はじめてのFortran

河合 直聡
東北大学サイバーサイエンスセンター

Fortranの特長

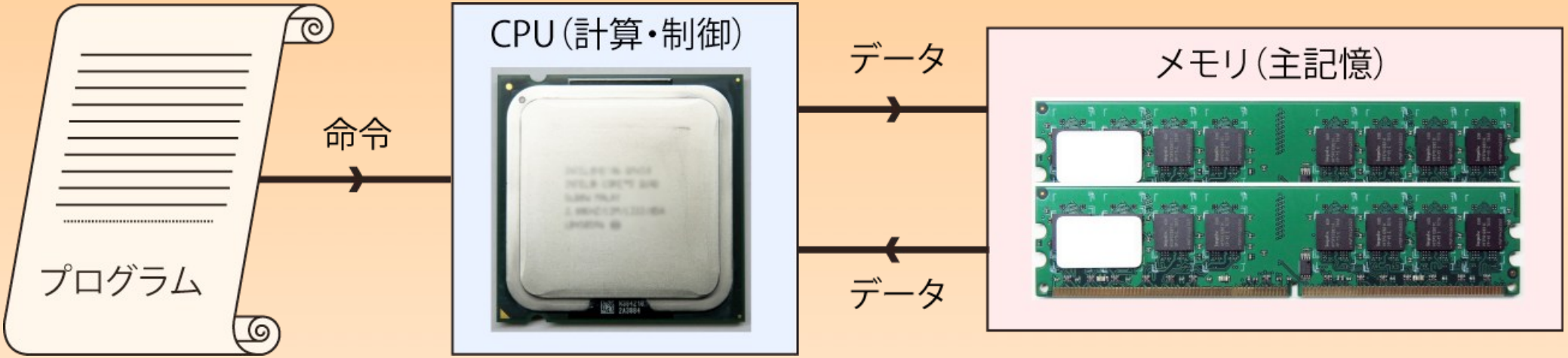
- 数値計算用の言語として古くから存在する
- 多くのスーパーコンピュータで利用可能
- 数値計算用として便利な書式が用意されている
- 定期的に仕様が更新され，オブジェクト指向や並列計算の概念をも取り入れている
- 比較的バグを見つけやすい
- やや柔軟性に欠ける(Cに比べて)
- 文字列を扱いづらい

Fortranの歴史

- FORmula　TRANSslation（数式変換）が語源
- 1956年に最初のマニュアルリリース
- 1957年に最初のコンパイラ開発
- FORTRAN IV　　（私が最初に学んだバージョン）
- FORTRAN 77　　（構造化プログラミングが可能に）

- Fortran 90　　（大幅に改訂，プログラムが楽になった）
- Fortran 95　　（マイナーチェンジ）→　今回の学習レベル
- Fortran 2003（オブジェクト指向導入）
- Fortran 2008（並列プログラミングが可能に）

計算機のしくみとプログラムとの対応



- 「プログラム」を逐次読み込んで実行する
- 計算と制御をするのが「CPU（Central Processing Unit）」
- データを一時保存するのが「メモリ（主記憶装置）でプログラム上では「変数」
- 周辺機器（キーボード，ディスプレイ，ハードディスクなど）とのやりとりは，「読み書き」

計算機でプログラムを実行するまでの流れ

プログラム

コンパイル（高級言語を機械語に翻訳）

機械語

リンク（ライブラリとの結合）

実行形式

実行

プログラムの一般的書式

動作指示語 + 動作制御パラメータ

```
real x,y,z
do i = 1, 100
print *,' x =  ',x(i),y(i)
if (x > 5) y = x**2
```

動作指示語のみ

```
stop
return
exit
cycle
```

基本的プログラミング

1．メインプログラムの開始と終了

実行開始

実行動作

実行終了

SURJUDP SURJ1

.....

.....

HQG SURJUDP SURJ1

メインプログラムの書き方

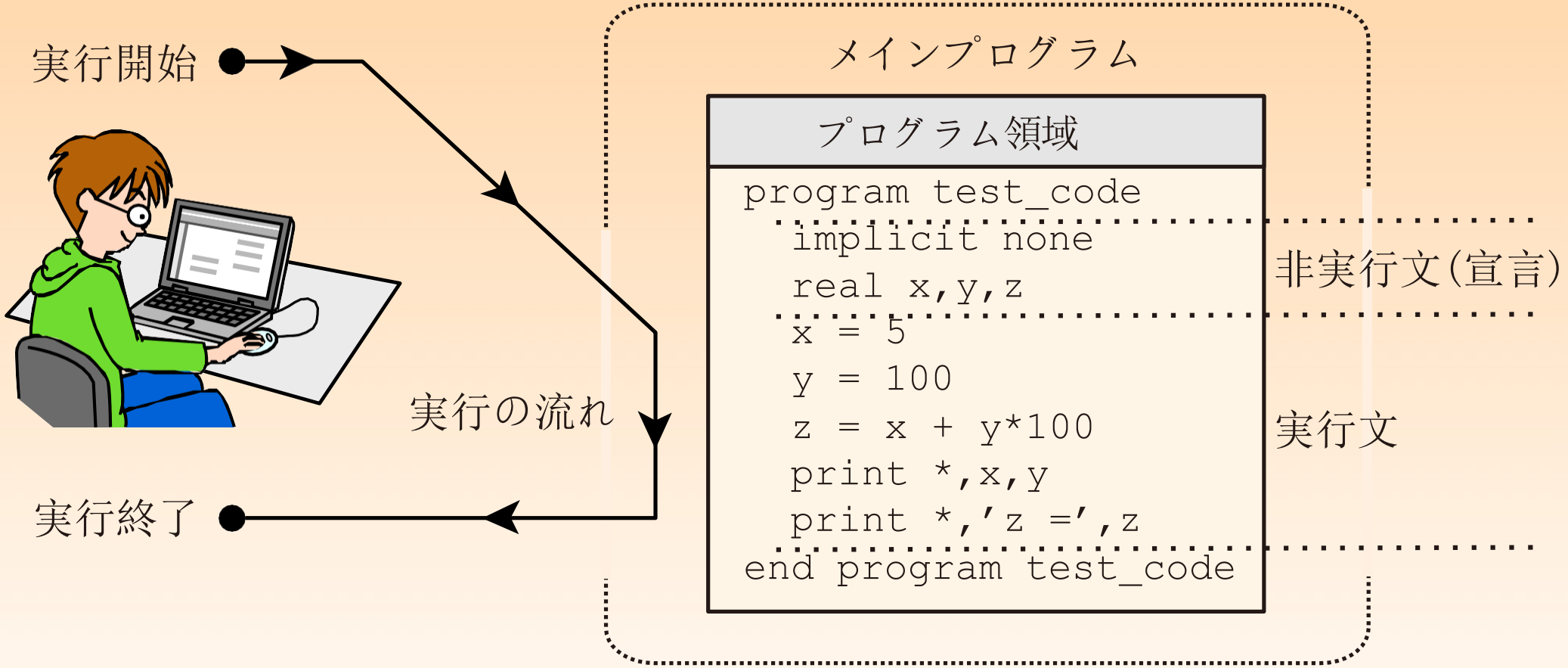
program文と end program文

```
program プログラム名      ← program文
 .......
  .......
end program プログラム名      ← end program文
```

- program文は書かなくてもメインプログラムと認識されるが，書いた方が良い
- end program文はendだけでもエラーではないが，書いた方が良い
- 途中で終了したいときには，stop文を書く

```
program test_code
  implicit none
  real x,y,z
  ............
  if (m < 0) stop     ← m<0 の場合にはここで終了する
  ............
end program test_code
```

メインプログラムでの動作



基本的プログラミング

2．代入文

変数　＝　数値

左辺の「変数」で示されたメモリに右辺で示された「数値」を格納する動作

ここで，「数値」とは

1．定数　　10.0，-500，12e-5　など

2．変数　　x，xyz など

3．計算式　10*5，sin(x)，x+20*(y-5) など

「計算式」とは，計算して得られる結果の数値を意味する

例えば，

```
x = 4 + 2
y = 9 - x
y = y + 1
```

代入文は，最後に処理するので，最後の文は，y+1の結果をyに代入する

基本的プログラミング
3．基本的な演算の書き方

演算記号	演算の意味	使用例	使用例の意味
+	足し算	x+y	x + y
−	引き算	x−y	x − y
*	掛け算	x*y	x ×y
/	割り算	x/y	x ÷y
**	べき乗	x**y	x^y
−	マイナス	−x	− x

かっこ ＞　べき乗 ＞　乗算または除算　＞　加算または減算

例えば,

```
f = x + (y-3)/z**2
```

なお,

```
f = x + y/a*b   →   f = x + (y/a)*b
```

なので注意しよう！

基本的プログラミング
4.1. 整数型

1. integer (kind) : 整数型

★ 整数のみ取り扱う：500とか，-11245など

★ kind = 型パラメータ、**ほとんど**の処理系ではバイト数

integer(4)：32bitの整数型、integer(8)：64bitの整数型 (処理系依存)

kindなしだとinteger(4)相当

★ 処理系依存ではない書き方のおすすめは2003の仕様で書く

use iso_fortran_env　でモジュールを取り込んだ上で

integer(int32)：32bitの整数型、integer(int64)：64bitの整数型

★ integer(int32) の表現可能範囲は -2^{31}～$2^{31}-1$

★ integer(int64) の表現可能範囲は -2^{63}～$2^{63}-1$

★ 割り算が切り捨てになる

★ 数値の比較が確実に行える　**（ループ変数は必ず整数型で）**

基本的プログラミング
4.2. 実数型

1. real (kind) : 実数型

★ 実数を扱える：0.125とか，-0.5など

★ kind = 型パラメータ、**ほとんど**の処理系ではバイト数

real(4)：単精度(32bit)、real(8)：倍精度(64bit)　(処理系依存)

kindなしだとreal(4)相当

★ 処理系依存ではない書き方は

real(kind(0e0))：単精度、real(kind(0d0))：倍精度

iso_fortran_envを取り込んだ上でのreal(real32)、real(real64)も可

★ 単精度の有効桁数はおおよそ7桁、指数は±38乗

★ 倍精度の有効桁数はおおよそ15桁、指数は±308乗

★ 単精度の数値の初期化は 1.5**e**-2など (0.0015)

★ 倍精度の数値の初期化は 1.5**d**-2など (0.0015)

数値型でよくやる間違い

例えば,

```
x = (5 / 2) ** 2
y = x ** (3 / 2)
t = 2 / 3 * y * x
```

の結果は，x=4，y=4，t=0である！これは整数が切り捨てだから

```
real(0d0) x
x = 0.0
```

の結果は0.00000000012とか、端数が残る
→ 型が倍精度で初期化が単精度だから

つまり、切り捨てを防いでかつ意図しない端数を防ぐには

```
x = (5.0d0 / 2.0d0) ** 2.0d0
y = x ** (3.0d0 / 2.0d0)
t = 2.0d0 / 3.0d0 * y * x
```

と書かなければならない
.0d0と書く癖をつける！

☆Fortranの便利な文法☆
Fortran には複素数型が用意されている

3. complex(kind)：複素数型

★ 有効数字が15桁程度の実数２個が「実部，虚部」として
一つの「数値」として取り扱える

★ 処理系非依存の宣言は

complex(kind(0e0))：単精度、complex(kind(0d0))：倍精度

★ 複素数定数は（実部の実数，虚部の実数）という形式
(0.0d0,1.0d0)，(1e-5,-5.2e3)，(-3200.0d0,0.005d0)など

★ 複素数に対する四則演算や関数計算ができる

★ 有効桁数、指数の範囲は単精度、倍精度に準拠

異なる数値型で計算をするときの型変換

- 整数型と実数型　→　実数型
- 整数型と複素数型　→　複素数型
- 実数型と複素数型　→　複素数型
- 代入する場合には右辺の数値を左辺の数値型に変換して代入する
  - 整数型＝実数型　　は右辺の値を「切り捨て」
  - 実数型＝複素数型　は右辺の実部を代入
  - 複素数型＝実数型　は複素数型の実部に代入して虚部は0

計算は左から行うのが基本なので，例えば，

```
t = 2/3*y*x
```

は0になるが，

```
t = y*x*2/3
```

ならばＯＫだが、おとなしく.0d0をつける

基本的プログラミング
5．変数の宣言

1.暗黙の宣言を禁止する（全ての変数は宣言して使う）

```
program code1
   implicit none     ← 必ずこの文を挿入する
   .........
```

★ 予期せぬ変数の書き間違いを防ぐ

★ 変数の型を意識する

【暗黙の宣言とは，頭文字が，A-H，O-Z が実数型，I,J,K,L,M,Nが整数型】

2.整数型変数の宣言

```
integer 変数１,変数２,...
```

3.倍精度実数型変数の宣言

```
real(0d0) 変数１,変数２,...
```

4.倍精度複素数型変数の宣言

```
complex(kind(0d0)) 変数１,変数２,...
```

変数名の付け方

- 頭文字はA-Z，それ以外は数字を交えても良い
  - a，b1，abc，ab12cd，など
- 大文字と小文字は区別をしない
  - ABCとabcとAbcは，全て同一変数
- 実数型が一番良く使うので，特に制限する必要はない
- 整数型は i-n の頭文字がよい
- 複素数型変数は少ないので，頭文字を決めておく方が良い
  - cやzが多い (BLASやLAPACKがこれ)
- プログラム全般で共通して使う変数は長くて意味が分かる名前を付ける

基本的プログラミング

6．組み込み関数

組み込み関数	名　称	数学的表現	必要条件	関数値f の範囲
sqrt(x)	平方根*	$\sqrt{x}$	$x \geqq 0$	
sin(x)	正弦関数*	$\sin x$		
cos(x)	余弦関数*	$\cos x$		
tan(x)	正接関数	$\tan x$		
asin(x)	逆正弦関数	$\sin^{-1} x$	$-1 \leqq x \leqq 1$	$-\pi/2 \leqq f \leqq \pi/2$
acos(x)	逆余弦関数	$\cos^{-1} x$	$-1 \leqq x \leqq 1$	$0 \leqq f \leqq \pi/2$
atan(x)	逆正接関数	$\tan^{-1} x$		$-\pi/2 < f < \pi/2$
atan2(y,x)	逆正接関数	$\tan^{-1}(y/x)$		$-\pi < f \leqq \pi$
exp(x)	指数関数*	e^x		
log(x)	自然対数*	$\log_e x$	$x > 0$	
log10(x)	常用対数	$\log_{10} x$	$x > 0$	
sinh(x)	双曲線正弦関数	$\sinh x$		
cosh(x)	双曲線余弦関数	$\cosh x$		
tanh(x)	双曲線正接関数	$\tanh x$		

☆Fortranの便利な文法☆
総称名機能あり

組み込み関数の使用例

```
c = exp(-x**2) + sin(10*x+3) - 2*tan(-2*log(x))**3
```

- 引数には数式を入れることも可能

総称名機能

- 表の＊の付いた関数は引数の数値型に対応した関数計算をする
  - ☆ xが実数型なら実数の値を出力　　x
  - ☆ が複素数型なら複素数の値を出力
  - ☆この表の引数の必要条件は「実数」を引数にしたとき

その他の組み込み関数

型変換関数や絶対値など

組み込み関数	名　称	引数の数値型	関数値の数値型	関数の意味
real(n)	実数化	整数	実数	実数型に変換
abs(n)	絶対値	整数	整数	n の絶対値
mod(m,n)	剰余	2 個の整数	整数	m を n で割った余り
int(x)	整数化	実数	整数	整数型に変換（切り捨て）
nint(x)	整数化	実数	整数	整数型に変換（四捨五入）
sign(x,s)	符号の変更	実数	実数	$s \geqq 0$ なら $\lvert x \rvert$，さもなくば $-\lvert x \rvert$
abs(x)	絶対値	実数または複素数	実数	x の絶対値
mod(x,y)	剰余	2 個の実数	実数	x を y で割った余り
real(z)	複素数の実部	複素数	実数	z の実部
imag(z)	複素数の虚部	複素数	実数	z の虚部
cmplx(x,y)	複素数化	2 個の実数	複素数	x + iy
conjg(z)	共役複素数	複素数	複素数	z の共役複素数

基本的プログラミング
7．簡易ディスプレイ出力 print文

```
print *,数値1,数値2,...
```

- ＊は「標準形式」を示すので，とりあえず付けておく
- 数値1，数値2，が横並びでディスプレイに出力される

```
integer n
n = 3
print *, 4+5, n, n*2, 2*n-11
```

→ 9 3 6 -5

- ’文字’の形式を「文字列」といい，その文字を出力する

```
real x
x = 3
print *,'x = ',x,'   x**3 = ',x**3
```

→ x = 3.000000000000000 x**3 = 27.000000000000000

基本的プログラミング
8．コメント文，継続行，複文

1. ! 以下の記述は無視される（コメント）

```
!  area of circles
   s = pi*r*r              !  円の面積
   v = 4*pi*r*r*r/3        !  球の体積
```

2. & を使って複数行で１つの文を記述できる（継続行）

```
   print *,alpha,beta,gamma &
                 ,delta,epsilon &
                 ,zeta,eta,iota
```

は，次の１行と同じ

```
   print *,alpha,beta,gamma,delta,epsilon,zeta,eta,iota
```

3. ; を使って複数の文を１行で書ける（複文）

```
   x = 1; y = 2; z = 3
```

ただし，やたらに１行で書かない方が良い

基本的プログラミング
9．プログラミングの注意事項

1. 計算機には，演算の得手・不得手がある

加減算　>=　乗算　≫　除算　≫　べき乗

例えば，次のような変形をすると速くなる

```
x = a/b/c   →   x = a/(b*c)
```

```
x = a**2 + b**3   →   x = a*a + b*b*b
```

関数計算も遅いので，同じ計算をくり返さないようにした方が良い

2. 桁落ちには気をつけなければならない

```
2000.06 - 2000.00 = 0.06
```

差の小さい数字の引き算をすると有効数字が下がる

２次方程式の解の計算など，注意が必要

プログラムの書き方のまとめ

- 文字の大文字と小文字の区別はありません.
- 1行はリターン記号で終わり.

  レガシーフォーマット(F70/77)だと1行は132文字まで(90以上で書く限りは気にしなくていい)

  gfortranでは132文字以上でエラーを出すので、--ffree-line-length-noneをつける
- ！を書くと、それ以降の文字は行末（リターン記号）までコメント

  （プログラムの実行に直接関係のない文字）として扱われます.
- 命令が画面上で1行に入りきらない場合などには空白部分で＆を書いて
  リターンします．この場合には次の行が＆の続きとして扱われます.
- １行の中に複数の命令文などをセミコロンで区切ることが可能です.
  ただし，プログラムが読みにくくなる場合が多いことから，
  1行はなるべく1文（1命令）のみ書くことを推奨します.
- プログラムは、必ず1行目がｐｒｏｇｒａｍ　プログラム名で始まり，
  最後が，ｓｔｏｐまたは ｅｎｄ ｐｒｏｇｒａｍ プログラム名で終わるように書く.

プログラムの書き方まとめ

```
program test1
implicit none
print *, 'Today is Wednesday'
print *, 'My Name is XXXX'
stop
end program test1
```

```
program test1
implicit none; print *, 'Today is Wednesday'; print *, 'My Name is XXXX'
stop
end program test1
```

```
program test1
implicit none; print *,&
'Hi Taro!'; print *, "Yah Hana-chan!"
stop
end program test1
```

プログラムを実行してみましょう（準備）

- システムにログインしてください.
  - ☆今回は小規模のプログラムのみ実行しますので，フロントエンドサーバでインタラクティブ動作で制御します.
  - ☆今日のディレクトリを作りましょう

```
#  mkdir 28Oct2024 #
cd 28Oct2024
#  vi test1.f95
```

    - ☆(emacs, nanoなど，他のエディタでもいいです)

(練習１)プログラムを実行してみましょう

```
program test1
implicit none
print *, 'Today is Wednesday'
print *, 'My Name is XXXX'
stop
end program test1
```

- 上記のプログラムをエディタを使って作成の上，ファイル名を”test1.f95”としてください.
- 以下の手順に従ってプログラムをコンパイル実行してください.

```
#  gfortan  -o  ./test1  ./test1.f95
#./test1
```

練習 2

```
program test2
implicit none
integer :: a, b, ans

a=1
b=2

and = a + b

print *, ans

stop
end program
```

- 上記のプログラムを作成・コンパイル・実行してください.
  - 適宜演算子，変数を代えてみてください.

課題 1

- 以下の式を解く(aを求める)プログラム(exec1.f95)を作成・コンパイル・実行してください.

$$a=3.141592r^2+3x^5+6.5\times10^{-5}x-10^5$$

繰り返しの基本

配列とは

- 複数の同等のデータをひとまとめにして扱う「変数」の一種
- 「要素番号（添字）」でデータを識別する
- 識別する要素番号の数で「次元」が異なる

```
1 次元配列　a(1)，a(15)，…
2 次元配列　b(1,1), b(3,2),b(100,200),…
3 次元配列　c(1,1,1),c(15,3,2),c(2,5,6),…
```

配列は，宣言文で型宣言をする時に，「次元」と「要素番号の最大値」を指定する

```
real(kind(0d0))         :: a(10),b(20,30)  !倍精度実数型配列
complex(kind(0d0)) :: a(10),b(20,30)  !倍精度複素型配列
integer                       :: a(10),b(20,30)  !整数型配列
```

指定された要素番号の要素は，その数値型を持った変数として利用できる

```
a(15) = b(2,3)*10 - cos(2*a(3))
b(i,j) = a(i+2*j)*a(j)
```

要素番号には変数や数式（ただし整数型のみ）を使っても良い

配列の要素番号の範囲

real(kind(0d0)) :: a(5)の宣言では

a(1),a(2),a(3),a(4),a(5)　の5個が使用可能

real(kind(0d0)) :: b(5,3)の宣言では

b(1,1),b(2,1),b(3,1),b(4,1),b(5,1),
b(1,2),b(2,2),b(3,2),b(4,2),b(5,2),
b(1,3),b(2,3),b(3,3),b(4,3),b(5,3)　の15個が使用可能

☆Fortranの便利な文法☆　Fortran では下限の指定ができる

real() :: 配列名(下限:上限)

real(kind(0d0)) :: ac(-2:2),bc(-5:20,0:100),cc(0:5,3)　など

例えば，acは，

ac(-2),ac(-1),ac(0),ac(1),ac(2)　の5個が使用可能

配列とは，連続したメモリのこと

real a(10)の宣言では

real a(10)

	メモリアドレス	内部メモリ
a →	101:	a(1)
	102:	a(2)
	103:	a(3)
	104:	a(4)
	105:	a(5)
	106:	a(6)
	107:	a(7)
	108:	a(8)
	109:	a(9)
	110:	a(10)
	....	

real ac(-3:5)の宣言では

real ac(-3:5)

	メモリアドレス	内部メモリ
ac →	201:	ac(-3)
	202:	ac(-2)
	203:	ac(-1)
	204:	ac(0)
	205:	ac(1)
	206:	ac(2)
	207:	ac(3)
	208:	ac(4)
	209:	ac(5)
	210:	
	....	

「配列名」は，メモリの先頭アドレスを示す

静的配列と動的配列

配列には静的配列と動的配列がある

- real(kind0d0)) :: a(10) !静的配列
- real(kind0d0)), allocatable :: a(:) !動的配列

- 静的配列はスタックに領域を確保
- 静的配列はヒープに領域を確保

スタックは大きいサイズの配列を確保できない(大体数メガ)ので、
大きい配列は動的配列を使う！

動的配列の使い方
real(kind(0d0)), allocatable :: array(:)

allocate(array(N)) !サイズNのメモリを確保

手順のくり返し ― do 文

```
GR L    1, 100

D(L)    0.1*L
E(L)    D(L)**2 + 20

HQGGR
```

do 文によるくり返し

- 同じパターンの動作を所定の回数くり返す時に使う

```
do 整数型変数 = 初期値, 終了値
  .....
  .....
enddo
```

↕ この間（doブロック）をくり返す

初期値，終了値は，整数型変数や整数型の計算式でも良い

do文の動作は，以下の通り

① 整数型変数（カウンタ変数）に初期値を代入する

② カウンタ変数と終了値を比較して，終了値以下ならdoブロックを実行

③ enddoに到達したら，カウンタ変数を1増加させて②に戻る

例えば，

```
do m = 1, 10
   a(m) = m
enddo
```

は，a(1)=1，a(2)=2，...，a(10)=10と同じ動作になる

増分付きのくり返しも可能

```
do 整数型変数 = 初期値, 終了値, 増分値
  .......
  .......
enddo
```

増分値は，整数型変数や整数型の計算式でも良い

この場合，do文の動作は，以下の通り

① 整数型変数（カウンタ変数）に初期値を代入する

② カウンタ変数と終了値を比較して，終了値以下ならdoブロックを実行

③ enddoに到達したら，カウンタ変数を「増分値」増加させて②に戻る

例えば，

```
do m =1, 10, 2
  a(m) = m
enddo
```

は，a(1)=1，a(3)=3，…，a(9)=9と同じ動作になる（奇数のみ実行）

増分は負数でも良い

```
do 整数型変数 = 初期値, 終了値, 負の増分値
 .......
 .......
enddo
```

この場合，do文の動作は，以下の通り

① 整数型変数（カウンタ変数）に初期値を代入する

② カウンタ変数と終了値を比較して，終了値以上ならdoブロックを実行

③ enddoに到達したら，カウンタ変数を「増分値」減少させて②に戻る

例えば，

```
do m =10, 1, -1
 a(m) =m
enddo
```

は，a(10)=10，a(9)=9，…，a(1)=1と同じ動作になる

do文を使うときの注意

- カウンタ変数は，doブロック内で変更してはいけない
- doブロックから外へ出ても良いが，外から中に入ることはできない
- カウンタ変数は，enddoに到達した段階で増分値を加えるので，doブロックを終了したときのカウンタ変数の値を使うときには注意する
- 初期値，終了値，増分値はdoブロックの開始時に決定されるので，doブロック内部で変更しても無関係

```
j = 10
k = 1
do i = 1, j, k
   print *,i,j,k
   j = 20
   k = 2
```

のように書いても良いが，終了値や増分値は最初決めた数値でくり返す

do文終了時のカウンタ変数

例えば,

```
do m =1, 3
  a(m) =m**2
enddo
```

というプログラムの動作は,

```
   m =1
 [ m>3の判定をする．満足しないので，doブロックを実行 ]
   a(m) =m**2
   m =m +1
   [m>3の判定をする．満足しないので，doブロックを実行 ]
   a(m) =m**2
   m =m +1
 [ m>3の判定をする．満足しないので，doブロックを実行 ]
   a(m) =m**2
   m =m +1
 [ m>3の判定をする．満足するので，doブロックを終了 ]
```

よって，このdoブロックを終了した時点で m には4が代入されている

doブロックの中にdoブロックを入れても良い

例えば，

```
do k = 1, 100
  a(k) = k**2
  do m = 1, 10
    b(m,k) = m*a(k)**3
    c(m,k) = b(m,k) + m*k
  enddo
  d(k) = a(k) + c(10,k)
enddo
```

ただし，内側のカウンタ変数と外側のカウンタ変数は

異なるものを使わなければならない　→　内部で変更できないから

合計を計算するdo文を覚えておこう

例えば，a(1)+a(2)+a(3)+...+a(10)を計算したいときは，

```
sum = 0
do m = 1, 10
  sum = sum + a(m)
enddo
```

のように書けばよい，これは，

```
sum = 0
m = 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
    .....
```

という動作だから

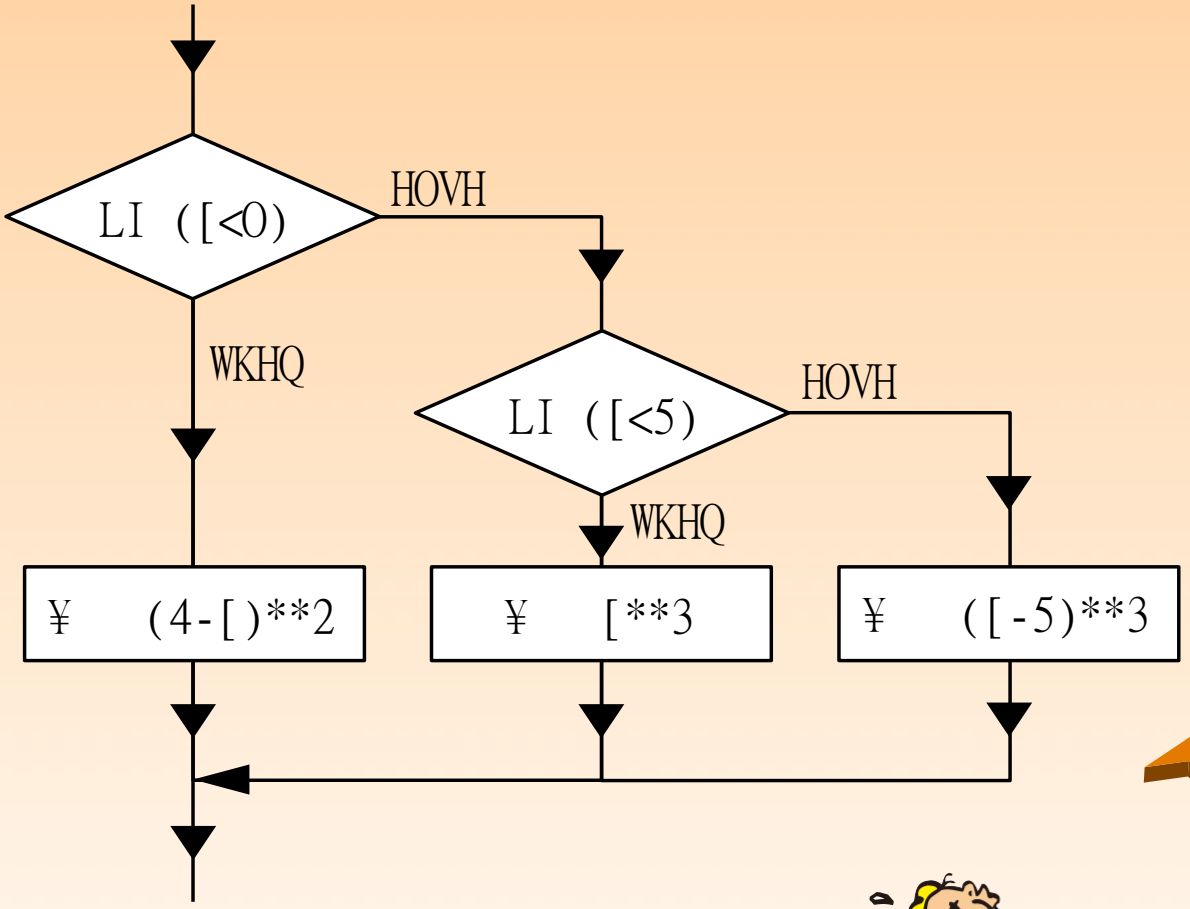
なお，最初の sum=0 を忘れないように！（ゼロリセット）

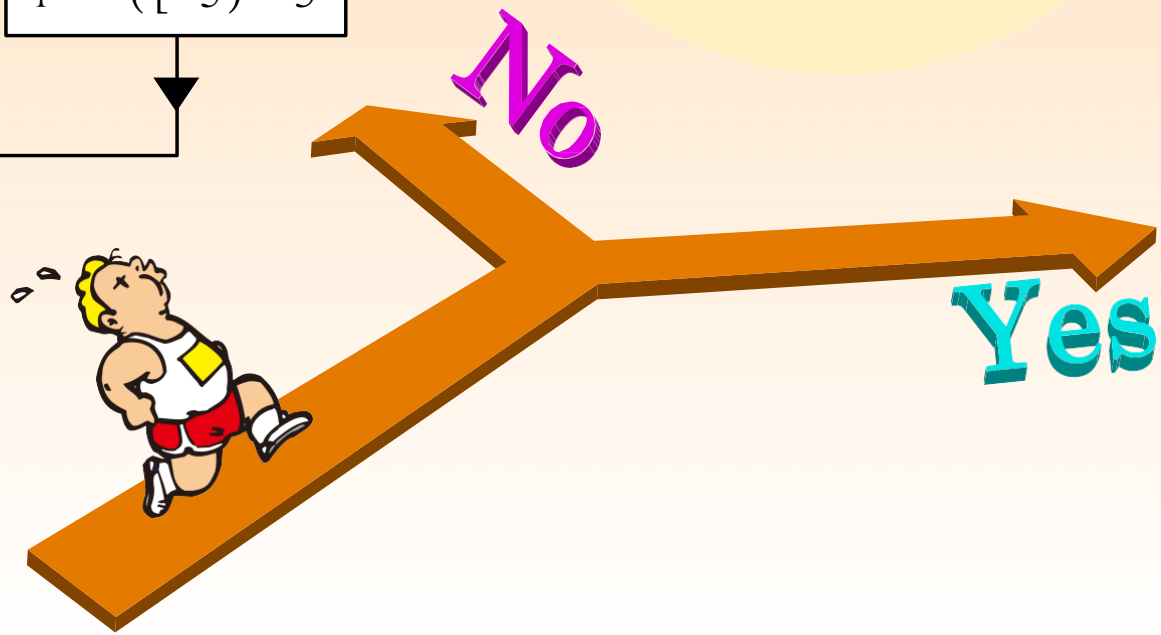
加算を乗算にすれば，階乗計算もできる

課題２

- １から１００までの総和を求めるプログラム(exec2.f95)を作成・コンパイル・実行してください.

条件分岐とジャンプ





if 文による条件分岐

- 条件に応じて異なる動作をさせるときに使う

単純if文

```
if (条件) 実行文
```

★ 条件が「真」の時，右の実行文を実行する
★「偽」の時は，何もせず次の文に移る
★ １個の実行文しか実行できない

ブロックif文

```
if (条件) then
  .......
  .......
endif
```

★「真」の時，ブロック内の実行文を実行する
★ 複数の実行文を実行できる

ブロックif 文による条件分岐

★ ブロックif文は「偽」の時に別の動作をさせることもできる

else付きのブロックif文

```
if (条件) then
  .....        条件が「真」のとき実行
 else
  .....        条件が「偽」のとき実行
endif
```

```
if (条件1) then
  .....        条件1が「真」のとき実行
 else if (条件2) then
  .....        条件1が「偽」で条件2が「真」のとき実行
 else
  .....        条件1も条件2も「偽」のとき実行
endif
```

★ else if文は複数指定することができる

★ if ブロックの中に別のif文やifブロックを入れることもできる

比較条件の書き方

比較条件の書き方は以下の通り

比較条件記号	記号の意味	使用例
== または .eq.	左辺と右辺が等しいとき	x == 10
/= または .ne.	左辺と右辺が等しくないとき	x+10 /= y-5
> または .gt.	左辺が右辺より大きいとき	2*x > 1000
>= または .ge.	左辺が右辺以上のとき	3*x+1 >= a(10)**2
< または .lt.	左辺が右辺より小さいとき	sin(x+10) < 0.5
<= または .le.	左辺が右辺以下のとき	tan(x)+5 <= log(y)

★ 比較条件は，両辺に数式を書いても良い

★ 両辺の数値型が合っていないときは，情報の多い方に合わせて比較する

ifブロックの使用例 1

単純if文

```
a  = 5
if (i < 0) a  = 10
b  = a**2
```

ブロックif文

```
a  = 5
b  = 2
if (i < 0) then
   a  = 10
   b  = 6
endif
c  = a*b
```

ifブロックの使用例２

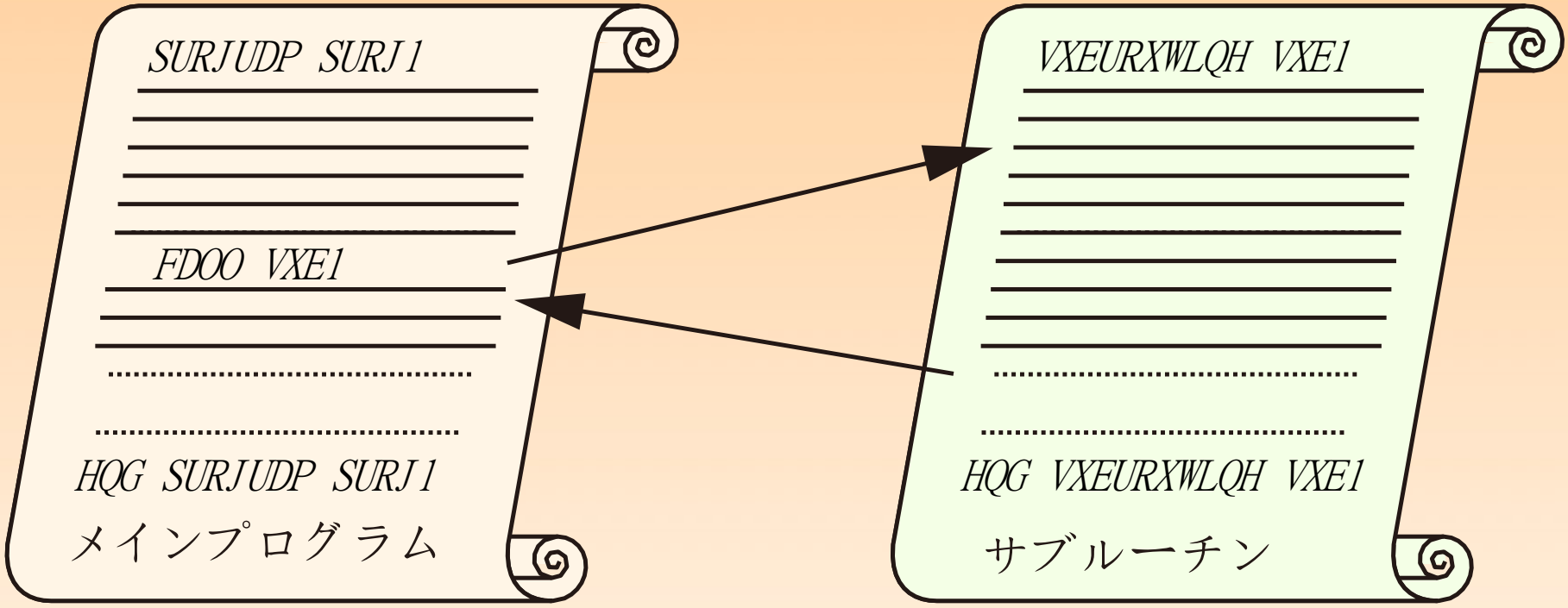
else 付きブロックif文

```
if (i < 0) then
  a  = 10
  b  = 6
 else
  a  = 5
  b  = 2
endif
c  = a*b
```

else if付きブロックif文

```
if (i < 0) then
  a  = 10
  b  = 6
 else if (i < 5) then
  a  = 4
  b  = 7
 else
  a  = 5
  b  = 2
endif
c  = a*b
```

サブルーチンの利用


SURJUDP SURJ1
FDOO VXE1
HQG SURJUDP SURJ1
メインプログラム
VXEURXWLQH VXE1
HQG VXEURXWLQH VXE1
サブルーチン

サブルーチンとは

- 開始点と終了点を持つ独立したプログラム（プログラム単位）
- 内部で宣言した変数は，内部でのみ有効（ローカル変数）
- 他のルーチン（メインプログラムかサブルーチン）からのみ利用可能

サブルーチンを利用する目的

- 複数の場所で同じ計算手順を使用するため
- 方程式の解法や行列式の計算などのような定型処理をするため
- 長いプログラムを分割してメンテナンスを容易にするため

  複数ファイルに分割して管理も可能(コンパイル時間の短縮)

サブルーチンの宣言と呼び出し

subroutine文と end subroutine文

```
subroutine サブルーチン名
  implicit none
  real a,b
  integer i
   .......
   .......
end subroutine サブルーチン名
```

★ subroutine 文から end subroutine 文までが１セット（プログラム単位）

★ implicit noneを書く

★ 宣言文などの非実行文を書く

★ 実行文を書く

　　→　構造は，メインプログラムと同じ

★ サブルーチンとメインプログラムの順序は自由である

引数なしサブルーチンと引数ありサブルーチン

引数なしサブルーチン (サブルーチン名のみ)

```
subroutine サブルーチン名
  implicit none
  real a,b
  integer i
   .......
   .......
end subroutine サブルーチン名
```

引数ありサブルーチン

```
subroutine サブルーチン名(引数1,引数2,...)
  implicit none
  real a,b
  integer i
   .......
   .......
end subroutine サブルーチン名
```

★「引数」は１変数（１語）

★ 引数も宣言しなければならない

サブルーチンの呼び出し ― call 文

- サブルーチンは，他のルーチンから呼び出して初めて有効
- 呼び出しにはcall文を使う

引数なしサブルーチンの呼び出し

```
call サブルーチン名
```

引数ありサブルーチンの呼び出し

```
call サブルーチン名(数値1, 数値2, 数値3,...)
```

★ 「数値」には「数式」を書いても良い（ただし注意あり！）
その場合には，数式の結果の数値がサブルーチンに与えられる

サブルーチン実行の流れ

例えば，次のプログラムの流れは．．．

```
program stest1
   implicit none
   real(kind(0d0)) x,y
   x = 5.0d0
   y = 100.0d0
   call subr(x,y,10) print ！サブルーチンの呼び出し
   *,x,y
end program stest1

subroutine subr(x,y,n)
   implicit none
   real(kind(0d0)) x,y
   integer n
   x = n
   y = y*x
end subroutine subr
```

サブルーチン実行の流れ

．．．このようになる

メインプログラム

プログラム領域

```
program stest1
  implicit none
  real x,y
  x = 5.0
  y = 100.0
  call subr(x,y,10)
  print *,x,y
end program stest1
```

実行の流れ

サブルーチン

プログラム領域

```
subroutine subr(x,y,n)
  implicit none
  real x,y
  integer n
  x = n
  y = y*x
end subroutine subr
```

① call 文を実行するとサブルーチンの開始点にジャンプ

② サブルーチンが終了するとcall文の次の行から実行を再開する

③ サブルーチンの途中で実行を終了したいときは，return文を書く

④ サブルーチンの途中で stop 文を実行すると，プログラムが終了する

interface文

- サブルーチンの宣言と呼び出しの型の一致はユーザーが担保

```
program stest1 implicit
  none real(kind(0e0)) x,y
  call subr(x,y,10)

end program stest1

subroutine subr(x,y,n)
  implicit none
  real(kind(0d0)) x,y
  integer n
end subroutine subr
```

左のサンプルの場合
　subrへの引数は単精度、単精度、整数
　subrの宣言は　倍精度、倍精度、整数
　→ 不一致で実行時エラー

下記のようなinterface宣言が
program stest1内にあると、
コンパイル時にエラーを出してくれる。

```
interface
  subroutine subr(x,y,n)
    implicit none
    real(kind(0d0)) x,y
    integer n
  end subroutine subr
end interface
```

interface文のメリット

多くのバグや性能劣化要因を防げる

- コンパイル時に引数の違いを指摘してくれる
- リテラル引数を扱う場合のバグ回避
  リテラル引数とはsubr(x, y, 10)の10のこと
  リテラル引数への代入はNG
  防ぐ方法としてintent属性を付与
    integer, intent(in) :: n
    → n = 20でコンパイル時にエラー
    integer, intent(out) :: n
    → subr(x, y, 10)でコンパイルエラー
- 形状引継ぎ配列が使える
  (配列のサイズをサブルーチンで
  引き継ぐ機能)
- 部分配列を引数とする場合の
  一時配列作成の抑制
    call subr1(array(1:10:2)) !部分配列

```
program stest1
  implicit none
  real(kind(0d0)) x,y
  x = 5.0d0
  y = 100.0d0
  call subr(x,y,10)
  print *,x,y
end program stest1
subroutine subr(x,y,n)
  implicit none
  real(kind(0d0)) x,y
  integer n
  n = 20 !これはまずい
end subroutine subr
```

演習

次のプログラムを作成，コンパイル，実行し動作を確認してください.

```
program stest1
  implicit none
  real(kind(0d0)) x,y
  x = 5.0d0
  y = 100.0d0
  call subr(x,y,10)          ! サブルーチンの呼び出し
  print *,x,y
end program stest1

subroutine subr(x,y,n)
  implicit none
  real(kind(0d0)) x,y
  integer n
  x = n
  y = y*x
end subroutine subr
```

演習

先のプログラムに以下の変更を施してコンパイルエラーが出ることを確認してください。

・subrのinterface文を追加
・program stest1の宣言をreal(0d0)x,yをreal(0e0)x,yに変更、

まとめ

- Fotranの基本

- 本講習会の内容は摂南大学田口教授が作成したもののダイジェスト版になります．さらに実用的な内容をご希望される方は，サイバーサイエンスセンターの過去の講習会の資料をご参照ください．

ローカル変数

- サブルーチン内部で宣言した変数は，他のルーチンでの宣言とは無関係（ローカル変数）
- このため，同じ名前の変数を宣言してもかまわない

```
program stest1
  implicit none
  real x,y
  x = 10.0
  y = 30.0
  call subr1
end program stest1

subroutine  subr1
  implicit none
  real x,y
  print *,x,y                    このxとyは，10や30ではない！
end  subroutine subr1
```

★ メインプログラムとサブルーチンを別のファイルにして結合することも可能　→　これが「リンク」

★ あるルーチン内部から他のルーチンの内部は「見えない」！

数値の受け渡し法ー引数

- 数値の受け渡しには「引数」を使う

```
program stest2
  implicit none
  real x,y
  x = 10.0
  y = 30.0
  call subr2(x,y)      引数ありサブルーチン
end program stest2

subroutine  subr2(x,y)
  implicit none
  real x,y           引数を宣言する
  print *,x,y
end  subroutine subr2
```

★ 引数を使えば，数値の受け渡しができる

★「引数」とは，データの窓口であると考えられるが

あくまでもローカル変数であるので，サブルーチン側で決められる

```
subroutine  subr2(a,b)   このようにメインプログラムと無関係に決めても良い
  implicit none
  real a,b
  print *,a,b
end  subroutine subr2
```

引数を使うときの注意と戻り値

- 並びの順番や数は一致していなければならない
- call側の数値と対応する引数の型は一致していなければならない
- サブルーチン側で引数に代入をすると，コール側の対応する変数に値が代入される（戻り値）

例えば，

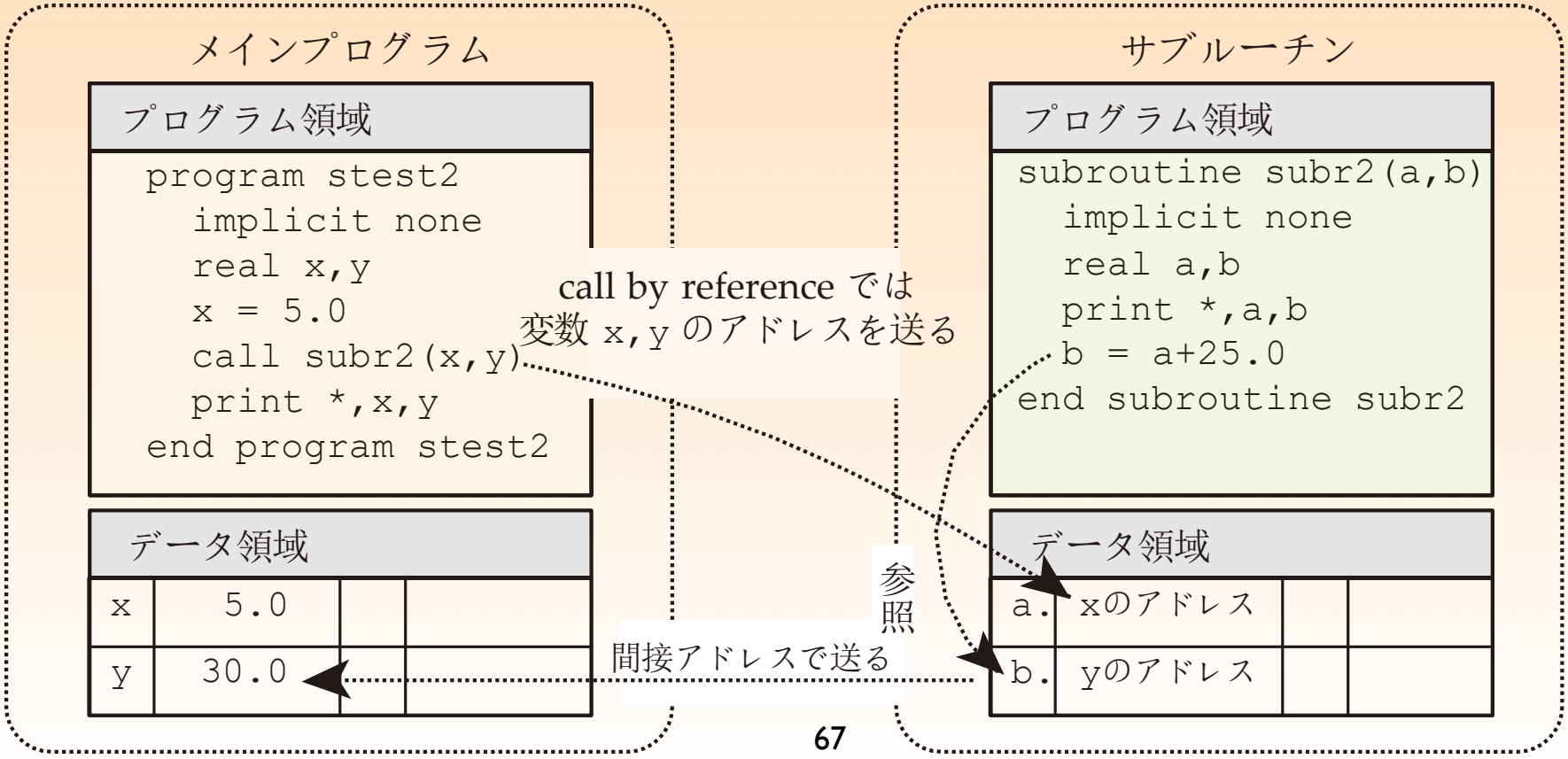
```
program stest4
  implicit none
  real x,y,p
  x = 10.0
  y = 30.0
  call subr4(x+y,20.0,p)    引数pにzの代入値が代入される
  print *,x,y,p
end program stest4

subroutine  subr4(x,y,z)
  implicit none
  real x,y,z
  z = x*y
end  subroutine subr4
```

の場合，pにはx+yと20.0の積，400.0が代入される

なぜ戻るのか？ー間接アドレス

- サブルーチン内では，「引数」は特別な存在
- サブルーチンの引数は「数値」渡しではなく，「アドレス」渡し
- 受け取ったアドレスを参照して元のルーチンの変数に代入する戻り値の引数に「定数」や「数式」を与えてはいけない！



配列を引数にする場合

- 配列の先頭アドレスを「引数」に与える
- 配列要素を与えると，それを先頭アドレスと見なす
- サブルーチン側で「配列」の利用形状を決める

例えば，

```
real a(10)
```

と宣言した配列について

```
call sub(a)　と　call sub(a(1))
```

は，同じ意味

```
call sub(a(3))
```

と書けば，a(3)が先頭要素の配列と見なす（配列の並びが重要！）

２次元配列の各行の要素を与えるため，

```
call sub(b(1,i))
```

などと記述することもある（配列の並びが重要！）

配列を引数にする場合

call sub(a)　に対し

サブルーチン側は配列でなくても良い

```
subroutine sub(x)
 implicit none
 real x              ! 単一変数宣言
 x = 10.0
end subroutine sub
```

この場合には，a(1)を利用してサブルーチンは動作する

サブルーチン側で配列として使いたければ，引数を配列宣言する

```
subroutine sub(x)
 implicit none
 real x(10)          ! 配列宣言
 integer i
 do i = 1, 10
 x(i) = i
 enddo
end subroutine sub
```

★ コール側と要素数が一致する必要はない（見えない！）

★ 次元でさえ一致させる必要はない

2次方程式の解計算の問題点

$$ax^2 + bx + c = 0$$

において，c が小さいとき，１解は０に近い

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

例えば，$b>0$の時は，x_1が０に近いので桁落ちする可能性がある

このときは，　$x_1 = \dfrac{c}{ax_2}$　で計算すると良い

$x_1 x_2 = \dfrac{c}{a}$　より