値が，スカラと 1 次元配列の場合（f1_function_interface_elemental.f90）

```
program f1_function_interface_elemental ! 主プログラム
  implicit none
  interface ! インターフェースブロック
     elemental real function efunc(x,y) result(z) ! elemental 接頭辞
       real, intent(IN) :: x,y
     end function efunc
  end interface
  print *, efunc(3.0, 4.0)
  print *, efunc( (/3.0, 30.0/), [4.0, 40.0] )
  stop
end program f1_function_interface_elemental
elemental real function efunc(x,y) result(z) ! elemental 接頭辞
  implicit none
  real, intent(IN) :: x,y
  z = x**2 + y**2
  z = sqrt(z)
  return
end function efunc
```

コンパイル，リンク，実行すると，

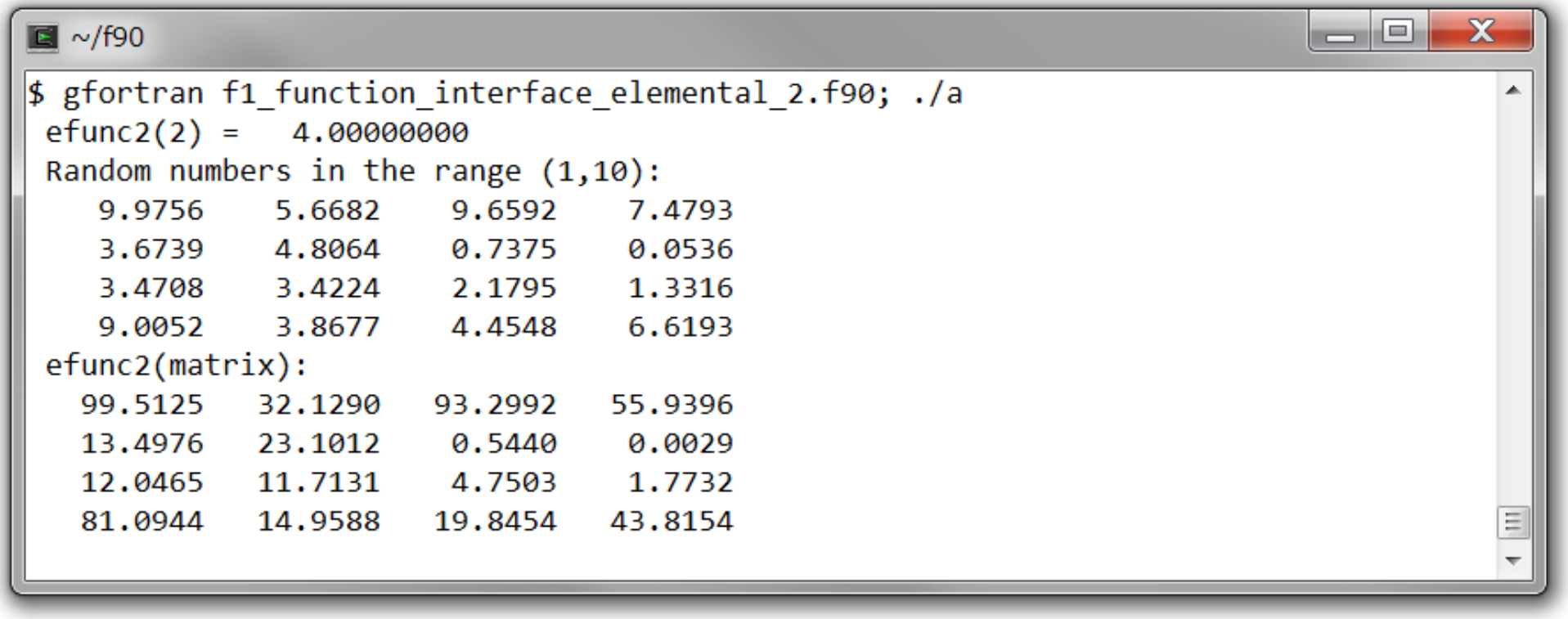

~/f90
$ gfortran f1_function_interface_elemental.f90; ./a
5.00000000
5.00000000 50.0000000

■ELEMENTAL FUNCTION に関するプログラム例 2　実引数および戻り値が，スカラと 2 次元配列の場合（f1_function_interface_elemental_2.f90）

```
program f1_function_interface_elemental_2
  implicit none
  real :: scalar = 2.0
  real, dimension(4,4) :: matrix

  interface ! インターフェースブロック
     real elemental function efunc2(x)
       real, intent(IN) :: x
     end function efunc2
  end interface

  call random_number(matrix)
  matrix = matrix * 10.0
  write(*,*) 'efunc2(2) =',efunc2(scalar)
  write(*,*) 'Random numbers in the range (1,10):'
  write(*,'(4f10.4)') matrix
  write(*,*) 'efunc2(matrix):'
  write(*,'(4f10.4)') efunc2(matrix)
  stop
end program f1_function_interface_elemental_2
```

```
real elemental function efunc2(x)
  real, intent(IN) :: x
  efunc2 = x*x
  return
end function efunc2
```

コンパイル，リンク，実行すると，



14　外部サブルーチン副プログラム

　サブルーチン副プログラムは引数を通じて値のやりとりを行うもので，主プログラムとは別の独立したプログラム単位（外部手続き（external procedure））とするサブルーチンを外部サブルーチン副プログラムと呼ぶ.

14.1　SUBROUTINE 文

14.1.1　属性を用いない場合

■Syntax

```
SUBROUTINE sub-name (d-arg-1 [, d-arg-2, ... ] )
     TYPE-a  d-arg-1 [, d-arg-2, ... ]
    ...
RETURN
END [SUBROUTINE [sub-name] ]
```

（互換性：FORTRAN 77～）

- *sub-name*：サブルーチン名.
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）.
- *TYPE-a*：仮引数の型.

14.1.2 CALL 文

サブルーチン副プログラムの実行（呼び出し）は，次のようになる．

```
call sub-name(a-arg-1 [, a-arg-2, ... ])
```

ただし，a-arg-1，a-arg-2 は実引数（actual argument）．したがって，外部サブルーチン副プログラムを主プログラムから呼び出す場合は，次のようになる．

```
PROGRAM prog-name ! 主プログラム
    ...
    CALL sub-name (a-arg-1 [, a-arg-2, ... ] ) ! サブルーチンの呼び出し
    ...
END [PROGRAM [prog-name] ]

SUBROUTINE sub-name (d-arg-1 [, d-arg-2, ... ] ) ! 外部（サブルーチン）副プログラム
    ...
END [SUBROUTINE [sub-name] ]
```

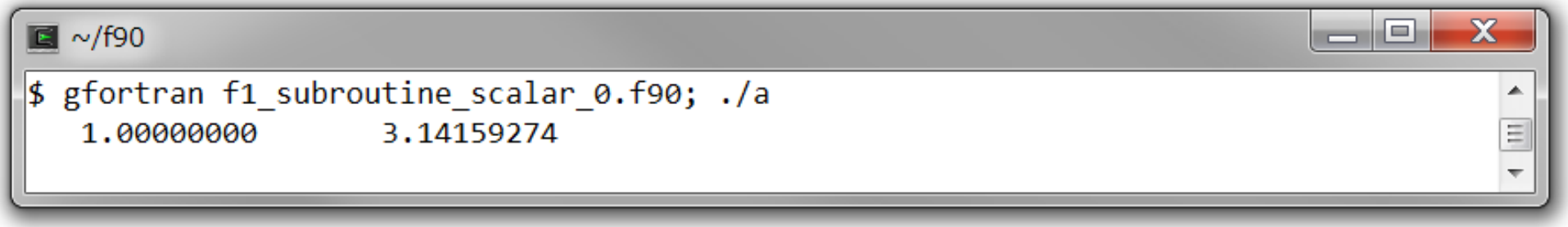
（互換性：FORTRAN 77〜）

定義した変数は，プログラム単位で各々独立である．

■属性を用いない外部サブルーチン副プログラム例 （f1_subroutine_scalar_0.f90）

```
program f1_subroutine_scalar_0 ! 主プログラム
  implicit none
  real r,th
  call sub(-1.0, 0.0 ,r,th) ! サブルーチンの呼び出し
  print *,r,th
  stop
end program f1_subroutine_scalar_0

subroutine sub(x,y, r,th) ! 外部サブルーチン副プログラム
  implicit none
  real x,y,r,th
  r = sqrt(x**2 + y**2)
  th = atan2(y,x)
  return
end subroutine sub
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_subroutine_scalar_0.f90; ./a
1.00000000 3.14159274

14.1.3　COMMON 文

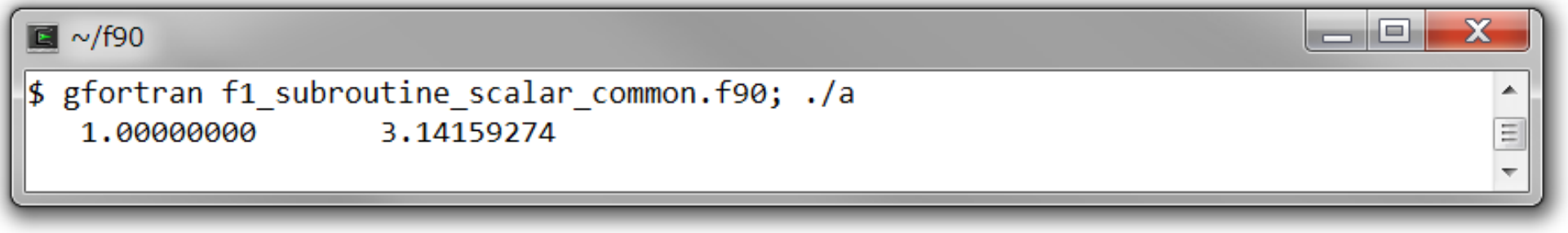
　サブルーチン副プログラムにおいても，引数とは別に，COMMON 文によりプログラム単位間の変数などの記憶場所を共有することができる.

■COMMON 文を用いたプログラム例 1　　スカラ変数を COMMON 文により共有する場合 (f1_subroutine_scalar_common.f90).

```
program f1_subroutine_scalar_common ! 主プログラム
  implicit none
  real r,th
  common /rth/ r,th ! コモン文
  call sub(-1.0, 0.0) ! サブルーチンの呼び出し
  print *,r,th
  stop
end program f1_subroutine_scalar_common

subroutine sub(x,y) ! 外部サブルーチン副プログラム
  implicit none
  real x,y,r,th
  common /rth/ r,th ! コモン文
  r = sqrt(x**2 + y**2)
```

```
  th = atan2(y,x)
  return
end subroutine sub
```

コンパイル，リンク，実行すると，

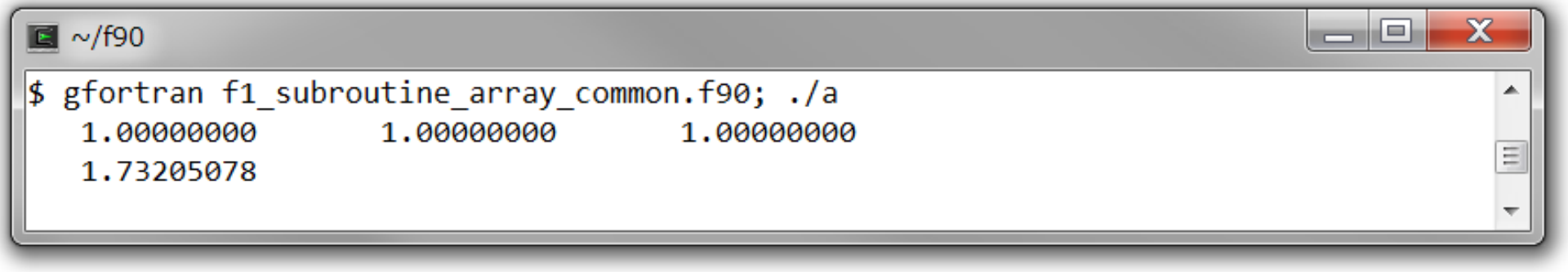


■COMMON 文を用いたプログラム例 2　配列を COMMON 文により共有する場合 (f1_subroutine_array_common.f90)

```
program f1_subroutine_array_common ! 主プログラム
  implicit none
  integer, parameter :: nn = 11
  integer :: n = 3
  real :: x(nn),a
  common /cb/ x

  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
  call suba(n, a)
  print *,a
  stop
end program f1_subroutine_array_common

subroutine suba(n, a) ! 外部サブルーチン副プログラム
  implicit none
  integer, parameter :: nn = 11
  integer, intent(IN) :: n
  real, intent(OUT) :: a
  real :: x(nn)
  integer :: i
```

```
  common /cb/ x
  a = 0.0
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
end subroutine suba
```

コンパイル，リンク，実行すると，

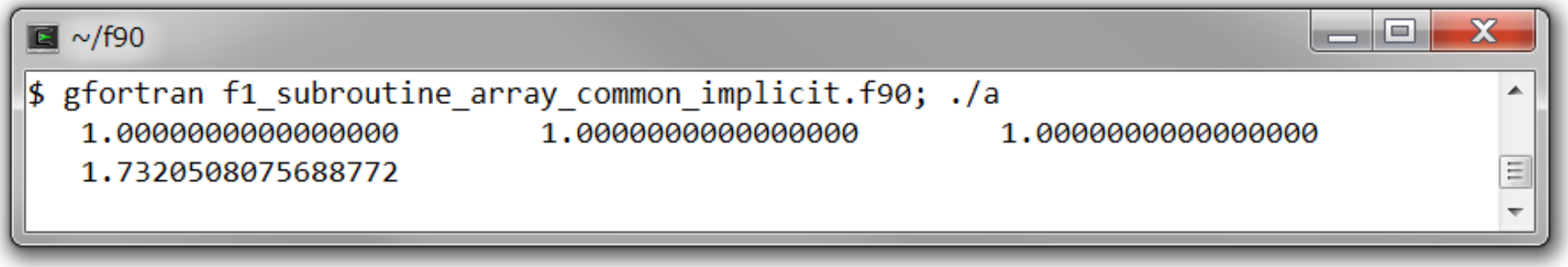


■COMMON 文を用いたプログラム例 3　暗黙の型宣言および IMPLICIT 文を用い，COMMON 文により共有する場合(f1_subroutine_array_common_implicit.f90)

```
program f1_subroutine_array_common_implicit ! 主プログラム
  implicit real*8(a-h,o-z)
  parameter(nn = 11)
  common /cb/ x(nn)
  x(1:3) = [1.0, 1.0, 1.0]
  print *,x(1:3)
  call suba(3, a)
  print *,a
  stop
end program f1_subroutine_array_common_implicit

subroutine suba(n, a) ! 外部サブルーチン副プログラム
  implicit real*8(a-h,o-z)
  parameter(nn = 11)
  common /cb/ x(nn)
  a = 0.0
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
```

```
end subroutine suba
```

コンパイル，リンク，実行すると，



14.1.4 INTENT 属性を用いる場合

次のように仮引数に INTENT 属性をつけることが望ましい.

```
SUBROUTINE sub-name (d-arg-1, d-arg-2, ... ... )
     TYPE-a, INTENT(IN) :: d-arg-1, ...
     TYPE-b, INTENT(OUT) :: d-arg-2, ...
    ...
RETURN
END [SUBROUTINE [sub-name] ]
```

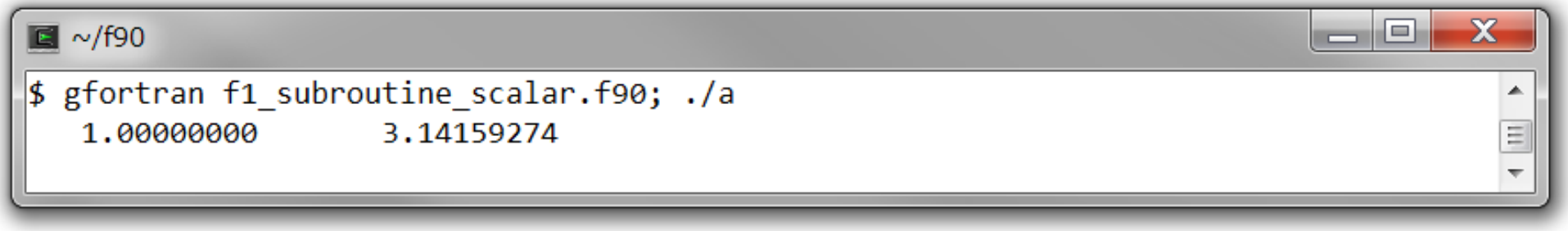
（互換性：Fortran 90〜）

引用仕様宣言を行わない場合，引数キーワード機能，OPTIONAL 属性，ELEMENTAL 接頭辞を使用できない.

■INTENT 属性をもたせた単精度プログラム例 （f1_subroutine_scalar.f90）

```
program f1_subroutine_scalar ! 主プログラム
  implicit none
  real :: r,th
  call sub(-1.0, 0.0 ,r,th) ! サブルーチンの呼び出し
  print *,r,th
  stop
end program f1_subroutine_scalar
subroutine sub(x,y, r,th) ! 外部サブルーチン副プログラム
  implicit none
  real, intent(IN) :: x,y ! intent 属性
  real, intent(OUT) :: r,th ! intent 属性
  r = sqrt(x**2 + y**2)
  th = atan2(y,x)
  return
end subroutine sub
```

コンパイル，リンク，実行すると，

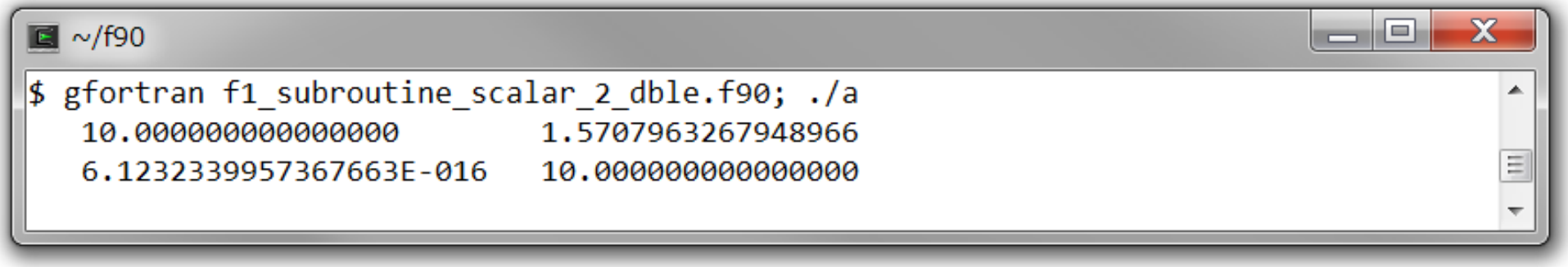


■INTENT 属性をもたせた倍精度プログラム例

(f1_subroutine_scalar_2_dble.f90)

```
program f1_subroutine_scalar_2_dble ! 主プログラム
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  real(8) :: x,y,r,phi
  r = 10.0D0
  phi = 90.0D0/q
  print *,r,phi
  call dsub2(r,phi, x,y) ! サブルーチンの呼び出し
  print *,x,y
  stop
end program f1_subroutine_scalar_2_dble

subroutine dsub2(r,phi, x,y) ! 外部サブルーチン副プログラム
  implicit none
  real(8), intent(IN) :: r,phi ! intent 属性
  real(8), intent(OUT) :: x,y ! intent 属性
  x = r*cos(phi)
  y = r*sin(phi)
  return
end subroutine dsub2
```

コンパイル，リンク，実行すると，



14.2　インターフェースブロック

外部サブルーチン副プログラムの引用仕様（interface）もまた，INTERFACE 文を用いたインターフェースブロック（interface block）によって行う（IMPLICIT NONE は不要）.

```
INTERFACE
    SUBROUTINE sub-name (d-arg-1 [, d-arg-2, ... ] )
        TYPE-a, INTENT( intent-spec ) :: d-arg-1 [, d-arg-2, ... ]
       .....
    END SUBROUTINE [sub-name] ]
END INTERFACE
```

（互換性：Fortran 90〜）

- *sub-name*：サブルーチン名.
- *TYPE-a*：仮引数の型.
- *d-arg-i (i=1,2, ...)*：仮引数（dummy argument）.
- *intent-spec*：IN, OUT, INOUT のいずれか.

インターフェースブロック内には，関数副プログラムに加えて複数のサブルーチン副プログラムを記述できる．これによって，引数キーワード指定ならびに OPTIONAL 属性等の

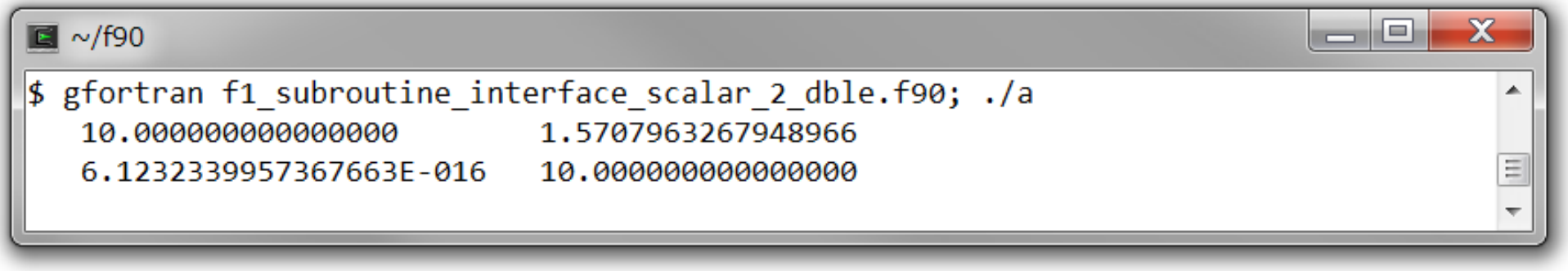
Fortran 90/95 の新機能を用いることができる.

■引数キーワードを指定したプログラム例　前節のプログラムにインタフェースブロックを用い，引数キーワードを使用できるようにすると，次のようになる (f1_subroutine_interface_scalar_2_dble.f90).

```
program f1_subroutine_interface_scalar_2_dble ! 主プログラム（倍精度）
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  real(8) :: x,y,r,phi

  interface ! インターフェースブロック
     subroutine dsub2(radius,angle_phi, rect_x,rect_y)
       real(8), intent(IN) :: radius,angle_phi
       real(8), intent(OUT) :: rect_x,rect_y
     end subroutine dsub2
  end interface

  r = 10.0D0
  phi = 90.0D0/q
  print *,r,phi
  call dsub2(rect_x=x, rect_y=y, radius=r, angle_phi=phi) ! 引数キーワード指定
  print *,x,y
  stop
end program f1_subroutine_interface_scalar_2_dble
```

```
subroutine dsub2(radius,angle_phi, rect_x,rect_y) ! 外部サブルーチン副プログラム
  implicit none
  real(8), intent(IN) :: radius,angle_phi
  real(8), intent(OUT) :: rect_x,rect_y
  rect_x = radius*cos(angle_phi)
  rect_y = radius*sin(angle_phi)
  return
end subroutine dsub2
```

コンパイル，リンク，実行すると，



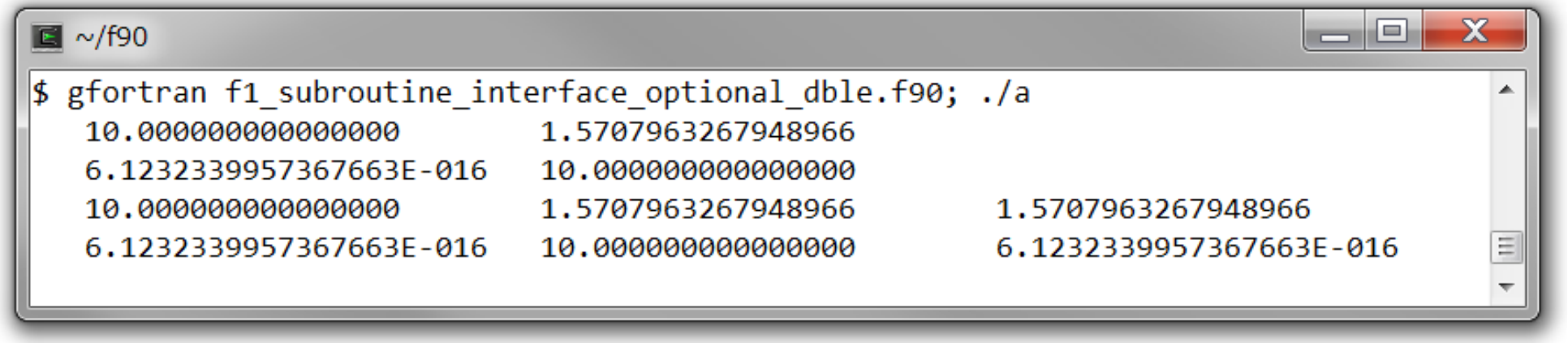
■引数キーワード指定ならびに OPTIONAL 属性をもたせたプログラム例（f1_subroutine_interface_optional_dble.f90）

```
program f1_subroutine_interface_optional_dble ! 主プログラム（倍精度）
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  real(8) :: x,y,r,phi
  real(8) :: z,th

  interface ! インターフェースブロック
     subroutine dsub2(radius,angle_theta,angle_phi, &
          rect_x,rect_y,rect_z)
       real(8), intent(IN) :: radius,angle_phi
       real(8), intent(OUT) :: rect_x,rect_y
       real(8), optional, intent(IN) :: angle_theta
       real(8), optional, intent(OUT) :: rect_z
     end subroutine dsub2
  end interface

  r = 10.0D0
  phi = 90.0D0/q
  print *,r,phi
  call dsub2(rect_x=x, rect_y=y, radius=r, angle_phi=phi) ! 引数を一部省略
  print *,x,y
```

```
  th = 90.0D0/q
  print *,r,th,phi
  call dsub2(rect_x=x, rect_y=y, rect_z=z, &
       radius=r, angle_phi=phi, angle_theta=th) ! 引数をすべて指定
  print *,x,y,z

  stop
end program f1_subroutine_interface_optional_dble

subroutine dsub2(radius,angle_theta,angle_phi, &
     rect_x,rect_y,rect_z) ! 外部サブルーチン副プログラム
  implicit none
  real(8), intent(IN) :: radius,angle_phi
  real(8), intent(OUT) :: rect_x,rect_y
  real(8), optional, intent(IN) :: angle_theta ! optional 属性
  real(8), optional, intent(OUT) :: rect_z ! optional 属性
  real(8) :: ct,st,cp,sp
  cp = cos(angle_phi)
  sp = sin(angle_phi)
  ct = 0.0D0
  st = 1.0D0
  if(present(angle_theta)) then ! present 関数
```

```
      ct = cos(angle_theta)
      st = sin(angle_theta)
   endif
   rect_x = radius*st*cp
   rect_y = radius*st*sp
   if(present(rect_z)) rect_z = radius*ct ! present 関数
   return
end subroutine dsub2
```

コンパイル，リンク，実行すると，



14.3　配列を引数とする方法

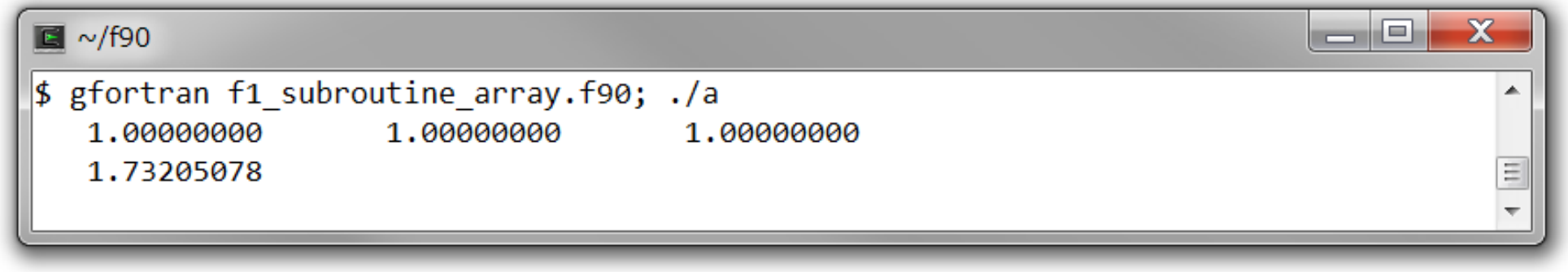
14.3.1　インターフェースブロックを用いない外部サブルーチン

■形状明示配列（explicit-shape array）を引数とする外部サブルーチン副プログラム例 1（f1_subroutine_array.f90）

```
program f1_subroutine_array ! 主プログラム
  implicit none
  integer, parameter :: nn = 11
  integer :: n = 3
  real :: x(nn),a
  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
  call suba(x,n,nn, a)
  print *,a
  stop
end program f1_subroutine_array

subroutine suba(x,n,nn, a) ! 外部サブルーチン副プログラム
  implicit none
  integer, intent(IN) :: n,nn
  real, intent(IN) :: x(nn) ! 形状明示配列（explicit-shape array）
```

```
  real, intent(OUT) :: a
  integer :: i
  a = 0.0
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
end subroutine suba
```

コンパイル，リンク，実行すると，



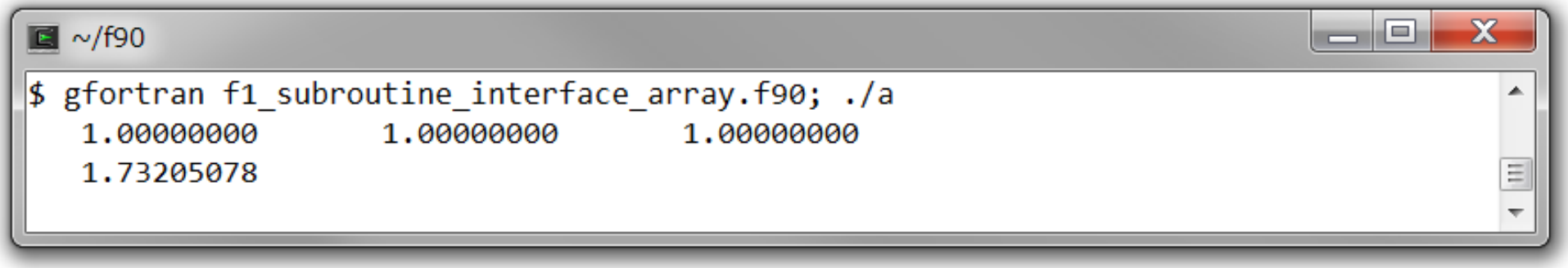
14.3.2　インターフェースブロックを用いた外部サブルーチン

■形状明示配列（explicit-shape array）を引数とする外部サブルーチン副プログラム例 2（f1_subroutine_interface_array.f90）

```
program f1_subroutine_interface_array ! 主プログラム
  implicit none
  integer, parameter :: nn = 11
  integer :: n = 3
  real :: x(nn),a

  interface ! インターフェースブロック
     subroutine suba(x,n,nn, a)
       integer, intent(IN) :: n,nn
       real, intent(IN) :: x(nn)
       real, intent(OUT) :: a
     end subroutine suba
  end interface

  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
  call suba(x,n,nn, a)
  print *,a
```

```
  stop
end program f1_subroutine_interface_array

subroutine suba(x,n,nn, a) ! 外部サブルーチン副プログラム
  implicit none
  integer, intent(IN) :: n,nn
  real, intent(IN) :: x(nn) ! 形状明示配列（explicit-shape array）
  real, intent(OUT) :: a
  integer :: i
  a = 0.0
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
end subroutine suba
```

コンパイル，リンク，実行すると，

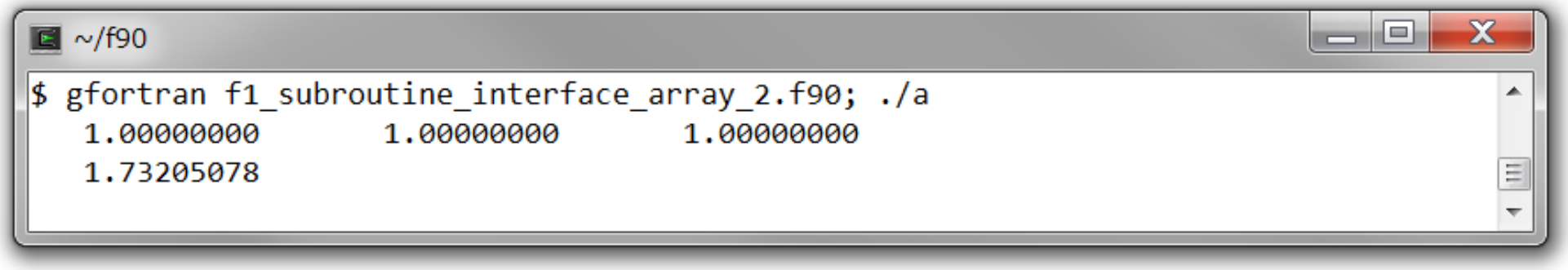

~/f90
$ gfortran f1_subroutine_interface_array.f90; ./a
1.00000000 1.00000000 1.00000000
1.73205078

■形状引継配列（assumed-shape array）を引数とする外部サブルーチン副プログラム例（f1_subroutine_interface_array_2.f90）

```
program f1_subroutine_interface_array_2 ! 主プログラム
  implicit none
  integer, parameter :: nn = 11
  integer :: n = 3
  real :: x(nn),a

  interface ! インターフェースブロック
     subroutine suba(x,n, a)
       integer, intent(IN) :: n
       real, intent(IN) :: x(:)
       real, intent(OUT) :: a
     end subroutine suba
  end interface

  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
  call suba(x,n, a)
  print *,a
  stop
end program f1_subroutine_interface_array_2
```

```
subroutine suba(x,n, a) ! 外部サブルーチン副プログラム
  implicit none
  integer, intent(IN) :: n
  real, intent(IN) :: x(:) ! 形状引継配列（assumed-shape array）
  real, intent(OUT) :: a
  integer :: i
  a = 0.0
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
end subroutine suba
```

コンパイル，リンク，実行すると，

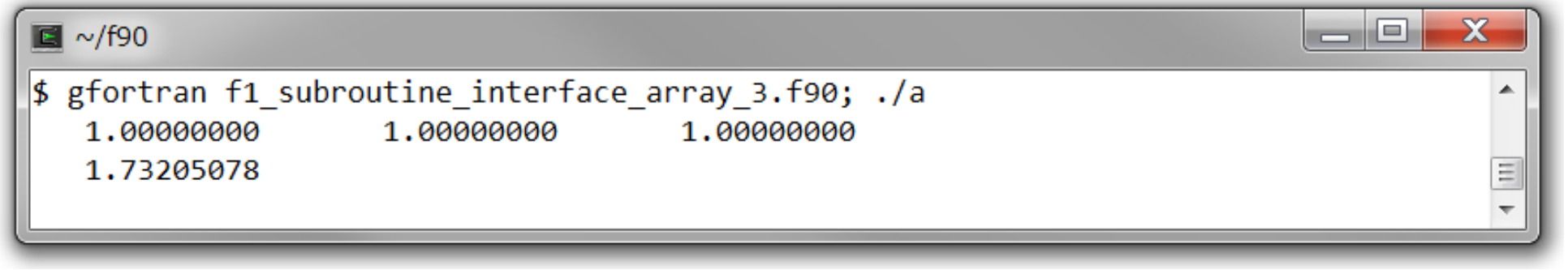


■動的割り付けした配列を引数とする外部サブルーチン副プログラム例 (f1_subroutine_interface_array_3.f90)

```
program f1_subroutine_interface_array_3 ! 主プログラム
  implicit none
  integer :: n = 3, is
  real, allocatable :: x(:) ! 形状無指定配列 (deferred-shape array)
  real :: a

  interface ! インターフェースブロック
     subroutine suba(x, a)
       real, intent(IN) :: x(:)
       real, intent(OUT) :: a
     end subroutine suba
  end interface

  allocate(x(n),stat=is) ! 配列の割り付け
  if(is/=0) stop 'cannot allocate'
  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
  call suba(x, a)
  print *,a
  deallocate(x) ! 割り付けの解除
  stop
```

```
end program f1_subroutine_interface_array_3

subroutine suba(x, a) ! 外部サブルーチン副プログラム
  implicit none
  real, intent(IN) :: x(:)
  real, intent(OUT) :: a
  integer :: i,n
  a = 0.0
  n = size(x) ! 配列の問い合わせ関数
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
end subroutine suba
```

コンパイル，リンク，実行すると，



14.4　関数を引数とする方法

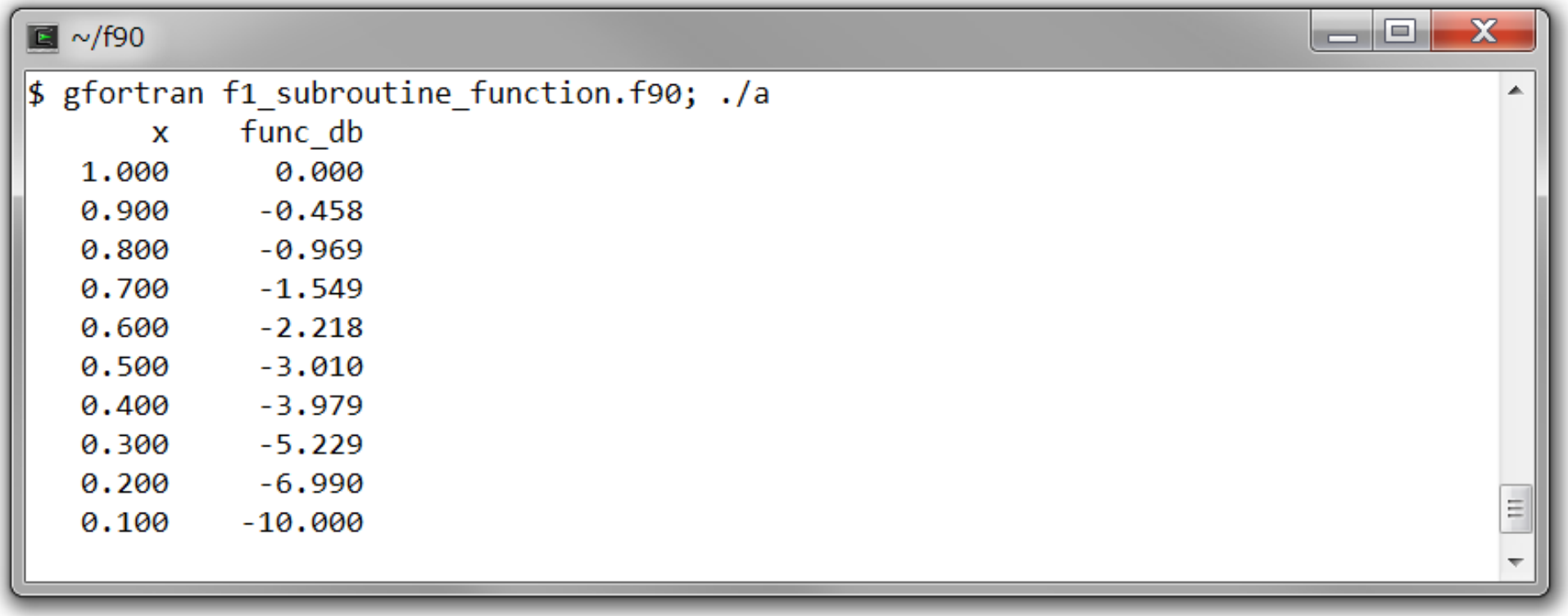
14.4.1　インターフェースブロックを用いない外部副プログラム

■スカラ関数副プログラムを引数とする倍精度の外部サブルーチン副プログラム例(f1_subroutine_function.f90)

```
program f1_subroutine_function ! 主プログラム
  implicit none
  real(8) :: x0 = 1.0D0, dx = -0.1D0
  real(8), external :: func_db
  integer :: nx = 10

  call sub_f(func_db, x0,dx,nx)

  stop
end program f1_subroutine_function

subroutine sub_f(func, x0,dx,nx) ! 外部サブルーチン副プログラム
  implicit none
  real(8), intent(IN) :: x0,dx
  integer, intent(IN) :: nx
  integer :: i
```

```
  real(8) :: func,x
  external func
  print '(a8,1x,a10)','x','func_db'
  do i=1,nx
     x = x0+dx*(i-1)
     print '(f8.3,1x,f10.3)',x,func(x)
  enddo
  return
end subroutine sub_f

real(8) function func_db(x) result(y) ! 外部関数副プログラム
  implicit none
  real(8), intent(IN) :: x
  y = abs(x)
  if(y/=0.0D0) then
     y = 10.0D0*log10(y)
  else
     y = -800.0D0
  endif
  return
end function func_db
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_subroutine_function.f90; ./a
x func_db
1.000 0.000
0.900 -0.458
0.800 -0.969
0.700 -1.549
0.600 -2.218
0.500 -3.010
0.400 -3.979
0.300 -5.229
0.200 -6.990
0.100 -10.000

14.4.2　インターフェースブロックを用いた外部副プログラム

■引数キーワードおよび OPTIONAL 属性をもたせた外部サブルーチン副プログラム例

前節のプログラムにインターフェースブロックを用いると次のようになる（f1_subroutine_function_interface.f90）.

```
program f1_subroutine_fucntion_interface ! 主プログラム
  implicit none
  real(8) :: x0 = 1.0D0, dx = -0.5D0
  integer :: n = 5

  interface ! インターフェースブロック
     subroutine sub_f(func, nx,xmin,delx)
       integer, intent(IN) :: nx
       real(8), optional :: xmin,delx
       real(8) :: func
       external func
     end subroutine sub_f
     real(8) function func_db(x) result(y)
       real(8), intent(IN) :: x
     end function func_db
  end interface
```

```
  call sub_f(func_db, n,x0,dx)
  call sub_f(func=func_db, nx=n,xmin=x0,delx=-0.05D0)
  call sub_f(func_db, n, xmin=1.0D0)
  call sub_f(func_db, 3)
  stop
end program f1_subroutine_fucntion_interface

subroutine sub_f(func, nx,xmin,delx) ! 外部サブルーチン副プログラム
  implicit none
  integer, intent(IN) :: nx
  real(8), optional :: xmin,delx ! optional 属性
  real(8) :: func
  integer :: i
  real(8) :: x,x0,dx
  external func
  if(present(xmin)) then ! present 関数
     x0 = xmin
  else
     x0 = 1.0D0
  endif
  if(present(delx)) then ! present 関数
     dx = delx
  else
```

```
      dx = -x0/(nx-1)
   endif
   print *,'x0=',x0,'dx=',dx
   print '(a8,1x,a10)','x','func_db'
   do i=1,nx
      x = x0+dx*(i-1)
      print '(f8.3,1x,f10.3)',x,func(x)
   enddo
   return
end subroutine sub_f

real(8) function func_db(x) result(y) ! 外部関数副プログラム
   implicit none
   real(8), intent(IN) :: x
   y = abs(x)
   if(y/=0.0D0) then
      y = 10.0D0*log10(y)
   else
      y = -800.0D0
   endif
   return
end function func_db
```

コンパイル，リンク，実行すると，

~/f90

```
$ gfortran f1_subroutine_function_interface.f90; ./a
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000      0.000
   0.500     -3.010
   0.000   -800.000
  -0.500     -3.010
  -1.000      0.000
 x0=   1.0000000000000000      dx=  -5.0000000000000003E-002
       x    func_db
   1.000      0.000
   0.950     -0.223
   0.900     -0.458
   0.850     -0.706
   0.800     -0.969
 x0=   1.0000000000000000      dx= -0.25000000000000000
       x    func_db
   1.000      0.000
   0.750     -1.249
   0.500     -3.010
   0.250     -6.021
   0.000   -800.000
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000      0.000
   0.500     -3.010
   0.000   -800.000
```

14.5 ELEMENTAL SUBROUTINE

■ELEMENTAL 接頭辞を用いた倍精度の外部サブルーチン副プログラム例
(f1_subroutine_interface_elemental_dble.f90).

```
program f1_subroutine_interface_elemental_dble ! 主プログラム（倍精度）
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  integer, parameter :: nn = 101
  real(8) :: x,y,r,phi
  real(8) :: z,th
  real(8) :: xx(nn),yy(nn),zz(nn),rr(nn),tth(nn),pphi(nn)
  real(8) :: phi0 = 0.0D0, dphi = 30.0D0/q
  integer :: i, nphi = 13

  interface ! インターフェースブロック
     elemental subroutine esub2(radius,angle_theta,angle_phi,&
          rect_x,rect_y,rect_z)
       real(8), intent(IN) :: radius,angle_phi
       real(8), intent(OUT) :: rect_x,rect_y
       real(8), optional, intent(IN) :: angle_theta
       real(8), optional, intent(OUT) :: rect_z
     end subroutine esub2
```

```
end interface

r = 10.0D0
phi = 90.0D0/q
print *,r,phi*q
call esub2(rect_x=x, rect_y=y, radius=r, angle_phi=phi) ! 引数はスカラー
print *,x,y

th = 90.0D0/q
print *,r,th*q,phi*q
call esub2(rect_x=x, rect_y=y, rect_z=z, &
     radius=r, angle_phi=phi, angle_theta=th) ! 引数はスカラー
print *,x,y,z

rr(1:nphi) = 10.0D0
pphi(1:nphi) = [ (phi0+dphi*(i-1),i=1,nphi) ]
call esub2(rect_x=xx, rect_y=yy, radius=rr, angle_phi=pphi) ! 引数は配列
print '(4a11)','rr','pphi[deg]','xx','yy'
do i=1,nphi
   print '(4(1x,f10.5))',rr(i),pphi(i)*q,xx(i),yy(i)
enddo

stop
```

```
end program f1_subroutine_interface_elemental_dble

elemental subroutine esub2(radius,angle_theta,angle_phi, &
     rect_x,rect_y,rect_z) ! ELEMENTAL SUBROUTINE
  implicit none
  real(8), intent(IN) :: radius,angle_phi
  real(8), intent(OUT) :: rect_x,rect_y
  real(8), optional, intent(IN) :: angle_theta ! optional 属性
  real(8), optional, intent(OUT) :: rect_z ! optional 属性
  real(8) :: ct,st,cp,sp
  cp = cos(angle_phi)
  sp = sin(angle_phi)
  ct = 0.0D0
  st = 1.0D0
  if(present(angle_theta)) then ! present 関数
     ct = cos(angle_theta)
     st = sin(angle_theta)
  endif
  rect_x = radius*st*cp
  rect_y = radius*st*sp
  if(present(rect_z)) rect_z = radius*ct ! present 関数
  return
end subroutine esub2
```

コンパイル，リンク，実行すると，

~/f90

```
$ gfortran f1_subroutine_interface_elemental_dble.f90; ./a
   10.000000000000000         90.000000000000000
   6.1232339957367663E-016    10.000000000000000
   10.000000000000000         90.000000000000000         90.000000000000000
   6.1232339957367663E-016    10.000000000000000         6.1232339957367663E-016
         rr  pphi[deg]          xx          yy
   10.00000    0.00000    10.00000    0.00000
   10.00000   30.00000     8.66025    5.00000
   10.00000   60.00000     5.00000    8.66025
   10.00000   90.00000     0.00000   10.00000
   10.00000  120.00000    -5.00000    8.66025
   10.00000  150.00000    -8.66025    5.00000
   10.00000  180.00000   -10.00000    0.00000
   10.00000  210.00000    -8.66025   -5.00000
   10.00000  240.00000    -5.00000   -8.66025
   10.00000  270.00000    -0.00000  -10.00000
   10.00000  300.00000     5.00000   -8.66025
   10.00000  330.00000     8.66025   -5.00000
   10.00000  360.00000    10.00000   -0.00000
```

15　内部副プログラム

主プログラムの内部，または外部副プログラムの内部の副プログラムを，内部副プログラム（internal subprogram）という．

15.1　CONTAINS 文

主プログラムに CONTAINS 文を一つだけおけ，その中の内部副プログラムは明示的引用仕様となる（インターフェースブロックとは別の方法）．

```
PROGRAM prog-name ! 主プログラム
    ...
    ... ! 内部副プログラムの呼び出し
    ...
     CONTAINS
        ... ! 内部副プログラム
        ...
END [PROGRAM [prog-name] ]
```

（互換性：Fortran 90〜）

主プログラムの変数は内部副プログラムで共有される（COMMON 文は不要）．そのため，主

プログラムで IMPLICIT NONE があれば，内部副プログラム内では不要である．また，内部副プログラム内で宣言した局所変数は独立である．

15.2　内部関数副プログラム

副プログラムが関数の場合，次のように内部関数副プログラムとなる．

```
PROGRAM prog-name ! 主プログラム
    ...
    func (a-arg-1 [, a-arg-2, ... ] ) ! 関数の呼び出し
    ...
    CONTAINS
    TYPE-f FUNCTION func (d-arg-1 [, d-arg-2, ... ] ) ! 内部関数副プログラム
        ...
    END [FUNCTION [func] ]
END [PROGRAM [prog-name] ]
```

（互換性：Fortran 90〜）

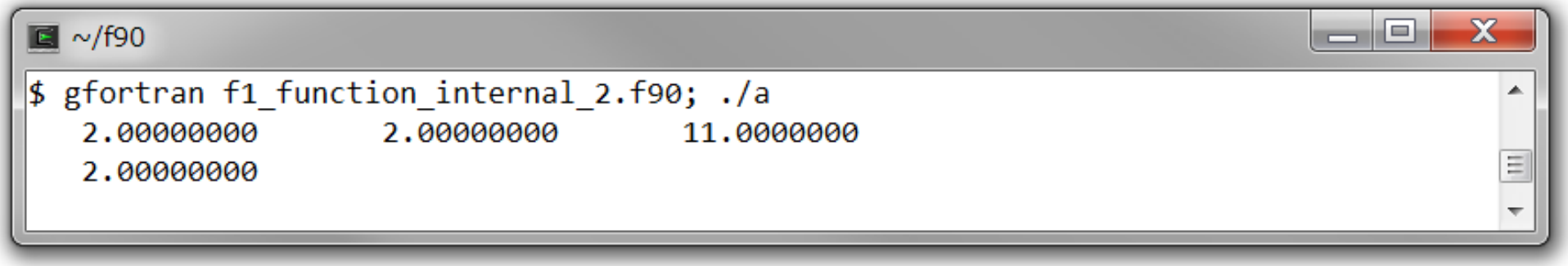
関数の呼び出し側で，関数の型宣言は不要である．

■内部副プログラムと外部副プログラムとの比較 （f1_function_internal_2.f90）

```
program f1_function_internal_2 ! 主プログラム
  implicit none
  real :: a=10.0
  real :: funce
  print *, funcb(1.0),funca(1.0),func()
  print *, funce(1.0)
  stop

contains
  real function funcb(b) ! 内部関数副プログラム
    real, intent(IN) :: b
    funcb = b+1.0
  end function funcb

  real function funca(a) ! 内部関数副プログラム
    real, intent(IN) :: a
    funca = a+1.0
  end function funca

  real function func() ! 内部関数副プログラム
    func = a+1.0
  end function func
```

```
end program f1_function_internal_2

real function funce(a) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: a
  funce = a+1.0
  return
end function funce
```

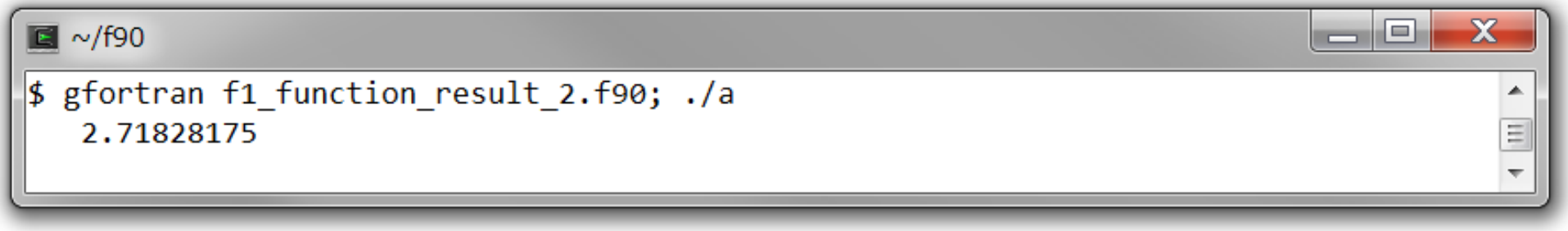
コンパイル，リンク，実行すると，



■戻り値がスカラ式の内部関数副プログラム例　(f1_function_result_2.f90)

```
program f1_function_result_2 ! 主プログラム
  implicit none
  Print *, func3(0.5,2.0) ! 関数の呼び出し
  stop

contains
  function func3(x,a) result(y) ! 関数副プログラム
    real, intent(IN) :: x,a
    real :: y
    y = exp(a*x)
  end function func3

end program f1_function_result_2
```

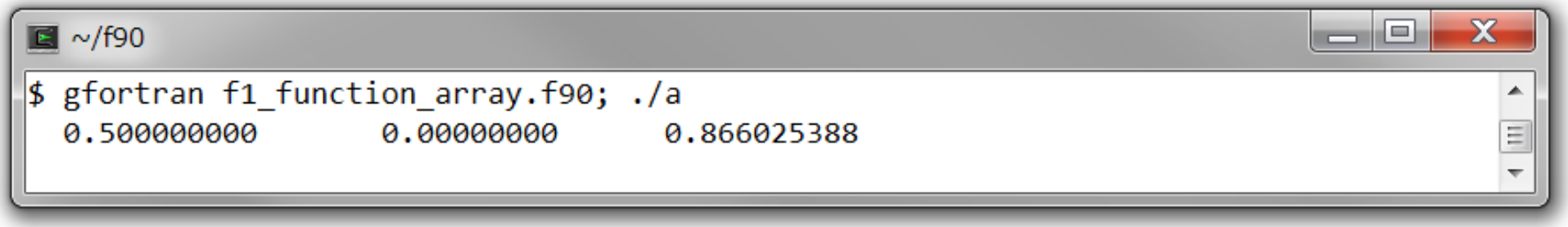
コンパイル，リンク，実行すると，



■戻り値が配列の内部関数副プログラム例　(f1_function_array.f90)

```
program f1_function_array
  implicit none
  real :: q = 45.0/atan(1.0)
  print *, func4(30.0/q, 0.0/q) ! degrees/q
  stop
contains
  function func4(th,phi) result(uvec) ! 戻り値が配列の場合
    real, intent(IN) :: th,phi
    real :: uvec(3)
    real :: st,ct,sp,cp
    st = sin(th);  ct = cos(th)
    sp = sin(phi); cp = cos(phi)
    uvec(1) = st*cp; uvec(2) = st*sp; uvec(3) = ct
  end function func4
end program f1_function_array
```

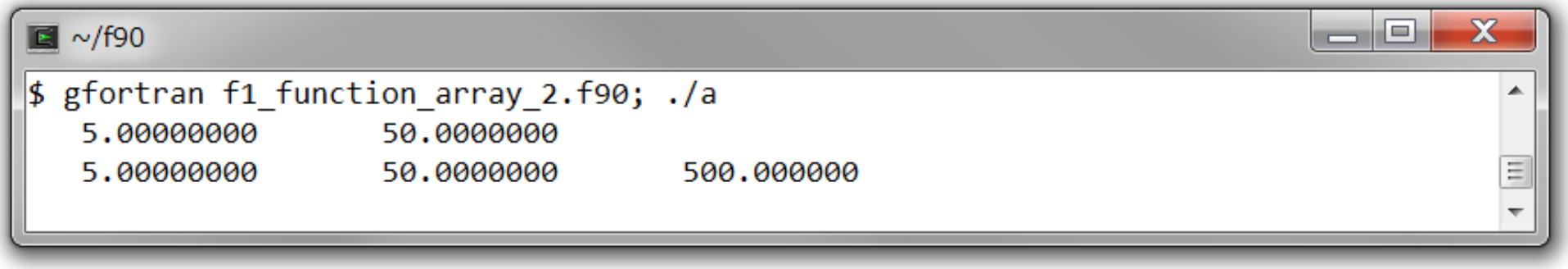
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_function_array.f90; ./a
0.500000000 0.00000000 0.866025388

■引数を配列とし，要素ごとの演算結果を配列で返す内部関数副プログラム例 (f1_function_array_2.f90)

```
program f1_function_array_2 ! 主プログラム
  implicit none
  print *, funca( (/3.0, 30.0/), [4.0, 40.0] )
  print *, funca( (/3.0, 30.0, 300.0/), [4.0, 40.0, 400.0] )
  stop
contains
  function funca(x,y) result(z) ! 仮引数，戻り値が配列
    real, intent(IN) :: x(:),y(:)
    real :: z(size(x))
    z(:) = sqrt(x(:)**2 + y(:)**2)
  end function funca
end program f1_function_array_2
```

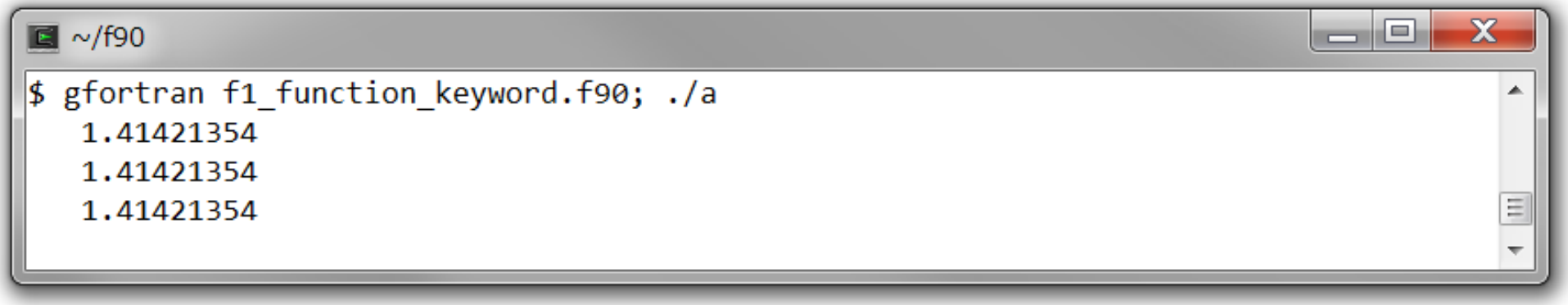
コンパイル，リンク，実行すると，



■引数キーワードに関する内部関数副プログラム例　（f1_function_keyword.f90）

```
program f1_function_keyword
  implicit none
  print *, kfunc(x=1.0,y=1.0,a=0.5)
  print *, kfunc(a=0.5,x=1.0,y=1.0)
  print *, kfunc(1.0,a=0.5,y=1.0)
  stop
contains
  function kfunc(x,y,a)
    real, intent(IN) :: x,y,a
    real :: kfunc
    kfunc = (x**2+y**2)**a
  end function kfunc
end program f1_function_keyword
```

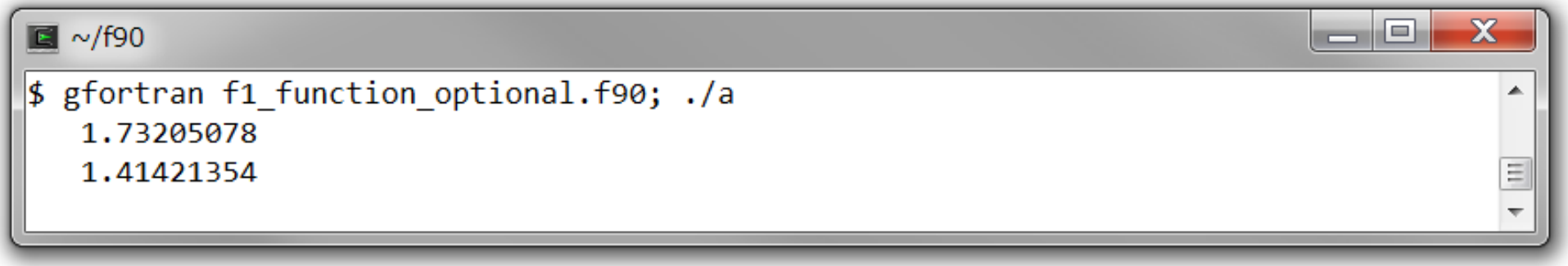
コンパイル，リンク，実行すると，



■OPTIONAL 属性に関する内部関数副プログラム例　内部関数で実引数の一部を省略した場合（f1_function_optional.f90).

```
program f1_function_optional
  implicit none
  print *, ofunc(1.0,1.0,1.0) ! 実引数を省略していない場合
  print *, ofunc(1.0,1.0) ! 実引数を省略した場合
  stop

contains
  function ofunc(x,y,z)
    real, intent(IN) :: x,y
    real, intent(IN), optional :: z ! optional 属性
    real :: ofunc
    if(present(z)) then ! present 組込関数
       ofunc = sqrt(x**2+y**2+z**2)
    else
       ofunc = sqrt(x**2+y**2)
    endif
  end function ofunc

end program f1_function_optional
```

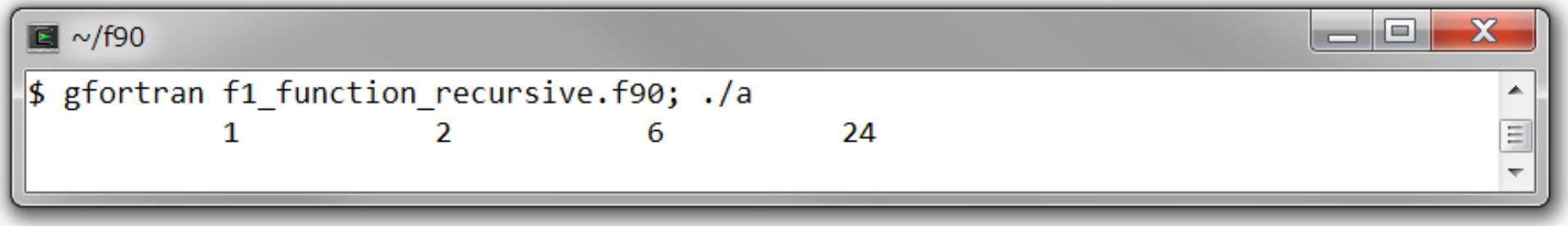
コンパイル，リンク，実行すると，



■RECURSIVE FUNCTION に関する内部関数副プログラム例　再帰コールによる階乗計算（f1_function_recursive.f90）.

```
program f1_function_recursive ! 主プログラム
  implicit none
  Print *, rfunc(1),rfunc(2),rfunc(3),rfunc(4)
  stop
contains

  recursive function rfunc(n) result(k) ! recursive 接頭辞
    integer, intent(IN) :: n
    integer :: k
    if(n==1) then
       k = 1
    else
       k = n*rfunc(n-1)
    endif
    return
  end function rfunc

end program f1_function_recursive
```

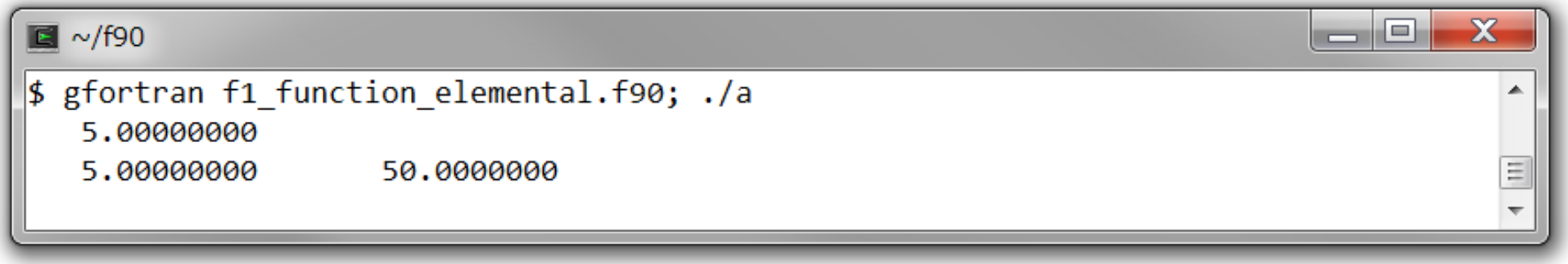
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_function_recursive.f90; ./a
1 2 6 24

■ELEMENTAL FUNCTION に関する内部関数副プログラム例　実引数および戻り値が，スカラと 1 次元配列の場合（f1_function_elemental.f90）.

```
program f1_function_elemental ! 主プログラム
  implicit none
  print *, efunc(3.0, 4.0)
  print *, efunc( (/3.0, 30.0/), [4.0, 40.0] )
  stop
contains
  elemental function efunc(x,y) result(z) ! elemental 接頭辞
    real, intent(IN) :: x,y
    real :: z
    z = x**2 + y**2
    z = sqrt(z)
  end function efunc
end program f1_function_elemental
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_function_elemental.f90; ./a
5.00000000
5.00000000 50.0000000

15.3　内部サブルーチン副プログラム

副プログラムがサブルーチンの場合，次のように内部サブルーチン副プログラムとなる.

```
PROGRAM prog-name ! 主プログラム
    ...
    CALL sub-name (a-arg-1 [, a-arg-2, ... ] ) ! サブルーチンの呼び出し
    ...
    CONTAINS
    SUBROUTINE sub-name (d-arg-1 [, d-arg-2, ... ] ) ! 内部サブルーチン副プログラム
        ...
    END [SUBROUTINE [sub-name] ]
END [PROGRAM [prog-name] ]
```

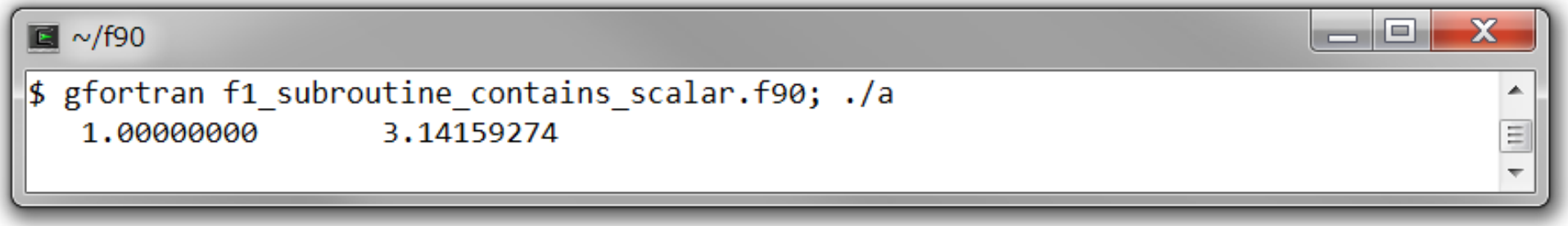
（互換性：Fortran 90〜）

■INTENT 属性を用いない内部サブルーチン副プログラム例

(f1_subroutine_contains_scalar.f90)

```
program f1_subroutine_contains_scalar ! 主プログラム
  implicit none
  real r,th
  call subi(-1.0, 0.0) ! サブルーチンの呼び出し
  print *,r,th
  stop
  contains
    subroutine subi(x,y) ! 内部サブルーチン副プログラム
      real x,y
      r = sqrt(x**2 + y**2); th = atan2(y,x)
      return
    end subroutine subi
end program f1_subroutine_contains_scalar
```

コンパイル，リンク，実行すると，

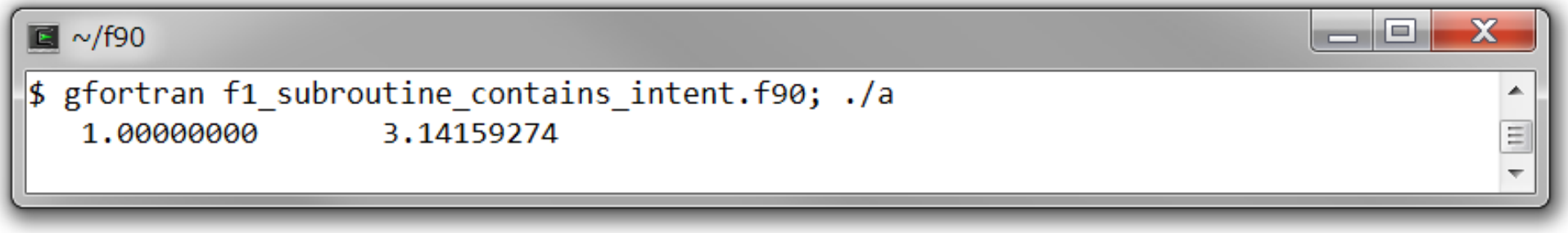


■INTENT 属性を用いた単精度の内部サブルーチン副プログラム例

(f1_subroutine_contains_intent.f90)

```
program f1_subroutine_contains_intent ! 主プログラム
  implicit none
  real :: rr,tth
  call sub(-1.0, 0.0 ,rr,tth) ! サブルーチンの呼び出し
  print *,rr,tth
  stop
  contains
    subroutine sub(x,y, r,th) ! 内部サブルーチン副プログラム
      real, intent(IN) :: x,y ! intent 属性
      real, intent(OUT) :: r,th ! intent 属性
      r = sqrt(x**2 + y**2); th = atan2(y,x)
      return
    end subroutine sub
end program f1_subroutine_contains_intent
```

コンパイル，リンク，実行すると，

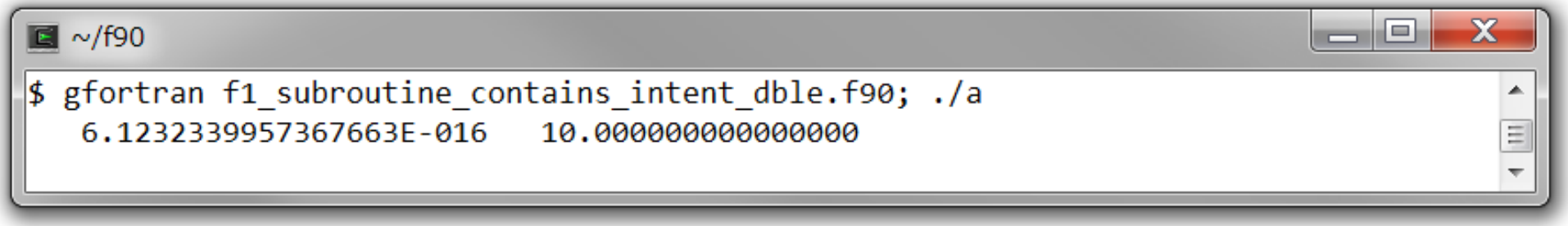

~/f90
$ gfortran f1_subroutine_contains_intent.f90; ./a
1.00000000 3.14159274

■INTENT 属性を用いた倍精度の内部サブルーチン副プログラム例

(f1_subroutine_contains_intent_dble.f90)

```
program f1_subroutine_contains_intent_dble ! 主プログラム
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  real(8) :: xx,yy
  call dsub2(10.0D0,90.0D0/q, xx,yy) ! サブルーチンの呼び出し
  print *,xx,yy
  stop
  contains
    subroutine dsub2(r,phi, x,y) ! 内部サブルーチン副プログラム
      real(8), intent(IN) :: r,phi ! intent 属性
      real(8), intent(OUT) :: x,y ! intent 属性
      x = r*cos(phi)
      y = r*sin(phi)
      return
    end subroutine dsub2
end program f1_subroutine_contains_intent_dble
```

コンパイル，リンク，実行すると，

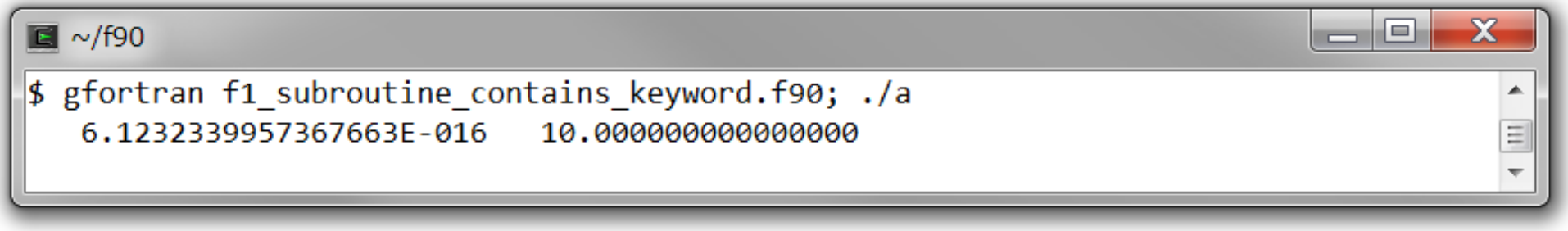

~/f90
$ gfortran f1_subroutine_contains_intent_dble.f90; ./a
6.1232339957367663E-016 10.000000000000000

■引数キーワードを指定した内部サブルーチン副プログラム例

(f1_subroutine_contains_keyword.f90)

```
program f1_subroutine_contains_keyword ! 主プログラム（倍精度）
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  real(8) :: x,y
  call dsub2(rect_x=x, rect_y=y, &
       radius=10.0D0, angle_phi=90.0D0/q) ! 引数キーワード指定
  print *,x,y
  stop

  contains
    subroutine dsub2(radius,angle_phi, rect_x,rect_y) ! 内部サブルーチン
      real(8), intent(IN) :: radius,angle_phi
      real(8), intent(OUT) :: rect_x,rect_y
      rect_x = radius*cos(angle_phi)
      rect_y = radius*sin(angle_phi)
      return
    end subroutine dsub2
end program f1_subroutine_contains_keyword
```

コンパイル，リンク，実行すると，

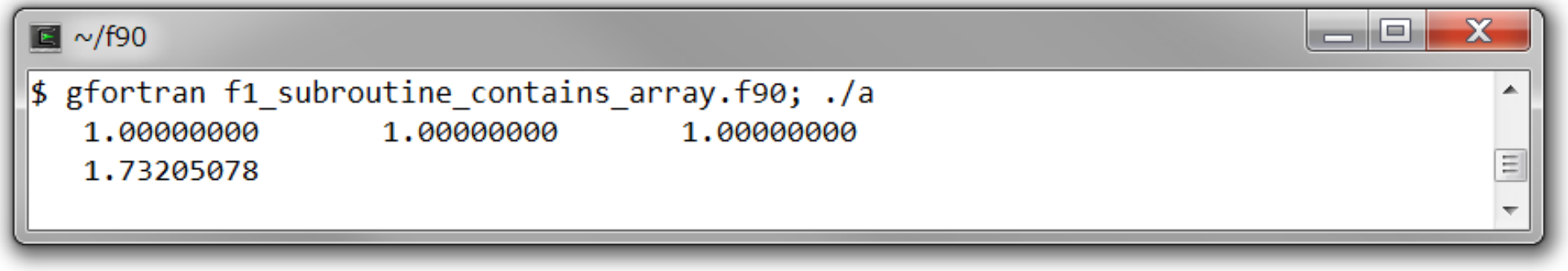

~/f90
$ gfortran f1_subroutine_contains_keyword.f90; ./a
6.1232339957367663E-016 10.000000000000000

■形状明示配列（explicit-shape array）を引数とする内部サブルーチン副プログラム例（f1_subroutine_contains_array.f90）

```
program f1_subroutine_contains_array ! 主プログラム
  implicit none
  integer, parameter :: nn = 11
  real :: xx(nn),aa
  xx(1:3) = [1.0, 1.0, 1.0]
  print *,xx(1:3)
  call suba(xx,3, aa)
  print *,aa
  stop

  contains
    subroutine suba(x,n, a) ! 内部サブルーチン副プログラム
      integer, intent(IN) :: n
      real, intent(IN) :: x(nn)
      real, intent(OUT) :: a
      integer :: i
      a = 0.0
      do i=1,n
         a = a+x(i)**2
      enddo
      a = sqrt(a)
```

```
      return
    end subroutine suba
end program f1_subroutine_contains_array
```

コンパイル，リンク，実行すると，

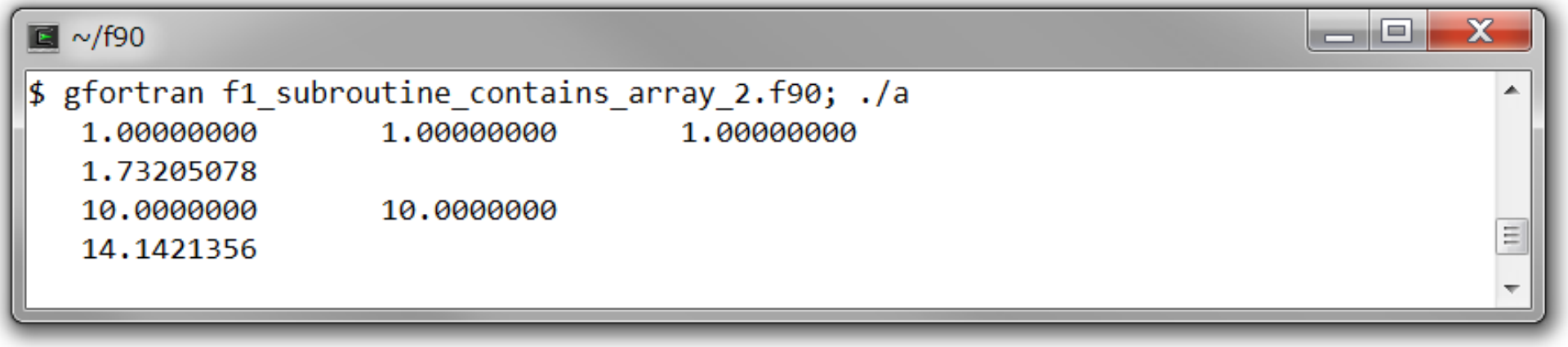


■形状引継配列（assumed-shape array）を引数とする内部サブルーチン副プログラム例（f1_subroutine_contains_array_2.f90）

```
program f1_subroutine_contains_array_2 ! 主プログラム
  implicit none
  integer, parameter :: nn = 11, mm = 5
  real :: xx(nn),yy(mm),aa,bb
  xx(1:3) = [1.0, 1.0, 1.0]
  print *,xx(1:3)
  call suba(xx,3, aa)
  print *,aa
  yy(1:2) = [10.0, 10.0]
  print *,yy(1:2)
  call suba(yy,2, bb)
  print *,bb
  stop

  contains
    subroutine suba(x,n, a) ! 内部サブルーチン副プログラム
      integer, intent(IN) :: n
      real, intent(IN) :: x(:)
      real, intent(OUT) :: a
      integer :: i
      a = 0.0
```

```
      do i=1,n
         a = a+x(i)**2
      enddo
      a = sqrt(a)
      return
    end subroutine suba
end program f1_subroutine_contains_array_2
```

コンパイル，リンク，実行すると，

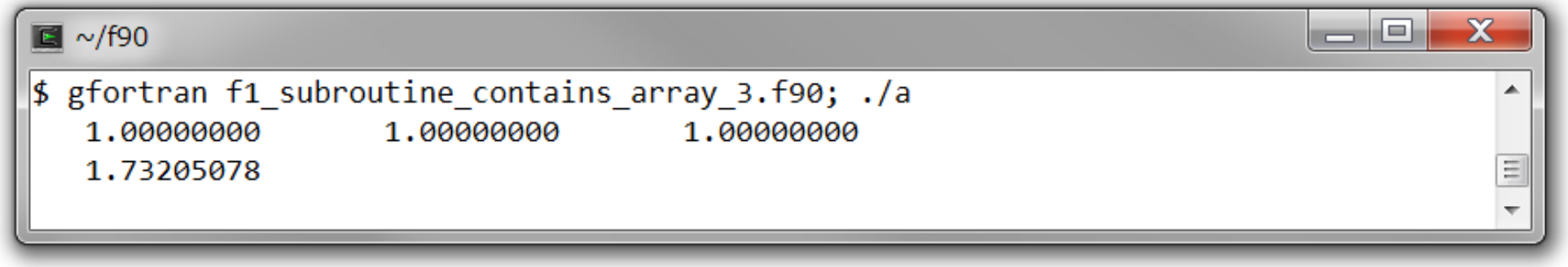


■動的割り付けした配列を引数とする内部サブルーチン副プログラム例（f1_subroutine_contains_array_3.f90）

```
program f1_subroutine_contains_array_3 ! 主プログラム
  implicit none
  integer :: nn = 3, is
  real, allocatable :: xx(:) ! 形状無指定配列 (deferred-shape array)
  real :: aa
  allocate(xx(nn),stat=is) ! 配列の割り付け
  if(is/=0) stop 'cannot allocate'
  xx(1:nn) = [1.0, 1.0, 1.0]
  print *,xx(1:nn)
  call suba(xx, aa)
  print *,aa
  deallocate(xx) ! 割り付けの解除
  stop

  contains
    subroutine suba(x, a) ! 内部サブルーチン副プログラム
      real, intent(IN) :: x(:)
      real, intent(OUT) :: a
      integer :: i,n
      a = 0.0
      n = size(x) ! 配列の問い合わせ関数
```

```
      do i=1,n
         a = a+x(i)**2
      enddo
      a = sqrt(a)
      return
    end subroutine suba
end program f1_subroutine_contains_array_3
```

コンパイル，リンク，実行すると，

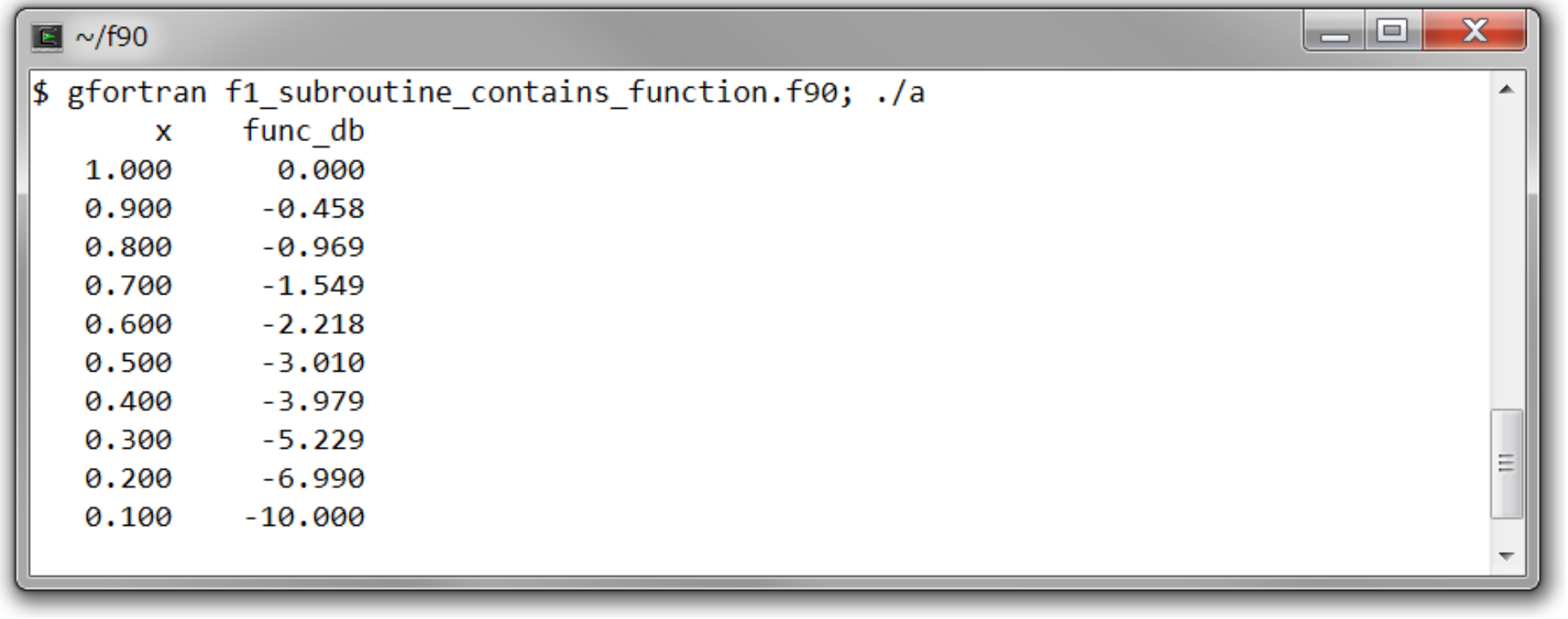


■スカラ関数副プログラムを引数とする倍精度の内部サブルーチン副プログラム例 1 (f1_subroutine_contains_function.f90)

```
program f1_subroutine_contains_function ! 主プログラム
  implicit none
  call sub_f(func_db, 1.0D0, -0.1D0, 10)
  stop
  contains
    subroutine sub_f(func, x0,dx,nx) ! 内部サブルーチン副プログラム
      real(8), intent(IN) :: x0,dx
      integer, intent(IN) :: nx
      integer :: i
      real(8) :: func,x
      external func
      print '(a8,1x,a10)','x','func_db'
      do i=1,nx
         x = x0+dx*(i-1)
         print '(f8.3,1x,f10.3)',x,func(x)
      enddo
      return
    end subroutine sub_f
    real(8) function func_db(x) result(y) ! 内部関数副プログラム
      real(8), intent(IN) :: x
      y = abs(x)
```

```
      if(y/=0.0D0) then
         y = 10.0D0*log10(y)
      else
         y = -800.0D0
      endif
      return
    end function func_db
end program f1_subroutine_contains_function
```

コンパイル，リンク，実行すると，



■スカラ関数副プログラムを引数とする倍精度の内部サブルーチン副プログラム例２（f1_subroutine_contains_function_optional.f90）

```
program f1_subroutine_contains_function_optional ! 主プログラム
  implicit none
  call sub_f(func_db, 5, 1.0D0, -0.5D0)
  call sub_f(func=func_db, nx=5,xmin=1.0D0,delx=-0.05D0)
  call sub_f(func_db, 5, xmin=1.0D0)
  call sub_f(func_db, 3)
  stop
  contains
    subroutine sub_f(func, nx,xmin,delx) ! 内部サブルーチン副プログラム
      integer, intent(IN) :: nx
      real(8), optional :: xmin,delx ! optional 属性
      real(8) :: func
      integer :: i
      real(8) :: x,x0,dx
      external func
      if(present(xmin)) then ! present 関数
         x0 = xmin
      else
         x0 = 1.0D0
      endif
      if(present(delx)) then ! present 関数
```

```
        dx = delx
      else
        dx = -x0/(nx-1)
      endif
      print *,'x0=',x0,'dx=',dx
      print '(a8,1x,a10)','x','func_db'
      do i=1,nx
        x = x0+dx*(i-1)
        print '(f8.3,1x,f10.3)',x,func(x)
      enddo
      return
    end subroutine sub_f
    real(8) function func_db(x) result(y) ! 内部関数副プログラム
      real(8), intent(IN) :: x
      y = abs(x)
      if(y/=0.0D0) then
        y = 10.0D0*log10(y)
      else
        y = -800.0D0
      endif
      return
    end function func_db
end program f1_subroutine_contains_function_optional
```

コンパイル，リンク，実行すると，

~/f90

```
$ gfortran f1_subroutine_contains_function_optional.f90; ./a
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000       0.000
   0.500      -3.010
   0.000    -800.000
  -0.500      -3.010
  -1.000       0.000
 x0=   1.0000000000000000      dx=  -5.0000000000000003E-002
       x    func_db
   1.000       0.000
   0.950      -0.223
   0.900      -0.458
   0.850      -0.706
   0.800      -0.969
 x0=   1.0000000000000000      dx= -0.25000000000000000
       x    func_db
   1.000       0.000
   0.750      -1.249
   0.500      -3.010
   0.250      -6.021
   0.000    -800.000
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000       0.000
   0.500      -3.010
   0.000    -800.000
```

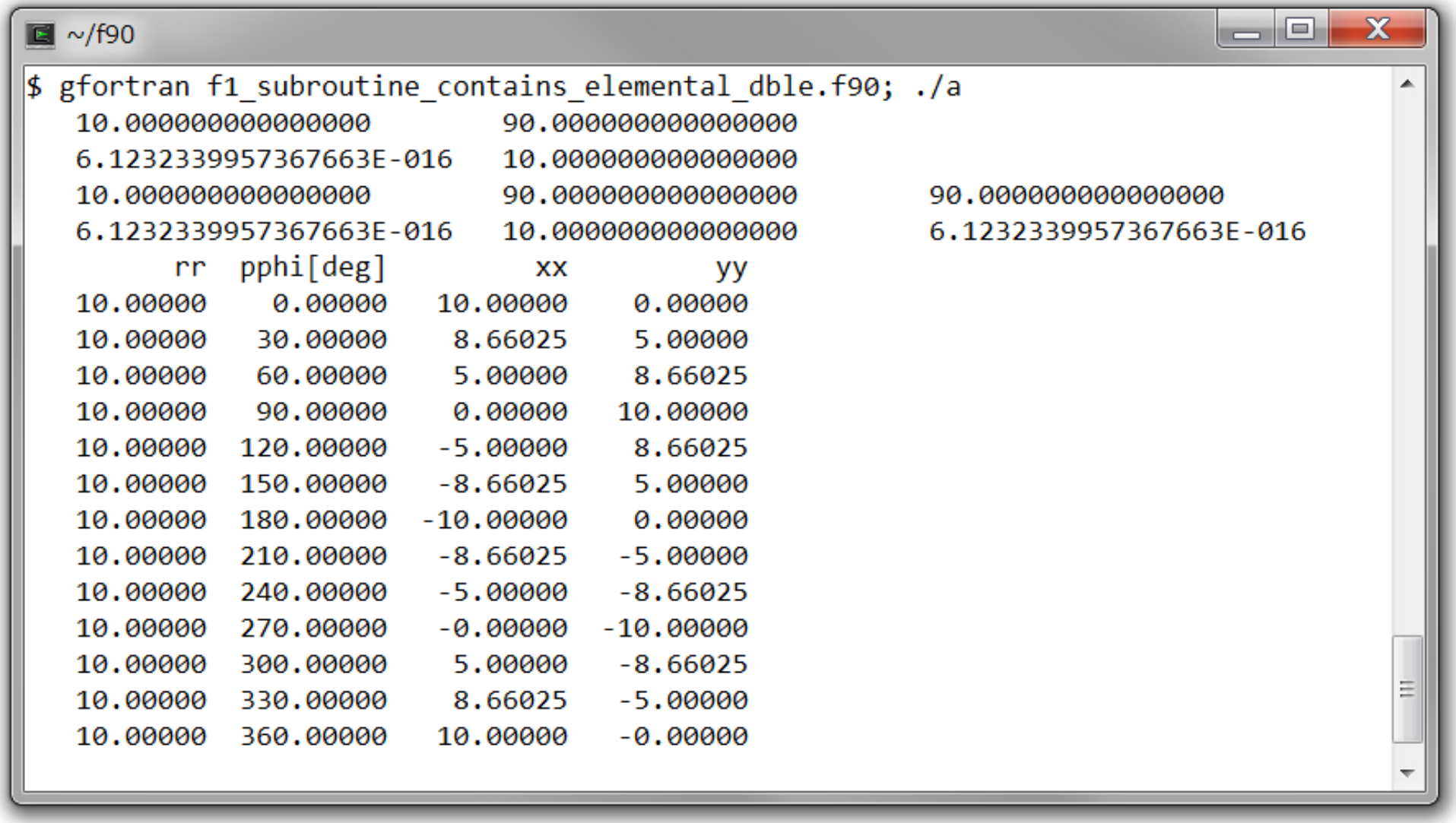
■ELEMENTAL 接頭辞を用いた倍精度の内部サブルーチン副プログラム例 (f1_subroutine_contains_elemental_dble.f90)

```
program f1_subroutine_contains_elemental_dble ! 主プログラム（倍精度）
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  integer, parameter :: nn = 101
  real(8) :: x,y,r,phi, z,th
  real(8) :: xx(nn),yy(nn),zz(nn),rr(nn),tth(nn),pphi(nn)
  real(8) :: phi0 = 0.0D0, dphi = 30.0D0/q
  integer :: i, nphi = 13

  r = 10.0D0; phi = 90.0D0/q
  print *,r,phi*q
  call esub2(rect_x=x, rect_y=y, radius=r, angle_phi=phi) ! 引数はスカラー
  print *,x,y

  th = 90.0D0/q
  print *,r,th*q,phi*q
  call esub2(rect_x=x, rect_y=y, rect_z=z, &
       radius=r, angle_phi=phi, angle_theta=th) ! 引数はスカラー
  print *,x,y,z

  rr(1:nphi) = 10.0D0
```

```
pphi(1:nphi) = [ (phi0+dphi*(i-1),i=1,nphi) ]
call esub2(rect_x=xx, rect_y=yy, radius=rr, angle_phi=pphi) ! 引数は配列
print '(4a11)','rr','pphi[deg]','xx','yy'
do i=1,nphi
   print '(4(1x,f10.5))',rr(i),pphi(i)*q,xx(i),yy(i)
enddo
stop
contains
  elemental subroutine esub2(radius,angle_theta,angle_phi, &
       rect_x,rect_y,rect_z) ! ELEMENTAL SUBROUTINE
    real(8), intent(IN) :: radius,angle_phi
    real(8), intent(OUT) :: rect_x,rect_y
    real(8), optional, intent(IN) :: angle_theta ! optional 属性
    real(8), optional, intent(OUT) :: rect_z ! optional 属性
    real(8) :: ct,st,cp,sp
    cp = cos(angle_phi); sp = sin(angle_phi)
    ct = 0.0D0; st = 1.0D0
    if(present(angle_theta)) then ! present 関数
       ct = cos(angle_theta)
       st = sin(angle_theta)
    endif
    rect_x = radius*st*cp
    rect_y = radius*st*sp
```

```
      if(present(rect_z)) rect_z = radius*ct ! present 関数
  return
end subroutine esub2
end program f1_subroutine_contains_elemental_dble
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_subroutine_contains_elemental_dble.f90; ./a
10.000000000000000 90.000000000000000
6.1232339957367663E-016 10.000000000000000
10.000000000000000 90.000000000000000 90.000000000000000
6.1232339957367663E-016 10.000000000000000 6.1232339957367663E-016
rr pphi[deg] xx yy
10.00000 0.00000 10.00000 0.00000
10.00000 30.00000 8.66025 5.00000
10.00000 60.00000 5.00000 8.66025
10.00000 90.00000 0.00000 10.00000
10.00000 120.00000 -5.00000 8.66025
10.00000 150.00000 -8.66025 5.00000
10.00000 180.00000 -10.00000 0.00000
10.00000 210.00000 -8.66025 -5.00000
10.00000 240.00000 -5.00000 -8.66025
10.00000 270.00000 -0.00000 -10.00000
10.00000 300.00000 5.00000 -8.66025
10.00000 330.00000 8.66025 -5.00000
10.00000 360.00000 10.00000 -0.00000

16　モジュール

モジュール（modules）とは，主プログラム，外部副プログラムとは別の新しい形式のプログラム単位で，ユーザー定義の演算・代入（関数・サブルーチンの高機能化）に用いられ，また，データの共用（COMMON 文の難点を解消する新機能），初期値の定義（DATA 文等より優れた新機能）にも使用でき，より安全で移植性の高いプログラムを作成することができる.

16.1　MODULE

モジュールの定義は,

```
MODULE module-name
    .....
END [MODULE [module-name] ]
```

（互換性：Fortran 90～）

- *module-name*：モジュール名.

16.1.1 USE 文

モジュールの参照は,

```
USE module-name
```

（互換性：Fortran 90～）

主プログラムで参照する場合は,

```
MODULE module-name ! モジュール
    .....
END [MODULE [module-name] ]

PROGRAM main-name ! 主プログラム
    USE module-name
    IMPLICUT NONE
    .....
    STOP
END [PROGRAM [main-name] ]
```

（互換性：Fortran 90～）

モジュールを参照するとき，名前を局所的に変更する場合は，

```
USE module-name, local-name => mod-name
```

（互換性：Fortran 90〜）

- *module-name*：モジュール名.
- *local-name*：モジュールを使用するプログラム単位の言語要素名.
- *mod-name*：モジュール内の公開言語要素名.

■use 文に関するプログラム例　　（f1_use.f90）

```
module set ! モジュール
  integer :: n = 1
  real :: a = 10.0
end module

program f1_use ! 主プログラム
use set ! use 文
!  implicit none
  print *,n,a
  call sub()
```

```
  print *,n,a
  stop
end program f1_use

subroutine sub() ! 外部サブルーチン副プログラム
  use set, x => a ! use 文
  n = 5 ! そのままの名前でデータ共有
  x = 5.0 ! 名前を変更してデータ共有
  return
end subroutine sub
```

コンパイル，リンク，実行すると，

16.2　モジュールインターフェース

モジュール内に，インターフェースブロックをおいて引用仕様宣言することができ，このようなモジュールをモジュールインターフェースともいう.

```
MODULE module-name ! モジュール
    INTERFACE ! インターフェース・ブロック
          .....
    END INTERFACE
END [MODULE [module-name] ]

PROGRAM main-name ! 主プログラム
    USE module-name
    IMPLICUT NONE
    .....
    STOP
END [PROGRAM [main-name] ]

..... ! 外部副プログラム
```

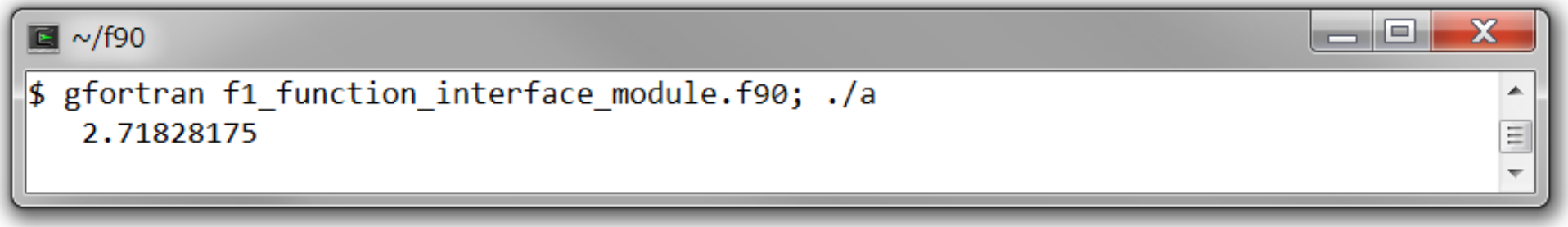
（互換性：Fortran 90～）

16.2.1　外部関数副プログラムのモジュールインターフェース

■RESULT 句を用いない外部関数副プログラムのモジュールインターフェースに関するプログラム例　　（f1_function_interface_module.f90）

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     real function func1(x,a)
       real, intent(IN) :: x,a
     end function func1
  end interface
end module interface_mod

program f1_function_interface_module ! 主プログラム
  use interface_mod ! use 文
  implicit none
  print *, func1(0.5,2.0) ! 関数の呼び出し
end program f1_function_interface_module

real function func1(x,a) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: x,a
  func1 = exp(a*x)
end function func1
```

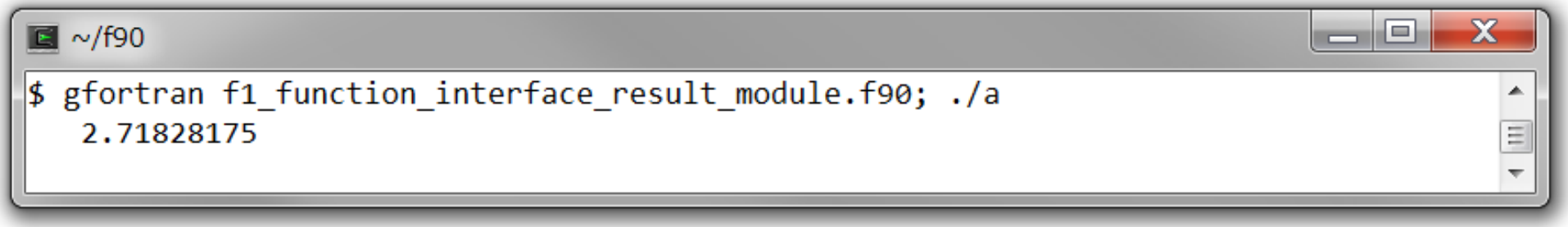
コンパイル，リンク，実行すると，



■RESULT 句によってスカラを返す外部関数副プログラムのモジュールインターフェースに関するプログラム例　（f1_function_interface_result_module.f90）

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     real function func3(x,a) result(y)
       real, intent(IN) :: x,a
     end function func3
  end interface
end module interface_mod

program f1_function_interface_result_module ! 主プログラム
  use interface_mod ! use 文
  implicit none
  real :: a=2.0
  print *, func3(0.5,a) ! 関数の呼び出し
end program f1_function_interface_result_module

real function func3(x,a) result(y) ! 外部関数副プログラム（戻り値がスカラ）
  implicit none
  real, intent(IN) :: x,a
  y = exp(a*x)
end function func3
```

コンパイル，リンク，実行すると，

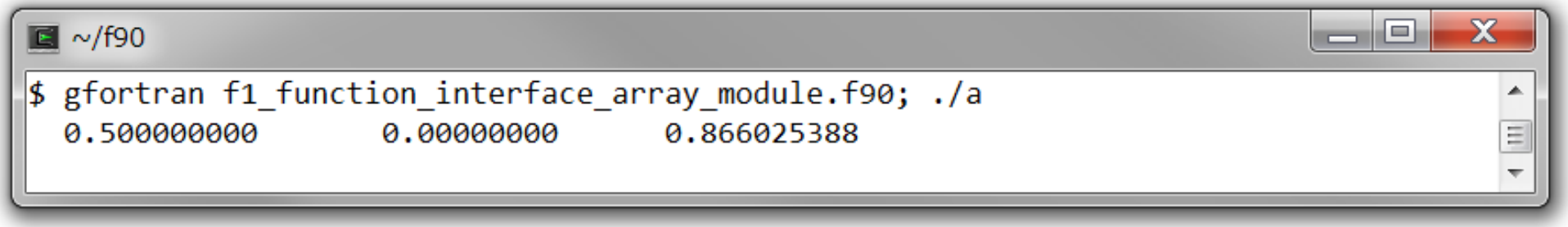


■RESULT 句によって配列を返す外部関数副プログラムのモジュールインターフェースに関するプログラム例　　(f1_function_interface_array_module.f90).

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     function func4(th,phi) result(uvec)
       real, intent(IN) :: th,phi
       real :: uvec(3)
     end function func4
  end interface
end module interface_mod

program f1_function_interface_array_module ! 主プログラム
  use interface_mod ! use 文
  implicit none
  real :: q = 45.0/atan(1.0)
  print *, func4(30.0/q, 0.0/q) ! degrees/q
  stop
end program f1_function_interface_array_module

function func4(th,phi) result(uvec) ! 外部関数副プログラム（戻り値が配列）
  implicit none
  real, intent(IN) :: th,phi
  real :: uvec(3),st,ct,sp,cp
```

```
  st = sin(th);  ct = cos(th)
  sp = sin(phi); cp = cos(phi)
  uvec(1) = st*cp; uvec(2) = st*sp; uvec(3) = ct
  return
end function func4
```

コンパイル，リンク，実行すると，

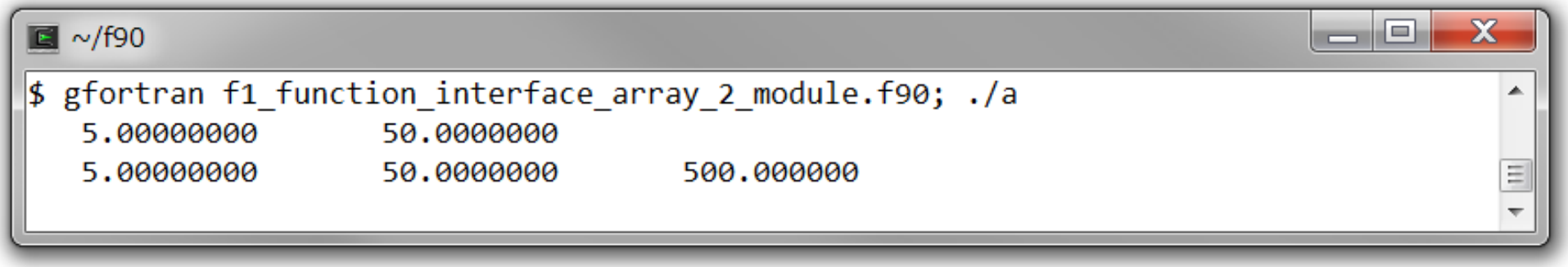


■仮引数および戻り値がとも配列の外部関数副プログラムのモジュールインターフェースに関するプログラム例　　(f1_function_interface_array_2_module.f90).

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     function funca(x,y) result(z)
       real, intent(IN) :: x(:),y(:)
       real :: z(size(x))
     end function funca
  end interface
end module interface_mod

program f1_function_interface_array_2_module ! 主プログラム
  use interface_mod ! use文
  implicit none
  print *, funca( (/3.0, 30.0/), [4.0, 40.0] )
  print *, funca( (/3.0, 30.0, 300.0/), [4.0, 40.0, 400.0] )
  stop
end program f1_function_interface_array_2_module

function funca(x,y) result(z) ! 外部関数副プログラム（仮引数，戻り値が配列）
  implicit none
  real, intent(IN) :: x(:),y(:)
  real :: z(size(x))
```

```
  z(:) = sqrt(x(:)**2 + y(:)**2)
  return
end function funca
```

コンパイル，リンク，実行すると，



16.2.2　外部サブルーチン副プログラムのモジュールインターフェース

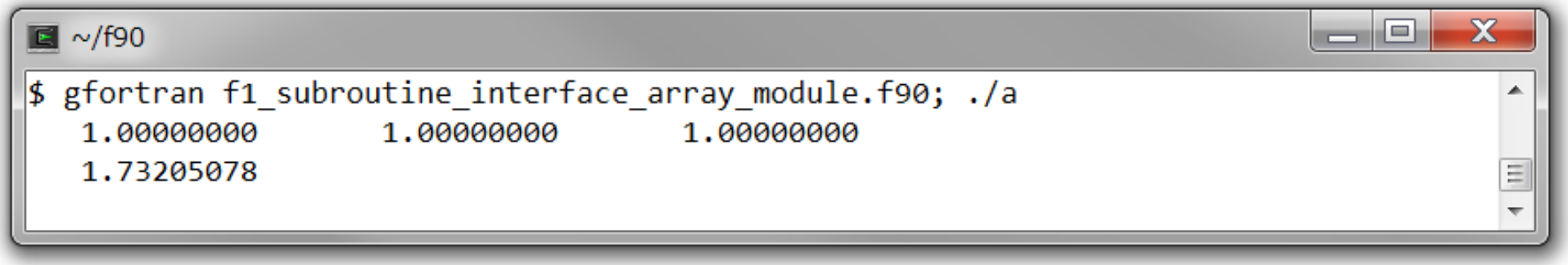
■形状明示配列（explicit-shape array）を引数とする外部サブルーチン副プログラムのモジュールインターフェースに関するプログラム例

（f1_subroutine_interface_array_module.f90）

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     subroutine suba(x,n,nn, a)
       integer, intent(IN) :: n,nn
       real, intent(IN) :: x(nn)
       real, intent(OUT) :: a
     end subroutine suba
  end interface
end module interface_mod
program f1_subroutine_interface_array_module ! 主プログラム
  use interface_mod ! use文
  implicit none
  integer, parameter :: nn = 11
  integer :: n = 3
  real :: x(nn),a
  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
```

```
  call suba(x,n,nn, a)
  print *,a
  stop
end program f1_subroutine_interface_array_module
subroutine suba(x,n,nn, a) ! 外部サブルーチン副プログラム
  implicit none
  integer, intent(IN) :: n,nn
  real, intent(IN) :: x(nn)
  real, intent(OUT) :: a
  integer :: i
  a = 0.0
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
end subroutine suba
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_subroutine_interface_array_module.f90; ./a
1.00000000 1.00000000 1.00000000
1.73205078

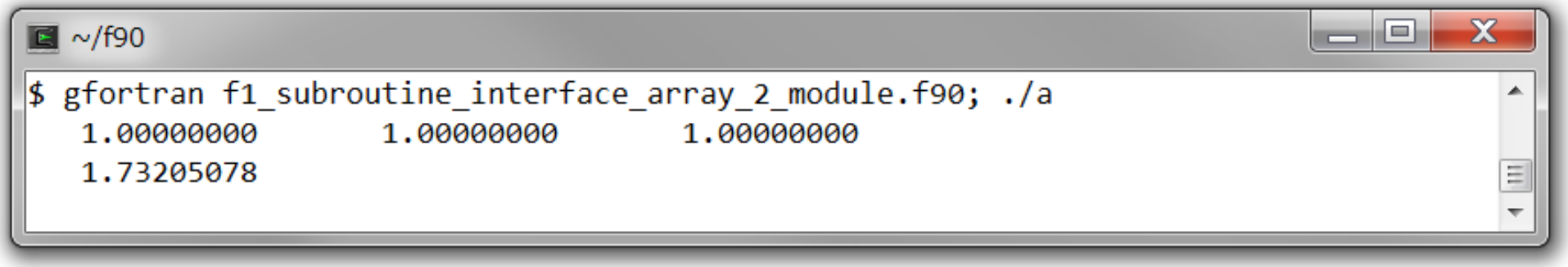
■形状引継配列（assumed-shape array）を引数とする外部サブルーチン副プログラムのモジュールインターフェースに関するプログラム例

（f1_subroutine_interface_array_2_module.f90）

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     subroutine suba(x,n, a)
       integer, intent(IN) :: n
       real, intent(IN) :: x(:)
       real, intent(OUT) :: a
     end subroutine suba
  end interface
end module interface_mod
program f1_subroutine_interface_array_2_module ! 主プログラム
  use interface_mod ! use文
  implicit none
  integer, parameter :: nn = 11
  integer :: n = 3
  real :: x(nn),a
  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
  call suba(x,n, a)
  print *,a
```

```
  stop
end program f1_subroutine_interface_array_2_module
subroutine suba(x,n, a) ! 外部サブルーチン副プログラム
  implicit none
  integer, intent(IN) :: n
  real, intent(IN) :: x(:)
  real, intent(OUT) :: a
  integer :: i
  a = 0.0
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
  return
end subroutine suba
```

コンパイル，リンク，実行すると，



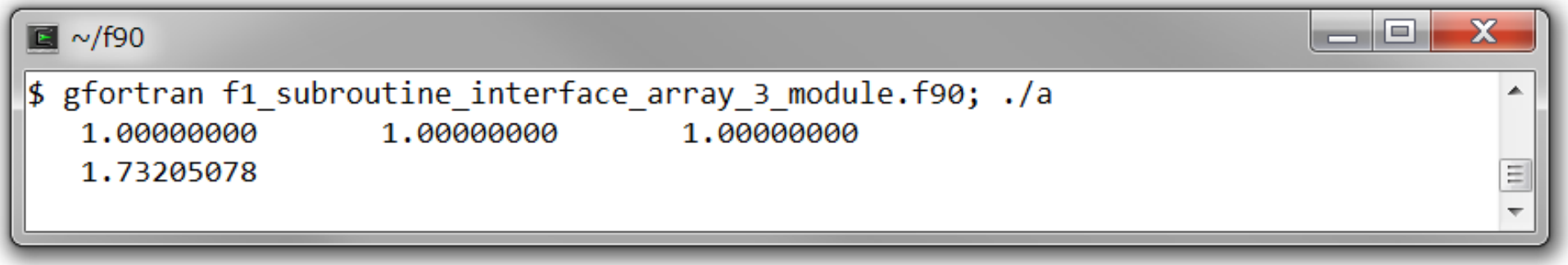
■動的割り付けした配列を引数とする外部サブルーチン副プログラムのモジュールインターフェースに関するプログラム例

（f1_subroutine_interface_array_3_module.f90）

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     subroutine suba(x, a)
       real, intent(IN) :: x(:)
       real, intent(OUT) :: a
     end subroutine suba
  end interface
end module interface_mod
program f1_subroutine_interface_array_3_module ! 主プログラム
  use interface_mod ! use 文
  implicit none
  integer :: n = 3, is
  real, allocatable :: x(:) ! 形状無指定配列 (deferred-shape array)
  real :: a
  allocate(x(n),stat=is) ! 配列の割り付け
  if(is/=0) stop 'cannot allocate'
  x(1:n) = [1.0, 1.0, 1.0]
  print *,x(1:n)
  call suba(x, a)
```

```
  print *,a
  deallocate(x) ! 割り付けの解除
  stop
end program f1_subroutine_interface_array_3_module
subroutine suba(x, a) ! 外部サブルーチン副プログラム
  implicit none
  real, intent(IN) :: x(:)
  real, intent(OUT) :: a
  integer :: i,n
  a = 0.0
  n = size(x) ! 配列の問い合わせ関数
  do i=1,n
     a = a+x(i)**2
  enddo
  a = sqrt(a)
end subroutine suba
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_subroutine_interface_array_3_module.f90; ./a
1.00000000 1.00000000 1.00000000
1.73205078

■引数キーワードおよびOPTIONAL属性をもたせた外部サブルーチン副プログラムのモジュールインターフェースに関するプログラム例

(f1_subroutine_function_interface_module.f90).

```
module interface_mod ! インターフェースモジュール
  interface ! インターフェースブロック
     subroutine sub_f(func, nx,xmin,delx)
       integer, intent(IN) :: nx
       real(8), optional :: xmin,delx
       real(8) :: func
       external func
     end subroutine sub_f
     real(8) function func_db(x) result(y)
       real(8), intent(IN) :: x
     end function func_db
  end interface
end module interface_mod

program f1_subroutine_fucntion_interface_module ! 主プログラム
  use interface_mod ! use文
  implicit none
  real(8) :: x0 = 1.0D0, dx = -0.5D0
  integer :: n = 5
```

```
  call sub_f(func_db, n,x0,dx)
  call sub_f(func=func_db, nx=n,xmin=x0,delx=-0.05D0)
  call sub_f(func_db, n, xmin=1.0D0)
  call sub_f(func_db, 3)
  stop
end program f1_subroutine_fucntion_interface_module

subroutine sub_f(func, nx,xmin,delx) ! 外部サブルーチン副プログラム
  implicit none
  integer, intent(IN) :: nx
  real(8), optional :: xmin,delx ! optional 属性
  real(8) :: func
  integer :: i
  real(8) :: x,x0,dx
  external func

  if(present(xmin)) then ! present 関数
     x0 = xmin
  else
     x0 = 1.0D0
  endif
  if(present(delx)) then ! present 関数
     dx = delx
```

```
  else
     dx = -x0/(nx-1)
  endif
  print *,'x0=',x0,'dx=',dx
  print '(a8,1x,a10)','x','func_db'
  do i=1,nx
     x = x0+dx*(i-1)
     print '(f8.3,1x,f10.3)',x,func(x)
  enddo
  return
end subroutine sub_f

real(8) function func_db(x) result(y) ! 外部関数副プログラム
  implicit none
  real(8), intent(IN) :: x
  y = abs(x)
  if(y/=0.0D0) then
     y = 10.0D0*log10(y)
  else
     y = -800.0D0
  endif
  return
end function func_db
```

コンパイル，リンク，実行すると，

~/f90

```
$ gfortran f1_subroutine_function_interface_module.f90; ./a
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000      0.000
   0.500     -3.010
   0.000   -800.000
  -0.500     -3.010
  -1.000      0.000
 x0=   1.0000000000000000      dx=  -5.0000000000000003E-002
       x    func_db
   1.000      0.000
   0.950     -0.223
   0.900     -0.458
   0.850     -0.706
   0.800     -0.969
 x0=   1.0000000000000000      dx= -0.25000000000000000
       x    func_db
   1.000      0.000
   0.750     -1.249
   0.500     -3.010
   0.250     -6.021
   0.000   -800.000
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000      0.000
   0.500     -3.010
   0.000   -800.000
```

16.3　モジュール副プログラム

モジュールの中に CONTAINS 文を用いれば，内部副プログラムをつくることができ，このようなモジュールを，モジュール副プログラムともいう．

```
MODULE module-name
    [specification-stmts]
[CONTAINS
    module-subprograms]
END [MODULE [module-name] ]
```

（互換性：Fortran 90〜）

16.3.1　モジュール関数副プログラム

```
MODULE module-name ! モジュール
    IMPLICUT NONE
    [specification-stmts] ! 宣言文
CONTAINS
    TYPE-f FUNCTION func (d-arg-1 [, d-arg-2, ... ] ) ! 内部関数副プログラム
        ...
    END [FUNCTION [func] ]
END [MODULE [module-name] ]

PROGRAM main-name ! 主プログラム
    USE module-name
    IMPLICUT NONE
    .....
    func (a-arg-1 [, a-arg-2, ... ] ) ! 関数の呼び出し
    .....
    STOP
END [PROGRAM [main-name] ]
```

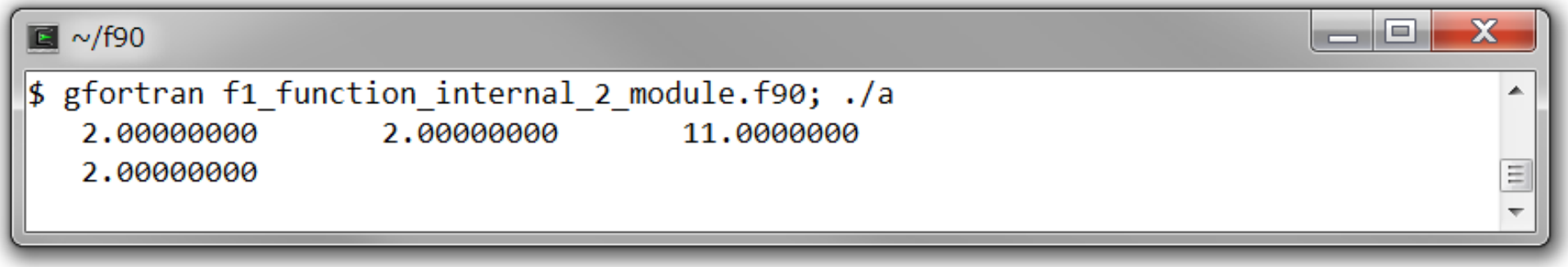
（互換性：Fortran 90〜）

■モジュール関数副プログラムと外部関数副プログラム（インターフェースブロックなし）との比較　（f1_function_internal_2_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none ! contains 文の中には implicit 文は不要
  real :: a=10.0

contains
  real function funcb(b) ! 内部関数副プログラム
    real, intent(IN) :: b
    funcb = b+1.0
  end function funcb

  real function funca(a) ! 内部関数副プログラム
    real, intent(IN) :: a
    funca = a+1.0
  end function funca

  real function func() ! 内部関数副プログラム
    func = a+1.0
  end function func

end module msubprog
```

```
program f1_function_internal_2_module ! 主プログラム
  use msubprog ! use 文は implicit 文の前におく
  implicit none
  real :: funce
  print *, funcb(1.0),funca(1.0),func()
  print *, funce(1.0)
  stop
end program f1_function_internal_2_module

real function funce(a) ! 外部関数副プログラム
  implicit none
  real, intent(IN) :: a
  funce = a+1.0
  return
end function funce
```

コンパイル，リンク，実行すると，

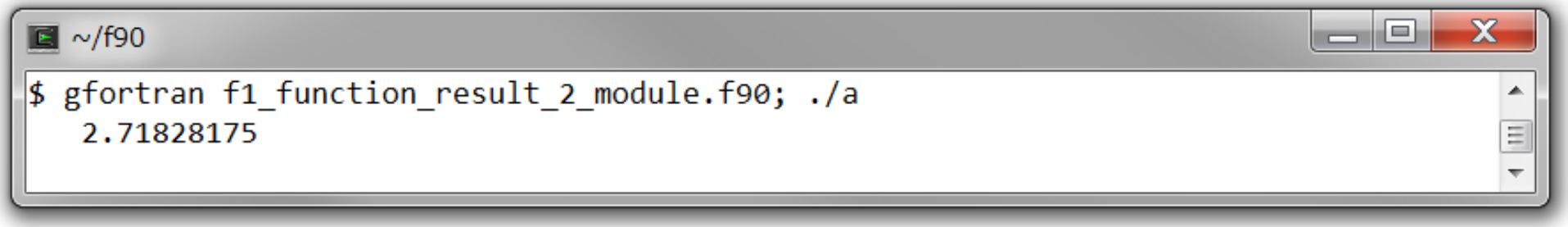


■戻り値がスカラ式のモジュール関数副プログラム例

（f1_function_result_2_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  function func3(x,a) result(y) ! 関数副プログラム
    real, intent(IN) :: x,a
    real :: y
    y = exp(a*x)
  end function func3
end module msubprog
program f1_function_result_2_module ! 主プログラム
  use msubprog ! use 文
  implicit none
  Print *, func3(0.5,2.0) ! 関数の呼び出し
end program f1_function_result_2_module
```

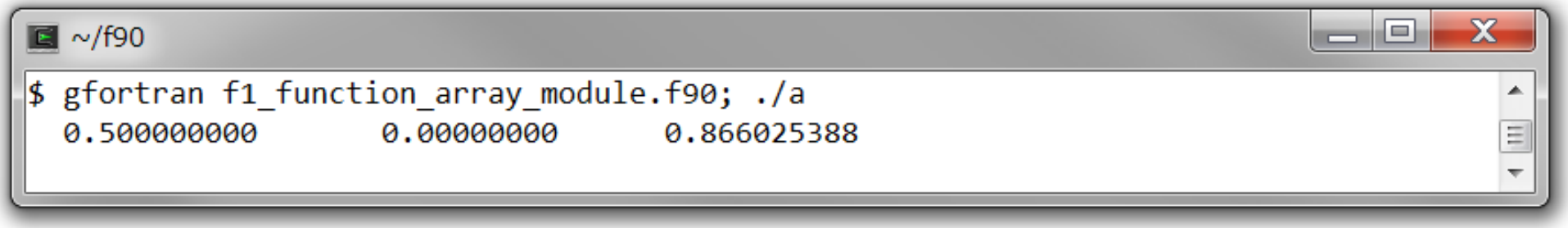
コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_function_result_2_module.f90; ./a
2.71828175

■戻り値が配列のモジュール関数副プログラム例 （f1_function_array_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none
  real :: q = 45.0/atan(1.0)
contains
  function func4(th,phi) result(uvec) ! 戻り値が配列の場合
    real, intent(IN) :: th,phi
    real :: uvec(3)
    real :: st,ct,sp,cp
    st = sin(th);  ct = cos(th)
    sp = sin(phi); cp = cos(phi)
    uvec(1) = st*cp
    uvec(2) = st*sp
    uvec(3) = ct
  end function func4
end module msubprog

program f1_function_array_module ! 主プログラム
  use msubprog ! use文
  implicit none
  print *, func4(30.0/q, 0.0/q) ! degrees/q
  stop
end program f1_function_array_module
```

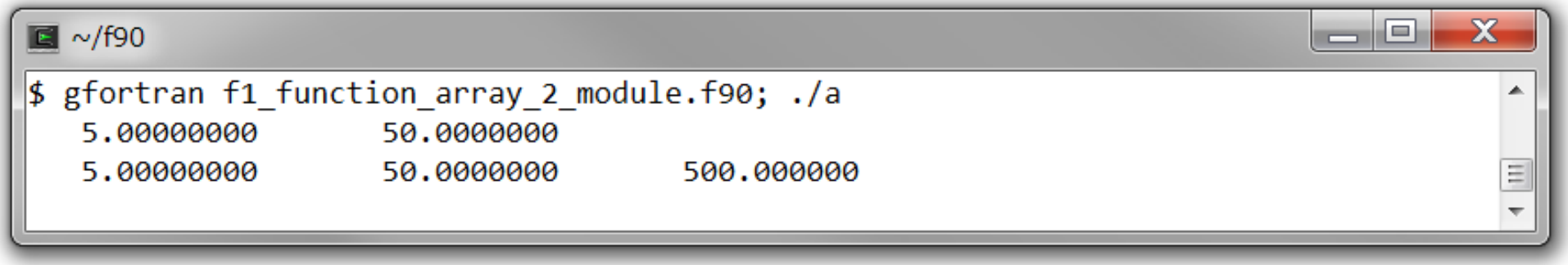
コンパイル，リンク，実行すると，



■引数を配列とし，要素ごとの演算結果を配列で返すモジュール関数副プログラム例(f1_function_array_2_module.f90)

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  function funca(x,y) result(z) ! 仮引数，戻り値が配列
    real, intent(IN) :: x(:),y(:)
    real :: z(size(x))
    z(:) = sqrt(x(:)**2 + y(:)**2)
  end function funca
end module msubprog

program f1_function_array_2_module ! 主プログラム
  use msubprog ! use文
  implicit none
  print *, funca( (/3.0, 30.0/), [4.0, 40.0] )
  print *, funca( (/3.0, 30.0, 300.0/), [4.0, 40.0, 400.0] )
  stop
end program f1_function_array_2_module
```

コンパイル，リンク，実行すると，

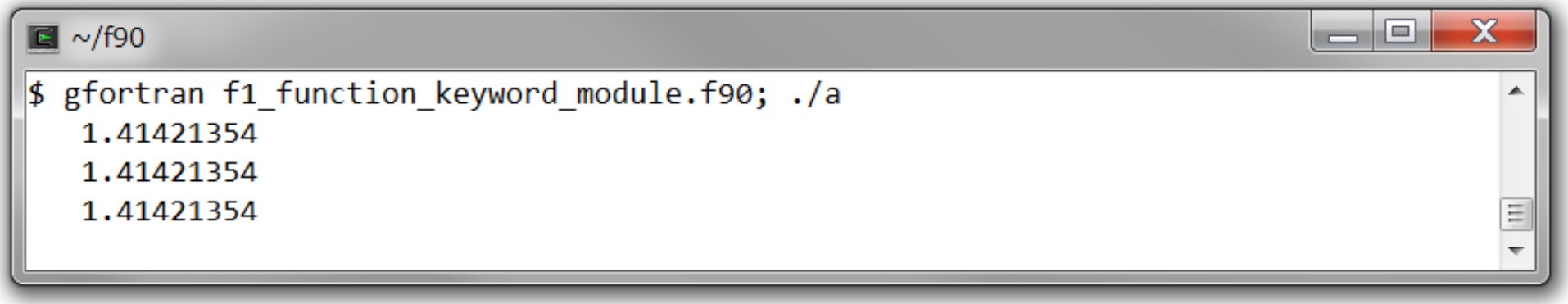

~/f90
$ gfortran f1_function_array_2_module.f90; ./a
5.00000000 50.0000000
5.00000000 50.0000000 500.000000

■引数キーワードに関するモジュール関数副プログラム例

（f1_function_keyword_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  function kfunc(x,y,a) ! 関数副プログラム
    real, intent(IN) :: x,y,a
    real :: kfunc
    kfunc = (x**2+y**2)**a
  end function kfunc
end module msubprog

program f1_function_keyword_module ! 主プログラム
  use msubprog ! use 文
  implicit none
  print *, kfunc(x=1.0,y=1.0,a=0.5)
  print *, kfunc(a=0.5,x=1.0,y=1.0)
  print *, kfunc(1.0,a=0.5,y=1.0)
  stop
end program f1_function_keyword_module
```

コンパイル，リンク，実行すると，

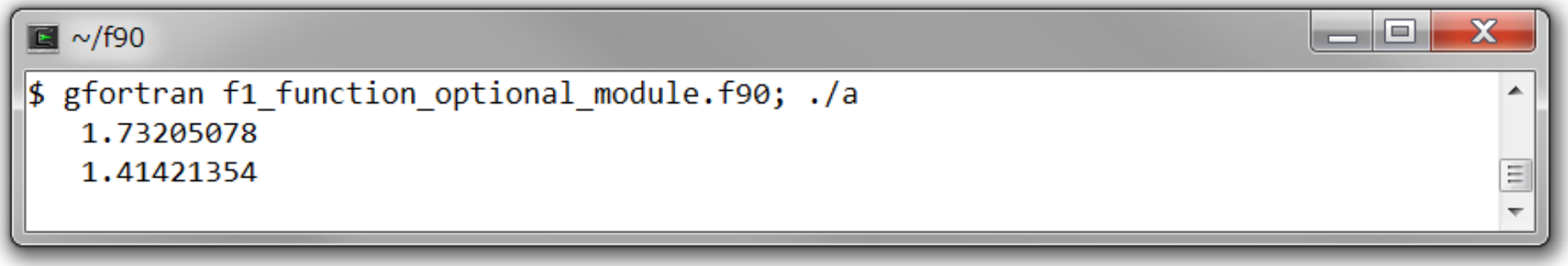

~/f90
$ gfortran f1_function_keyword_module.f90; ./a
1.41421354
1.41421354
1.41421354

■OPTIONAL 属性に関するモジュール関数副プログラム例　実引数の一部を省略した場合（f1_function_optional_module.f90).

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  function ofunc(x,y,z) ! 関数副プログラム
    real, intent(IN) :: x,y
    real, intent(IN), optional :: z ! optional 属性
    real :: ofunc
    if(present(z)) then ! present 組込関数
       ofunc = sqrt(x**2+y**2+z**2)
    else
       ofunc = sqrt(x**2+y**2)
    endif
  end function ofunc
end module msubprog
program f1_function_optional_module ! 主プログラム
  use msubprog ! use 文
  implicit none
  print *, ofunc(1.0,1.0,1.0) ! 実引数を省略していない場合
  print *, ofunc(1.0,1.0) ! 実引数を一部省略した場合
  stop
```

```
end program f1_function_optional_module
```

コンパイル，リンク，実行すると，

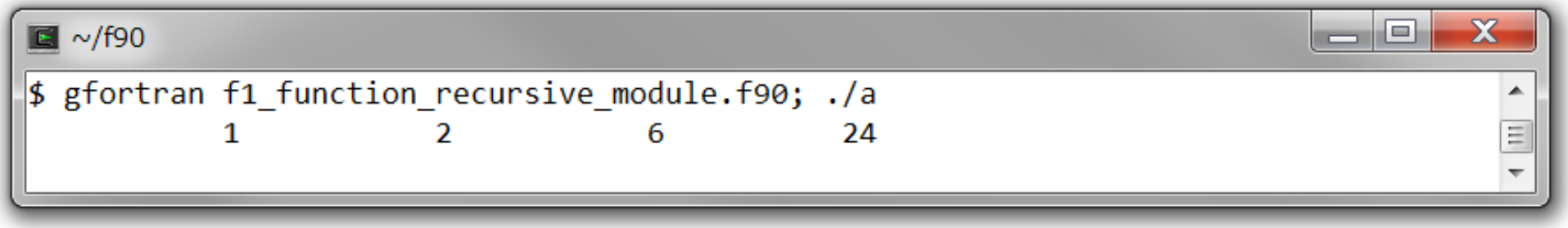


■RECURSIVE FUNCTION に関するモジュール関数副プログラム例　再帰コールによる階乗計算（f1_function_recursive_module.f90).

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  recursive function rfunc(n) result(k) ! recursive 接頭辞
    implicit none
    integer, intent(IN) :: n
    integer :: k
    if(n==1) then
       k = 1
    else
       k = n*rfunc(n-1)
    endif
    return
  end function rfunc
end module msubprog
program f1_function_recursive_module ! 主プログラム
  use msubprog ! use 文
  implicit none
  Print *, rfunc(1),rfunc(2),rfunc(3),rfunc(4)
  stop
```

```
end program f1_function_recursive_module
```

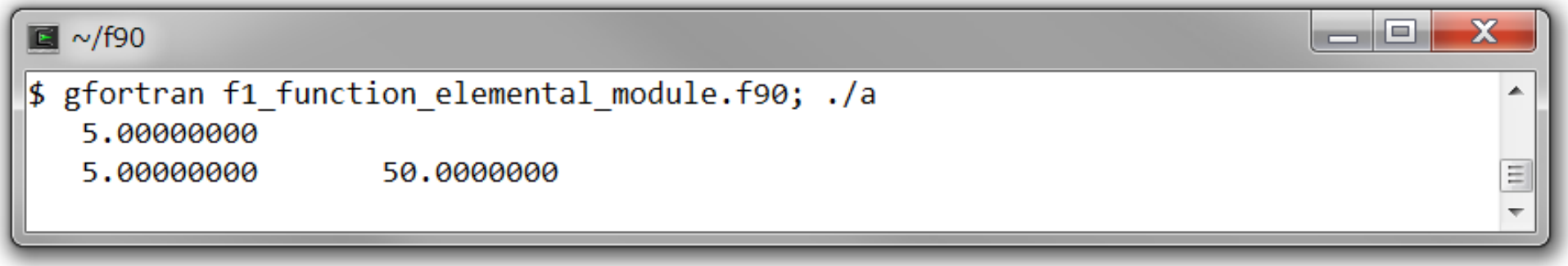
コンパイル，リンク，実行すると，



■ELEMENTAL FUNCTION に関するモジュール関数副プログラム例　実引数および戻り値が，スカラと 1 次元配列の場合（f1_function_elemental_module.f90）.

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  elemental function efunc(x,y) result(z) ! elemental 接頭辞
    real, intent(IN) :: x,y
    real :: z
    z = x**2 + y**2
    z = sqrt(z)
  end function efunc
end module msubprog

program f1_function_elemental_module ! 主プログラム
  use msubprog ! use 文
  implicit none
  print *, efunc(3.0, 4.0)
  print *, efunc( (/3.0, 30.0/), [4.0, 40.0] )
  stop
end program f1_function_elemental_module
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_function_elemental_module.f90; ./a
5.00000000
5.00000000 50.0000000

16.3.2　モジュールサブルーチン副プログラム

```
MODULE module-name ! モジュール
    IMPLICUT NONE
    [specification-stmts] ! 宣言文
CONTAINS
    SUBROUTINE sub-name (d-arg-1 [, d-arg-2, ... ] ) ! モジュールサブルーチン副プログラム
        TYPE-a  d-arg-1 [, d-arg-2, ... ]
       .....
    END [SUBROUTINE [sub-name] ]
END [MODULE [module-name] ]

PROGRAM main-name ! 主プログラム
    USE module-name
    IMPLICUT NONE
   .....
    CALL sub-name (a-arg-1 [, a-arg-2, ... ] )
   .....
    STOP
END [PROGRAM [main-name] ]
```

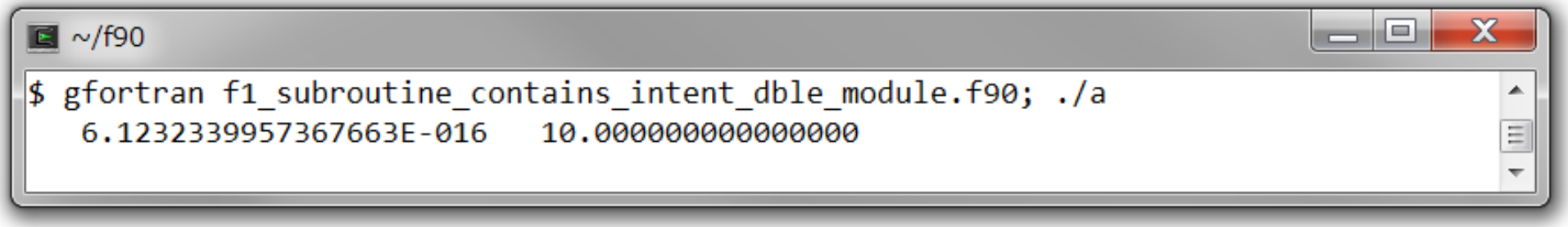
（互換性：Fortran 90〜）

■INTENT 属性を用いた倍精度のモジュールサブルーチン副プログラム例

(f1_subroutine_contains_intent_dble_module.f90)

```
module msubprog ! モジュール副プログラム
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
contains
  subroutine dsub2(r,phi, x,y) ! サブルーチン副プログラム
    real(8), intent(IN) :: r,phi ! intent 属性
    real(8), intent(OUT) :: x,y ! intent 属性
    x = r*cos(phi)
    y = r*sin(phi)
    return
  end subroutine dsub2
end module msubprog

program f1_subroutine_contains_intent_dble_module ! 主プログラム
  use msubprog ! use 文
  implicit none
  real(8) :: xx,yy
  call dsub2(10.0D0,90.0D0/q, xx,yy) ! サブルーチンの呼び出し
  print *,xx,yy
  stop
end program f1_subroutine_contains_intent_dble_module
```

コンパイル，リンク，実行すると，

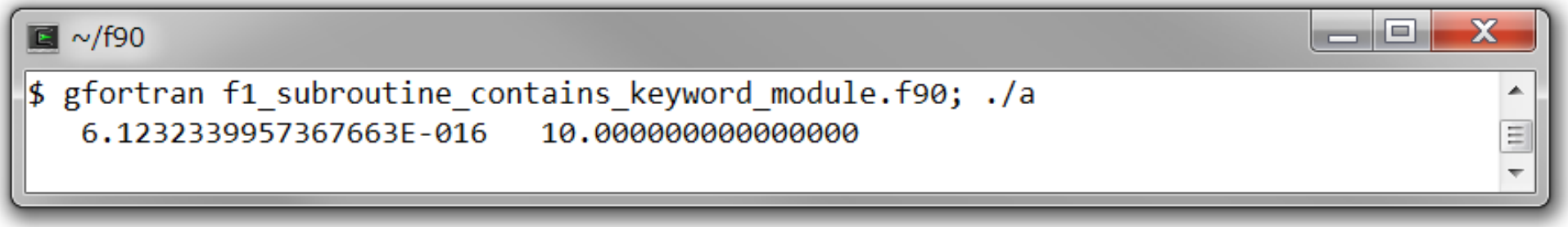


■引数キーワードを指定したモジュールサブルーチン副プログラム例（f1_subroutine_contains_keyword_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
contains
  subroutine dsub2(radius,angle_phi, rect_x,rect_y) ! サブルーチン副プログラム
    real(8), intent(IN) :: radius,angle_phi
    real(8), intent(OUT) :: rect_x,rect_y
    rect_x = radius*cos(angle_phi)
    rect_y = radius*sin(angle_phi)
    return
  end subroutine dsub2
end module msubprog
program f1_subroutine_contains_keyword_module ! 主プログラム（倍精度）
  use msubprog ! use文
  implicit none
  real(8) :: x,y
  call dsub2(rect_x=x, rect_y=y, &
       radius=10.0D0, angle_phi=90.0D0/q) ! 引数キーワード指定
  print *,x,y
  stop
```

```
end program f1_subroutine_contains_keyword_module
```

コンパイル，リンク，実行すると，

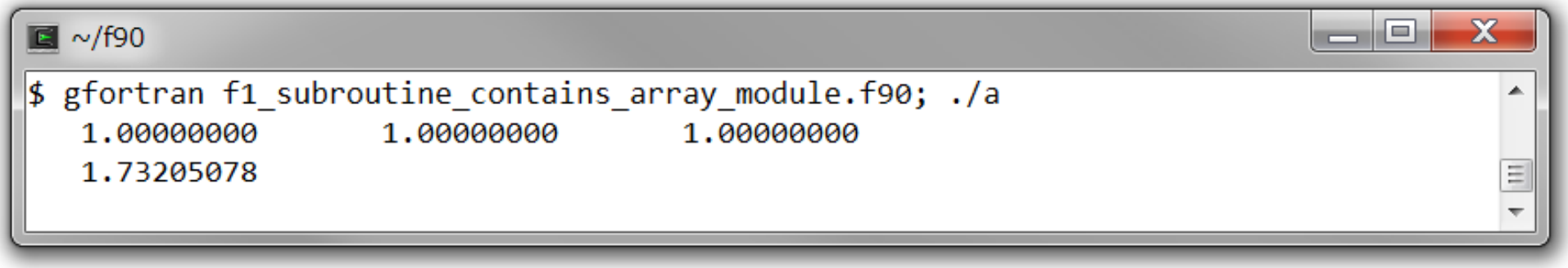


■形状明示配列（explicit-shape array）を引数とするモジュールサブルーチン副プログラム例 （f1_subroutine_contains_array_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none
  integer, parameter :: nn = 11
contains
  subroutine suba(x,n, a) ! サブルーチン副プログラム
    integer, intent(IN) :: n
    real, intent(IN) :: x(nn)
    real, intent(OUT) :: a
    integer :: i
    a = 0.0
    do i=1,n
       a = a+x(i)**2
    enddo
    a = sqrt(a)
    return
  end subroutine suba
end module msubprog

program f1_subroutine_contains_array_module ! 主プログラム
  use msubprog ! use 文
  implicit none
```

```
  real :: xx(nn),aa
  xx(1:3) = [1.0, 1.0, 1.0]
  print *,xx(1:3)
  call suba(xx,3, aa)
  print *,aa
  stop
end program f1_subroutine_contains_array_module
```

コンパイル，リンク，実行すると，

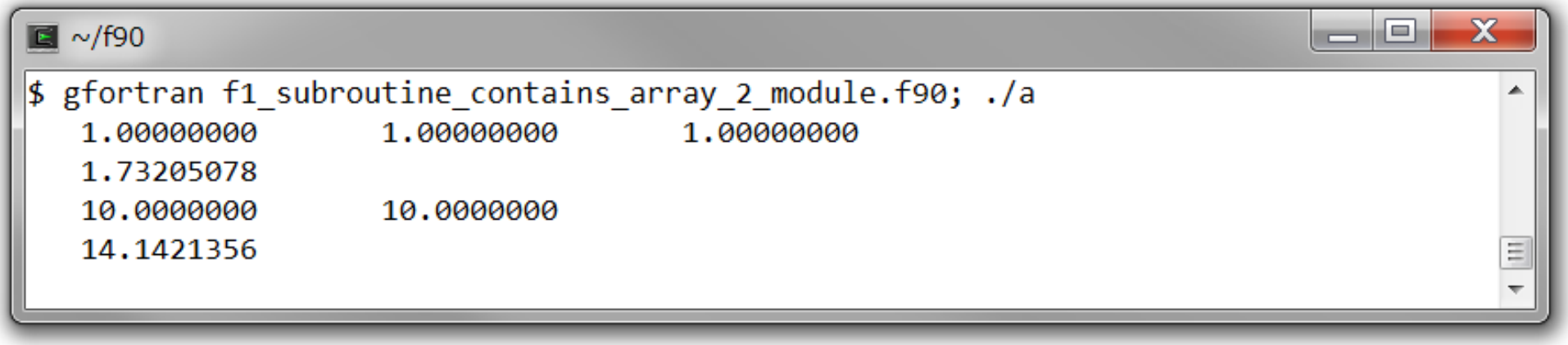


■形状引継配列（assumed-shape array）を引数とするモジュールサブルーチン副プログラム例 （f1_subroutine_contains_array_2_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none
  integer, parameter :: nn = 11, mm = 5
contains
  subroutine suba(x,n, a) ! サブルーチン副プログラム
    integer, intent(IN) :: n
    real, intent(IN) :: x(:)
    real, intent(OUT) :: a
    integer :: i
    a = 0.0
    do i=1,n
       a = a+x(i)**2
    enddo
    a = sqrt(a)
    return
  end subroutine suba
end module msubprog

program f1_subroutine_contains_array_2_module ! 主プログラム
  use msubprog ! use文
  implicit none
```

```
  real :: xx(nn),yy(mm),aa,bb
  xx(1:3) = [1.0, 1.0, 1.0]
  print *,xx(1:3)
  call suba(xx,3, aa)
  print *,aa
  yy(1:2) = [10.0, 10.0]
  print *,yy(1:2)
  call suba(yy,2, bb)
  print *,bb
  stop
end program f1_subroutine_contains_array_2_module
```

コンパイル，リンク，実行すると，

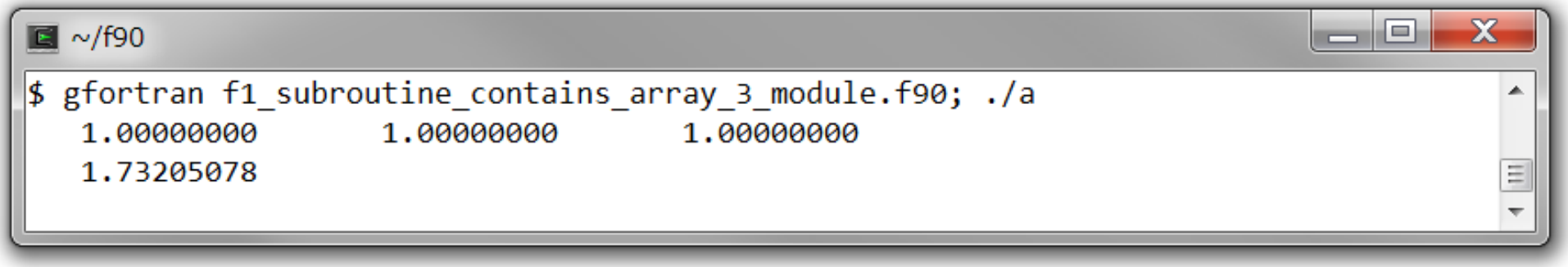


■動的割り付けした配列を引数とするモジュールサブルーチン副プログラム例（f1_subroutine_contains_array_3_module.f90）

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  subroutine suba(x, a) ! サブルーチン副プログラム
    real, intent(IN) :: x(:)
    real, intent(OUT) :: a
    integer :: i,n
    a = 0.0
    n = size(x) ! 配列の問い合わせ関数
    do i=1,n
       a = a+x(i)**2
    enddo
    a = sqrt(a)
    return
  end subroutine suba
end module msubprog

program f1_subroutine_contains_array_3_module ! 主プログラム
  use msubprog ! use文
  implicit none
  integer :: nn = 3, is
```

```
  real, allocatable :: xx(:) ! 形状無指定配列 (deferred-shape array)
  real :: aa
  allocate(xx(nn),stat=is) ! 配列の割り付け
  if(is/=0) stop 'cannot allocate'
  xx(1:nn) = [1.0, 1.0, 1.0]
  print *,xx(1:nn)
  call suba(xx, aa)
  print *,aa
  deallocate(xx) ! 割り付けの解除
  stop
end program f1_subroutine_contains_array_3_module
```

コンパイル，リンク，実行すると，



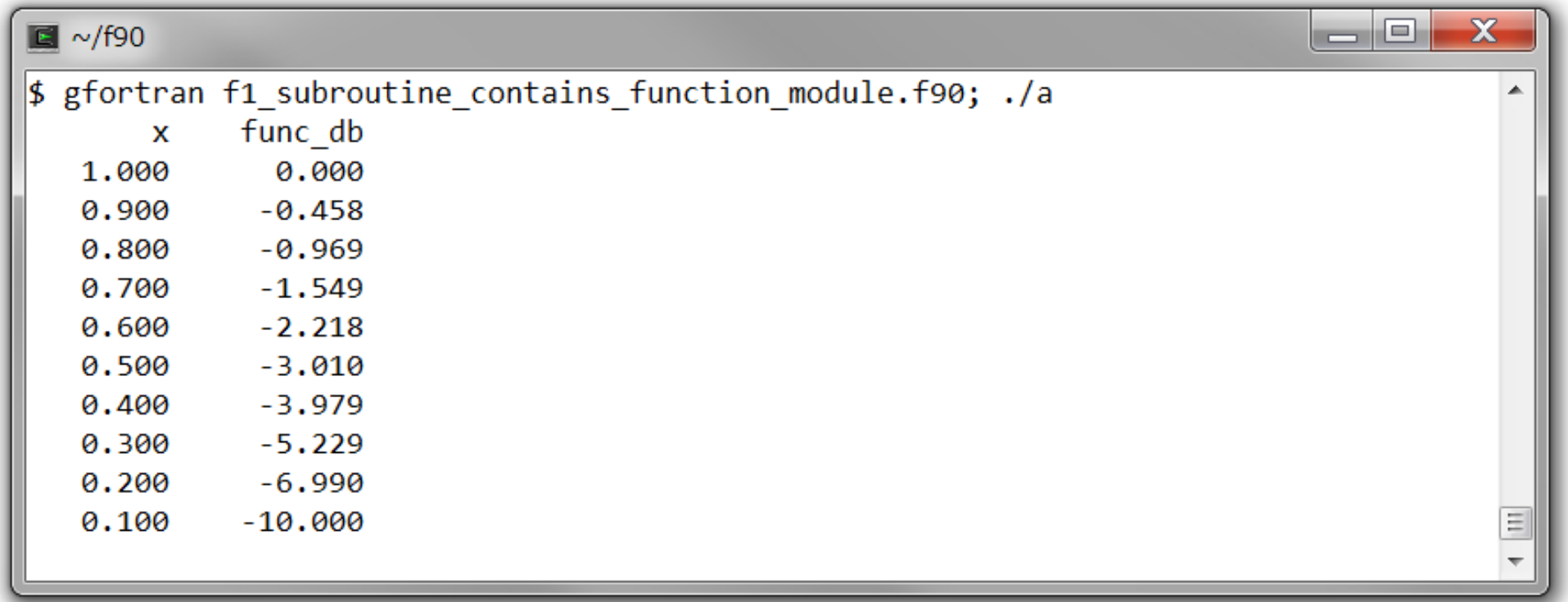
■スカラ関数副プログラムを引数とするサブルーチン副プログラムに関する倍精度モジュールサブルーチン副プログラム例 1

(f1_subroutine_contains_function_module.f90)

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  subroutine sub_f(func, x0,dx,nx) ! サブルーチン副プログラム
    real(8), intent(IN) :: x0,dx
    integer, intent(IN) :: nx
    integer :: i
    real(8) :: func,x
    external func
    print '(a8,1x,a10)','x','func_db'
    do i=1,nx
       x = x0+dx*(i-1)
       print '(f8.3,1x,f10.3)',x,func(x)
    enddo
    return
  end subroutine sub_f

  real(8) function func_db(x) result(y) ! 関数副プログラム
    real(8), intent(IN) :: x
```

```
    y = abs(x)
    if(y/=0.0D0) then
       y = 10.0D0*log10(y)
    else
       y = -800.0D0
    endif
    return
  end function func_db
end module msubprog

program f1_subroutine_contains_function_module ! 主プログラム
  use msubprog ! use文
  implicit none
  call sub_f(func_db, 1.0D0, -0.1D0, 10)
  stop
end program f1_subroutine_contains_function_module
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_subroutine_contains_function_module.f90; ./a
x func_db
1.000 0.000
0.900 -0.458
0.800 -0.969
0.700 -1.549
0.600 -2.218
0.500 -3.010
0.400 -3.979
0.300 -5.229
0.200 -6.990
0.100 -10.000

■スカラ関数副プログラムを引数とするサブルーチン副プログラムに関する倍精度モジュールサブルーチン副プログラム例 2

(f1_subroutine_contains_function_optional_module.f90)

```
module msubprog ! モジュール副プログラム
  implicit none
contains
  subroutine sub_f(func, nx,xmin,delx) ! サブルーチン副プログラム
    integer, intent(IN) :: nx
    real(8), optional :: xmin,delx ! optional 属性
    real(8) :: func
    integer :: i
    real(8) :: x,x0,dx
    external func
    if(present(xmin)) then ! present 関数
       x0 = xmin
    else
       x0 = 1.0D0
    endif
    if(present(delx)) then ! present 関数
       dx = delx
    else
       dx = -x0/(nx-1)
```

```
    endif
    print *,'x0=',x0,'dx=',dx
    print '(a8,1x,a10)','x','func_db'
    do i=1,nx
       x = x0+dx*(i-1)
       print '(f8.3,1x,f10.3)',x,func(x)
    enddo
    return
  end subroutine sub_f

  real(8) function func_db(x) result(y) ! 関数副プログラム
    real(8), intent(IN) :: x
    y = abs(x)
    if(y/=0.0D0) then
       y = 10.0D0*log10(y)
    else
       y = -800.0D0
    endif
    return
  end function func_db
end module msubprog

program f1_subroutine_contains_function_optional_module ! 主プログラム
```

```
  use msubprog ! use文
  implicit none
  call sub_f(func_db, 5, 1.0D0, -0.5D0)
  call sub_f(func=func_db, nx=5,xmin=1.0D0,delx=-0.05D0)
  call sub_f(func_db, 5, xmin=1.0D0)
  call sub_f(func_db, 3)
  stop
end program f1_subroutine_contains_function_optional_module
```

コンパイル，リンク，実行すると，

~/f90

```
$ gfortran f1_subroutine_contains_function_optional_module.f90; ./a
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000      0.000
   0.500     -3.010
   0.000   -800.000
  -0.500     -3.010
  -1.000      0.000
 x0=   1.0000000000000000      dx=  -5.0000000000000003E-002
       x    func_db
   1.000      0.000
   0.950     -0.223
   0.900     -0.458
   0.850     -0.706
   0.800     -0.969
 x0=   1.0000000000000000      dx= -0.25000000000000000
       x    func_db
   1.000      0.000
   0.750     -1.249
   0.500     -3.010
   0.250     -6.021
   0.000   -800.000
 x0=   1.0000000000000000      dx= -0.50000000000000000
       x    func_db
   1.000      0.000
   0.500     -3.010
   0.000   -800.000
```

■ELEMENTAL 接頭辞を用いた倍精度のモジュールサブルーチン副プログラム例

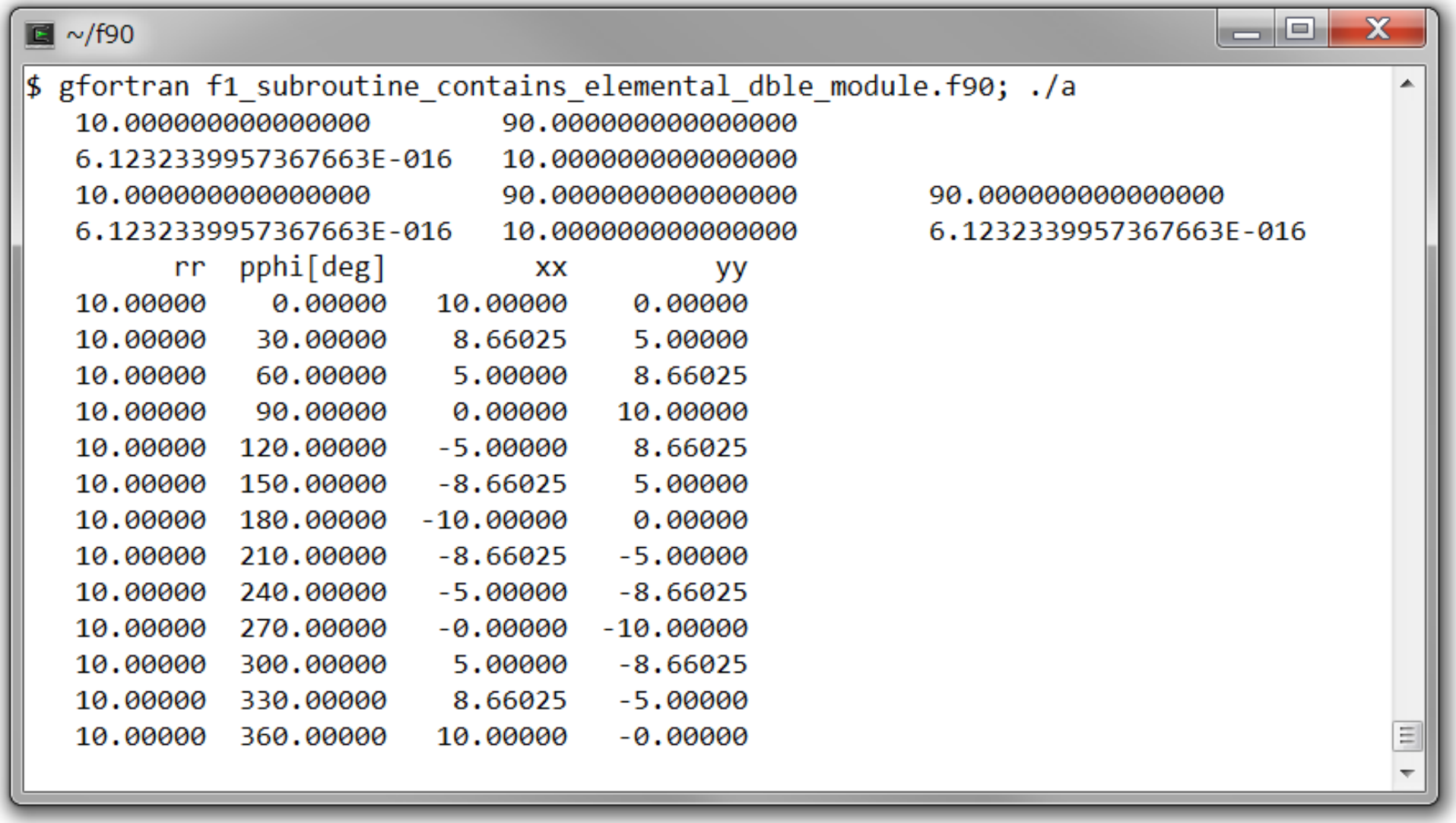
(f1_subroutine_contains_elemental_dble_module.f90)

```
module msubprog ! モジュール副プログラム
  implicit none
  real(8), parameter :: pi = 4.0D0*atan(1.0D0), q = 180.0D0/pi
  integer, parameter :: nn = 101
contains
  elemental subroutine esub2(radius,angle_theta,angle_phi, &
       rect_x,rect_y,rect_z) ! ELEMENTAL SUBROUTINE
    real(8), intent(IN) :: radius,angle_phi
    real(8), intent(OUT) :: rect_x,rect_y
    real(8), optional, intent(IN) :: angle_theta ! optional 属性
    real(8), optional, intent(OUT) :: rect_z ! optional 属性
    real(8) :: ct,st,cp,sp
    cp = cos(angle_phi)
    sp = sin(angle_phi)
    ct = 0.0D0
    st = 1.0D0
    if(present(angle_theta)) then ! present 関数
       ct = cos(angle_theta)
       st = sin(angle_theta)
    endif
    rect_x = radius*st*cp
```

```
    rect_y = radius*st*sp
    if(present(rect_z)) rect_z = radius*ct ! present 関数
    return
  end subroutine esub2
end module msubprog

program f1_subroutine_contains_elemental_dble_module ! 主プログラム（倍精度）
  use msubprog ! use 文
  implicit none
  real(8) :: x,y,r,phi
  real(8) :: z,th
  real(8) :: xx(nn),yy(nn),zz(nn),rr(nn),tth(nn),pphi(nn)
  real(8) :: phi0 = 0.0D0, dphi = 30.0D0/q
  integer :: i, nphi = 13

  r = 10.0D0
  phi = 90.0D0/q
  print *,r,phi*q
  call esub2(rect_x=x, rect_y=y, radius=r, angle_phi=phi) ! 引数はスカラー
  print *,x,y

  th = 90.0D0/q
  print *,r,th*q,phi*q
```

```
  call esub2(rect_x=x, rect_y=y, rect_z=z, &
       radius=r, angle_phi=phi, angle_theta=th) ! 引数はスカラー
  print *,x,y,z

  rr(1:nphi) = 10.0D0
  pphi(1:nphi) = [ (phi0+dphi*(i-1),i=1,nphi) ]
  call esub2(rect_x=xx, rect_y=yy, radius=rr, angle_phi=pphi) ! 引数は配列
  print '(4a11)','rr','pphi[deg]','xx','yy'
  do i=1,nphi
     print '(4(1x,f10.5))',rr(i),pphi(i)*q,xx(i),yy(i)
  enddo
  stop
end program f1_subroutine_contains_elemental_dble_module
```

コンパイル，リンク，実行すると，


~/f90
$ gfortran f1_subroutine_contains_elemental_dble_module.f90; ./a
10.000000000000000 90.000000000000000
6.1232339957367663E-016 10.000000000000000
10.000000000000000 90.000000000000000 90.000000000000000
6.1232339957367663E-016 10.000000000000000 6.1232339957367663E-016
rr pphi[deg] xx yy
10.00000 0.00000 10.00000 0.00000
10.00000 30.00000 8.66025 5.00000
10.00000 60.00000 5.00000 8.66025
10.00000 90.00000 0.00000 10.00000
10.00000 120.00000 -5.00000 8.66025
10.00000 150.00000 -8.66025 5.00000
10.00000 180.00000 -10.00000 0.00000
10.00000 210.00000 -8.66025 -5.00000
10.00000 240.00000 -5.00000 -8.66025
10.00000 270.00000 -0.00000 -10.00000
10.00000 300.00000 5.00000 -8.66025
10.00000 330.00000 8.66025 -5.00000
10.00000 360.00000 10.00000 -0.00000

16.4　モジュールによる変数の共有等

16.4.1　ONLY 句

一部の変数に制限を設ける場合,

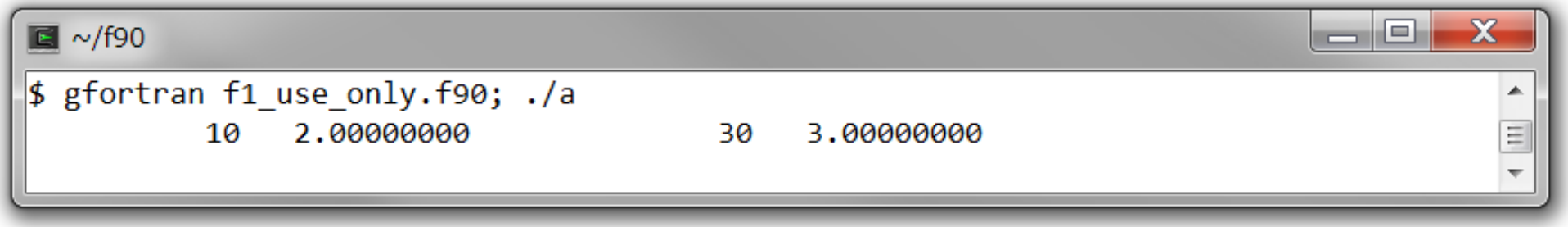
USE *module-name*, ONLY : *[local-name* => *] mod-name*

（互換性：Fortran 90〜）

■ONLY 句に関するプログラム例　　（f1_use_only.f90）

```
module set ! モジュール
  real :: a = 1.0, b = 2.0, c = 3.0
end module

program f1_use_only ! 主プログラム
  use set, only: b, x=>c ! use 文, only 句
!  implicit none
  integer :: a = 10, c = 30
  print *,a,b,c,x
  stop
end program f1_use_only
```

コンパイル，リンク，実行すると，



16.4.2　グローバル変数モジュール

変数（あるいは定数）を SAVE 属性をつけて宣言（あるいは初期化）し，参照するためだけのモジュールをグローバル変数モジュールという．

```
MODULE module-name
    [specification-stmts] ！定数（parameter 属性），変数（save 属性）の宣言
END [MODULE [module-name] ]
```

（互換性：Fortran 90〜）

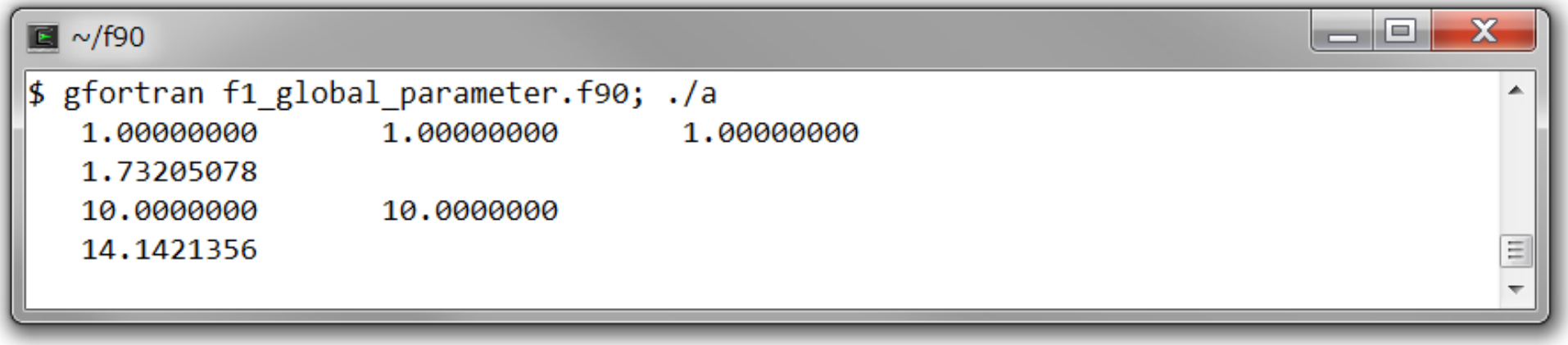
変数を参照するためのモジュールは，COMMON の代用となる．このように宣言文のみでモジュールを構成すれば，IMPLICIT NONE は不要である．

■グローバル変数（パラメータ属性）モジュールに関するプログラム例（f1_global_parameter.f90）

```
module parameter_mod !  （グローバル変数）モジュール
  integer, parameter :: nn = 11, mm = 5
end module parameter_mod
module msubprog ! モジュール（副プログラム）
  implicit none
contains
  subroutine suba(x,n, a) ! サブルーチン副プログラム
    integer, intent(IN) :: n
    real, intent(IN) :: x(:) ! 形状引継配列（assumed-shape array）
    real, intent(OUT) :: a
    integer :: i
    a = 0.0
    do i=1,n
       a = a+x(i)**2
    enddo
    a = sqrt(a)
    return
  end subroutine suba
end module msubprog

program f1_global_parameter ! 主プログラム
```

```
  use parameter_mod ! パラメータの参照
  use msubprog ! use 文
  implicit none
  real :: xx(nn),yy(mm),aa,bb
  xx(1:3) = [1.0, 1.0, 1.0]
  print *,xx(1:3)
  call suba(xx,3, aa)
  print *,aa
  yy(1:2) = [10.0, 10.0]
  print *,yy(1:2)
  call suba(yy,2, bb)
  print *,bb
  stop
end program f1_global_parameter
```

コンパイル，リンク，実行すると，

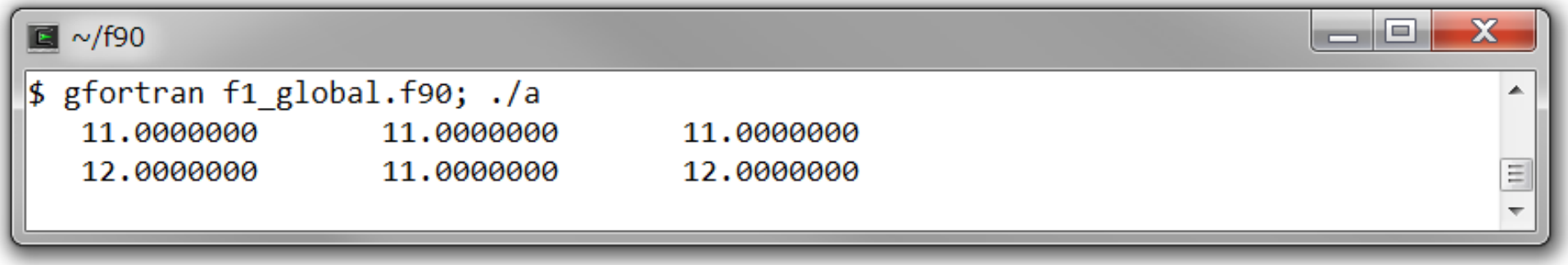


■グローバル変数モジュール（COMMON の代用）に関するプログラム例（f1_global.f90）

```
module global_mod ! （グローバル変数）モジュール
  real, save :: a=10.0
  real, save :: b
end module global_mod

module msubprog ! モジュール（副プログラム）
!  implicit none
contains
  real function func() ! 内部関数副プログラム
    use global_mod, only: a ! グローバル変数 a の参照
    implicit none
    a = a+1.0
    func = a
  end function func
  real function funca(a) ! 内部関数副プログラム
    implicit none
    real, intent(IN) :: a
    funca = a+1.0
  end function funca
  real function funcb() ! 内部関数副プログラム
    use global_mod, only: b ! グローバル変数 b の参照
```

```
    implicit none
    b = b+1.0
    funcb = b
  end function funcb
end module msubprog

program f1_global ! 主プログラム
  use global_mod, only: b ! グローバル変数 b の参照
  use msubprog ! use 文
  implicit none
  b = 10.0
  print *, func(),funca(10.0),funcb()
  print *, func(),funca(10.0),funcb()
  stop
end program f1_global
```

コンパイル，リンク，実行すると，



16.4.3 PRIVATE 指定子

参照を許可しない設定（非公開）とする指定子で，PRIVATE 属性と PRIVATE 文があり，これを指定したモジュール以外からの参照できなくなる．PRIVATE 属性では，

```
TYPE, PRIVATE :: var-list
```

（互換性：Fortran 90〜）

- *TYPE*：変数の型．
- *var-list*：変数名，配列名の並び．

また，PRIVATE 文では，サブルーチン等を指定できる．

```
PRIVATE procedure-list
```

（互換性：Fortran 90〜）

- *procedure-list*：サブルーチン等の手続き名の並び．

すべて非公開にする場合は，

```
PRIVATE
```

16.4.4 PUBLIC 指定子

参照を許可する設定（公開）とする指定子で，PUBLIC 属性と PUBLIC 文があり，これを指定したモジュール以外から参照できる．PUBLIC 属性では，

TYPE, PUBLIC :: *var-list*

（互換性：Fortran 90〜）

- *TYPE*：変数の型．
- *var-list*：変数名，配列名の並び．

また，PUBLIC 文では，サブルーチン等を指定できる．

PUBLIC *procedure-list*

（互換性：Fortran 90〜）

- *procedure-list*：サブルーチン等の手続き名の並び．

■PRIVATE 文，PUBLIC 文に関するプログラム例　（f1_private.f90）

```
module steps ! モジュール
  implicit none
  private ! private 文
  integer :: ii = 2
  public :: sub,func ! public 文
contains
  integer function func()
    func = ii ! 同じプログラム単位内ではアクセス可能
  end function func
  subroutine sub
    ii = ii + 10 ! 同上でアクセス可能
  end subroutine sub
end module steps

program f1_private
  use steps ! use 文
  implicit none
!  print *,ii ! priave 属性ゆえ，use 文があってもアクセス不可能
  print *, func()
  call sub
  print *, func()
end program f1_private
```

コンパイル，リンク，実行すると，

16.5 GENERIC SUBPROGRAM（総称名副プログラム）

16.5.1 外部副プログラムを用いた総称名副プログラム

モジュールインターフェースにおいて，総称名をインターフェースブロックで引用仕様宣言する場合，

```
MODULE module-name ! モジュール
    INTERFACE generic-name ! 総称名インターフェースブロック
          .....
    END [INTERFACE [generic-name] ]
END [MODULE [module-name] ]

PROGRAM main-name ! 主プログラム
    USE module-name
    IMPLICUT NONE
    ..... ! 総称名副プログラム generic-name の呼び出し .....
    STOP
END [PROGRAM [main-name] ]

..... ! 複数の外部副プログラム subprog-name1, subprog-name2, .....
```

（互換性：Fortran 90〜）

■1 次元配列（整数型，実数型，複素数型）を出力する外部副プログラムを用いた総称名副プログラム例　（f1_generic_external.f90）

```
module ginterface_mod ! モジュール（総称名副プログラム）
  interface print_vec ! 総称名インターフェースブロック
     subroutine print_vec_int(ii,nmin,nmax)
       integer, intent(IN) :: ii(:), nmin, nmax
     end subroutine print_vec_int
     subroutine print_vec_real(aa,nmin,nmax)
       real, intent(IN) :: aa(:)
       integer, intent(IN) :: nmin, nmax
     end subroutine print_vec_real
     subroutine print_vec_complex(zz,nmin,nmax)
       complex, intent(IN) :: zz(:)
       integer, intent(IN) :: nmin, nmax
     end subroutine print_vec_complex
  end interface print_vec
end module ginterface_mod

program f1_generic_external ! 主プログラム
  use ginterface_mod ! use 文
  implicit none
  integer, parameter :: nn=10
  integer :: ii(nn), nmin=2, nmax=5, i
```

```
  real :: aa(nn)
  complex :: zz(nn)
  ii(:) = [ (i*3,i=1,nn) ]
!  call print_vec_int(ii,nmin,nmax)
  call print_vec(ii,nmin,nmax)
  print *,
  aa(:) = ii(:)*10.0
!  call print_vec_real(aa,nmin,nmax)
  call print_vec(aa,nmin,nmax)
  print *,
  zz(:) = aa(:)*(0.0,10.0)
!  call print_vec_complex(zz,nmin,nmax)
  call print_vec(zz,nmin,nmax)
  stop
end program f1_generic_external

subroutine print_vec_int(ii,nmin,nmax) ! 整数型用副プログラム
  implicit none
  integer, intent(IN) :: ii(:), nmin, nmax
  integer i
  print *,'lbound=',lbound(ii),', ubound=',ubound(ii)
  do i=nmin,nmax
     print *, i,ii(i)
```

```
  enddo
end subroutine print_vec_int
subroutine print_vec_real(aa,nmin,nmax) ! 実数型用副プログラム
  implicit none
  real, intent(IN) :: aa(:)
  integer, intent(IN) :: nmin, nmax
  integer i
  print *,'lbound=',lbound(aa),', ubound=',ubound(aa)
  do i=nmin,nmax
     print *, i,aa(i)
  enddo
end subroutine print_vec_real
subroutine print_vec_complex(zz,nmin,nmax) ! 複素数型用副プログラム
  implicit none
  complex, intent(IN) :: zz(:)
  integer, intent(IN) :: nmin, nmax
  integer i
  print *,'lbound=',lbound(zz),', ubound=',ubound(zz)
  do i=nmin,nmax
     print *, i,zz(i)
  enddo
end subroutine print_vec_complex
```

コンパイル，リンク，実行すると，

~/f90

```
$ gfortran f1_generic_external.f90; ./a
 lbound=           1 , ubound=          10
           2           6
           3           9
           4          12
           5          15

 lbound=           1 , ubound=          10
           2   60.0000000
           3   90.0000000
           4   120.000000
           5   150.000000

 lbound=           1 , ubound=          10
           2 (  0.00000000    ,  600.000000    )
           3 (  0.00000000    ,  900.000000    )
           4 (  0.00000000    ,  1200.00000    )
           5 (  0.00000000    ,  1500.00000    )
```

■データ（整数型，実数型）を入出力する外部副プログラムを用いた総称名副プログラム例　（f1_generic_2_external.f90）

```
module ginterface_mod ! モジュール（総称名副プログラム）
  interface iodata ! 総称名インターフェースブロック
     subroutine iodata_int(title, ivar)
       character(*), intent(OUT) :: title
       integer, intent(OUT) :: ivar
     end subroutine iodata_int
     subroutine iodata_int2(title, ivar1, ivar2)
       character(*), intent(OUT) :: title
       integer, intent(OUT) :: ivar1, ivar2
     end subroutine iodata_int2
     subroutine iodata_dble(title, dvar)
       character(*), intent(OUT) :: title
       real(8), intent(OUT) :: dvar
     end subroutine iodata_dble
  end interface iodata
end module ginterface_mod

program f1_generic_2_external ! 主プログラム
  use ginterface_mod ! use文
  implicit none
  character(10) :: title1, title2, title3
```

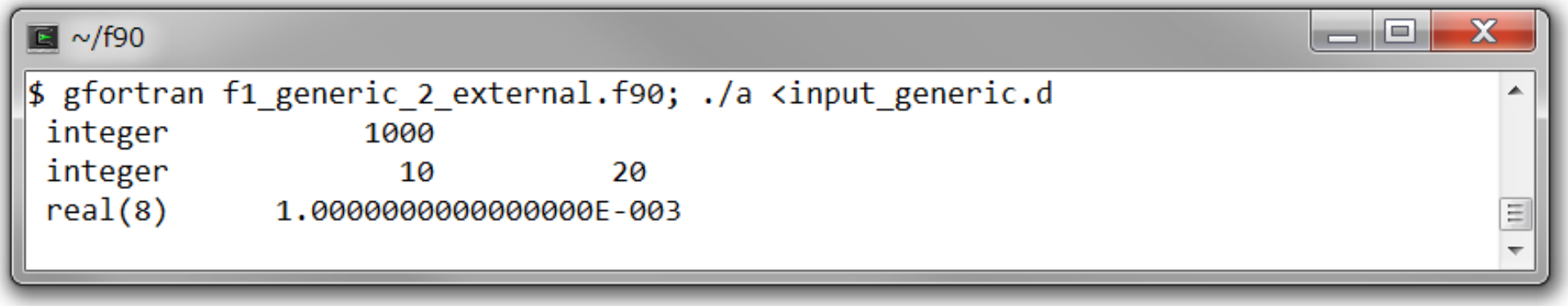
```
  integer :: n1, n2, n3
  real(8) a
  call iodata(title1, n1) ! 整数型の引数
  call iodata(title2, n2, n3) ! 整数型２変数の引数
  call iodata(title3, a) ! 倍精度実数型の引数
  stop
end program f1_generic_2_external

subroutine iodata_int(title, ivar) ! 整数型用副プログラム
  implicit none
  character(*), intent(OUT) :: title
  integer, intent(OUT) :: ivar
  read *, title, ivar
  print *, title, ivar
end subroutine iodata_int
subroutine iodata_int2(title, ivar1, ivar2) ! 整数型２変数用副プログラム
  implicit none
  character(*), intent(OUT) :: title
  integer, intent(OUT) :: ivar1, ivar2
  read *, title, ivar1, ivar2
  print *, title, ivar1, ivar2
end subroutine iodata_int2
subroutine iodata_dble(title, dvar) ! 倍精度実数型用副プログラム
```

```
  implicit none
  character(*), intent(OUT) :: title
  real(8), intent(OUT) :: dvar
  read *, title, dvar
  print *, title, dvar
end subroutine iodata_dble
```

入力データ input_generic.d は,

```
integer 1000
integer 10 20
real(8) 1.0D-3
```

コンパイル，リンク，実行すると,



16.5.2　内部副プログラムを用いた総称名副プログラム

モジュールにおいて，内部副プログラムを定義し，総称名を引用仕様宣言する場合，

```
MODULE module-name ! モジュール
    IMPLICUT NONE
    [specification-stmts] ! 宣言文
    INTERFACE generic-name ! 総称名インターフェースブロック
       MODULE PROCEDURE subprog-name1, subprog-name2, .....
    END [INTERFACE [generic-name] ]
CONTAINS
    ..... ! 内部副プログラム subprog-name1, subprog-name2, .....
END [MODULE [module-name] ]

PROGRAM main-name ! 主プログラム
    USE module-name
    IMPLICUT NONE
    ..... ! 総称名副プログラム generic-name の呼び出し .....
    STOP
END [PROGRAM [main-name] ]
```

（互換性：Fortran 90〜）

■1 次元配列（整数型，実数型，複素数型）を出力する内部副プログラムを用いた総称名副プログラム例　（f1_generic_internal.f90）

```
module  gsubprog ! モジュール（総称名副プログラム）
  implicit none
  interface print_vec ! 総称名インターフェースブロック
     module procedure print_vec_int, print_vec_real, print_vec_complex
  end interface print_vec
contains
  subroutine print_vec_int(ii,nmin,nmax) ! 整数型用副プログラム
    integer, intent(IN) :: ii(:), nmin, nmax
    integer i
    print *,'lbound=',lbound(ii),', ubound=',ubound(ii)
    do i=nmin,nmax
       print *, i,ii(i)
    enddo
  end subroutine print_vec_int
  subroutine print_vec_real(aa,nmin,nmax) ! 実数型用副プログラム
    real, intent(IN) :: aa(:)
    integer, intent(IN) :: nmin, nmax
    integer i
    print *,'lbound=',lbound(aa),', ubound=',ubound(aa)
    do i=nmin,nmax
       print *, i,aa(i)
```

```
    enddo
  end subroutine print_vec_real
  subroutine print_vec_complex(zz,nmin,nmax) ! 複素数型用副プログラム
    complex, intent(IN) :: zz(:)
    integer, intent(IN) :: nmin, nmax
    integer i
    print *,'lbound=',lbound(zz),', ubound=',ubound(zz)
    do i=nmin,nmax
       print *, i,zz(i)
    enddo
  end subroutine print_vec_complex
end module gsubprog

program f1_generic_internal ! 主プログラム
  use gsubprog ! use文
  implicit none
  integer, parameter :: nn=10
  integer :: ii(nn), nmin=2, nmax=5, i
  real :: aa(nn)
  complex :: zz(nn)
  ii(:) = [ (i*3,i=1,nn) ]
!  call print_vec_int(ii,nmin,nmax)
  call print_vec(ii,nmin,nmax)
```

```
  print *,
  aa(:) = ii(:)*10.0
!  call print_vec_real(aa,nmin,nmax)
  call print_vec(aa,nmin,nmax)
  print *,
  zz(:) = aa(:)*(0.0,10.0)
!  call print_vec_comples(zz,nmin,nmax)
  call print_vec(zz,nmin,nmax)
  stop
end program f1_generic_internal
```

実行結果は外部副プログラムを用いた総称名副プログラム例とまったく同じゆえ省略する.

■データ（整数型，実数型）を入出力する内部副プログラムを用いた総称名副プログラム例　（f1_generic_2_internal.f90）

```
module  gsubprog ! モジュール（総称名副プログラム）
  implicit none
  interface iodata ! 総称名インターフェースブロック
     module procedure iodata_int, iodata_int2, iodata_dble
  end interface iodata
contains
  subroutine iodata_int(title, ivar) ! 整数型用副プログラム
    character(*), intent(OUT) :: title
    integer, intent(OUT) :: ivar
    read *, title, ivar
    print *, title, ivar
  end subroutine iodata_int
  subroutine iodata_int2(title, ivar1, ivar2) ! 整数型２変数用副プログラム
    character(*), intent(OUT) :: title
    integer, intent(OUT) :: ivar1, ivar2
    read *, title, ivar1, ivar2
    print *, title, ivar1, ivar2
  end subroutine iodata_int2
  subroutine iodata_dble(title, dvar) ! 倍精度実数型用副プログラム
    character(*), intent(OUT) :: title
    real(8), intent(OUT) :: dvar
```

```
    read *, title, dvar
    print *, title, dvar
  end subroutine iodata_dble
end module gsubprog

program f1_generic_2_internal ! 主プログラム
  use gsubprog ! use 文
  implicit none
  character(10) :: title1, title2, title3
  integer :: n1, n2, n3
  real(8) a
  call iodata(title1, n1) ! 整数型の引数
  call iodata(title2, n2, n3) ! 整数型２変数の引数
  call iodata(title3, a) ! 倍精度実数型の引数
  stop
end program f1_generic_2_internal
```

実行結果は外部副プログラムを用いた総称名副プログラム例とまったく同じゆえ省略する.

付録 A　Fortran 言語の歴史（Fortran Builder ヘルプより抜粋）

■始まりから Fortran 77 まで　コンピュータ上での計算の歴史は 1954 年に IBM の John Backus 氏のチームが Fortran（最初の高級言語）を開発したときに始まりました．当時，Fortran の競合相手はアセンブリ言語であったため，効率が重要視され，最初のコンパイラも最適化コンパイラでした．この効率への取り組みは Fortran の歴史を通じて重要視されることになります．1958 年に IBM によりリリースされたこの言語の第二版（Fortran II）ではサブルーチンのコンセプトが導入されました．この言語の開発には IBM のみならず多くの他のメーカーも行うことになり，それにより実装ごとの非互換性が生じました．WATFOR や WATFIV のような IBM 以外のバージョンも広く普及しました．

1962 年には Fortran のバージョンの違いによる互換性の問題を解消するために標準化への動きが高まりました．各メーカーはそれぞれ独自の拡張を行う事ができましたが，標準に添ったユーザーコードは異なるマシンでコンパイルでき，そして実行もできました．その後 1966 年に ANSI から最初のプログラム言語規格（現在では Fortran 66 として知られている）が出版されました．

Fortran 66 はその後も 10 年間，標準の Fortran として使用されることになりますが，言語の限界への不満から 1978 年には ANSI より改編された規格が出版されました．この規

格は 1978 年に出版されたにも関わらず Fortran 77 として知られていますが，これは技術的な内容が 1977 年に決定されたためです．Fortran 77 での主な新機能は CHARACTER データ型とブロック IF (IF - THEN - ELSE - ENDIF) です．Fortran 77 は Fortran 66 の上位互換ではなく，細かい部分で互換性がありませんでした．例えば，拡張範囲 DO ループは Fortran 77 規格では完全に削除されました．

1980 年には Fortran 77 規格が変更されることなく ISO より出版されました．

■最近の Fortran 言語の規格　Fortran 82，Fortran 8x，そして Fortran 88 などのフライングと多くの論争の後に，1991 年に ISO により大きく改編された Fortran 規格が出版されました．この規格も Fortran 77 と同様，1990 年に内容が決定されていたため，Fortran 90 と呼ばれています．Fortran 90 は Fortran 77 の完全上位互換（Fortran 77 の全てが Fortran 90 に含まれている）でした．この時点で NAG はコンパイラ市場に参加することになり，Fortran 90 規格が出版された直後に世界初の Fortran 90 コンパイラをリリースしました．その後２年間，他のコンパイラメーカーも後に続きました．

Fortran 77 と Fortran 90 の間には長いギャップがあったことや，Fortran 90 がメジャーな改編であったため，最初の２年ほどのうちにいくつかの間違いが発見されました．そのため 1995 年に向けてマイナーな改編を再度行い，これらの間違いを修正することに

なりました．当初は機能追加は行わず修正のみを行う予定でしたが，HPF を推進する団体が委員会に多少の拡張は良いということを納得させることになりました．

これにより Fortran 95 が標準化されました．標準化の遅れにより標準規格の出版は 1997 年の 12 月まで行われませんでした．主な新機能は要素別処理手続，FORALL 構文，ユーザー定義型のデフォルト初期化でした．

Fortran 90 の場合と異なり，Fortran 95 は上位互換ではなく，いくつかの古い機能が削除されました．しかしながら実際には多くの Fortran 95 コンパイラで，これらの削除された機能を拡張機能として残しています．

■Fortran 2003 標準　Fortran 95 が出版される前からもメジャーな規格改編の動きはありました．この動きは Fortran 2000 となるはずでしたが，内容の決定が遅れ，Fortran 2003 となりました．

Fortran 2003 標準は 2004 年の暮れに出版されました．

主な新機能は C との相互利用とオブジェクト指向です．多くの Fortran 95 コンパイラのメーカーは現在 Fortran 2003 の機能の開発を行っていますが，多くの機能追加が必要なため，全ての Fortran 2003 の機能をサポートする前に，Fortran 2008 標準の一部の機能をサポートするメーカーも出てきています．

■Fortran 2008 標準　引き続き Fortran 標準は更新され，新しい版が 2008 年に同意され，そして 2010 年 12 月に出版されました．これには多くの新しい機能が含まれますが，Fortran 2003 の時程ではありません．主な新機能に COARRAY による分散メモリ並列プログラミングと BLOCK 構文による実行部内における局所的な宣言が可能となり，更に多くの新たな組込関数が追加されました．

参考文献

[1] 浦 昭二, “FORTRAN77 入門,” 培風館, 1982.

[2] David R. Lemmon, Joseph L. Schafer, “*Developing Statistical Software in Fortran 95*,” Springer, 2005.

[3] 牛島 省, “数値計算のための Fortran90/95 プログラミング入門,” 森北出版, 2007.

[4] Stephen Chapman, “*Fortran 95/2003 for Scientists & Engineers*,” McGraw-Hill, 2007.

[5] 冨田 博之, 齋藤 泰洋, “Fortran90/95 プログラミング,” 培風館, 2011.

[6] Michael Metcalf, John Reid, Malcolm Cohen, “*Modern Fortran Explained*,” Oxford, 4th Revised, 2011.

[7] “*NAG Fortran Builder* ヘルプ,” The Numerical Algorithms Group Limited, 2012.

[8] “インテル *Fortran* コンパイラー *XE 13.0* ユーザー・リファレンス・ガイド,” Intel Corporation, 2013.

[9] “*GNU Fortran Manual – Using GNU Fortran*,” Free Software Foundation, 2013.