Fortran90/95入門と演習 前半

担当： 坪倉　誠
（神戸大学大学院システム情報学研究科）

目標

- 本演習で用いる数値計算用プログラム言語「Fortran90/95」の基礎を習得する。

参考資料：

TECS-KOBE第二回シミュレーションスクール(神戸大学） 2010/12/6 :Fortran 講義ノート （平尾　一）

「Fortran90/95入門」 2010年度計算科学演習I　講義資料、神戸大院システム情報学専攻・陰山聡

http://bit.ly/1n1E3ht

http://bit.ly/1fSA8Mi

※本資料は2014年度の臼井先生の資料や陰山先生作成の資料を基に、坪倉が適宜加筆しました

予定

1. イントロダクション
2. 入出力
3. 変数の型
4. 演算の基礎
5. 条件の扱い
6. 繰り返し処理
7. 配列
8. 副プログラム
9. 数値計算に向けて
10. 付録

前半

後半

イントロダクション

Fortranとは

- **世界初の高級プログラミング言語**

- **Fortran ＝ “For-mula Tran-slation”（数式翻訳）**
  ⇒数値計算に適した言語

- **なぜFortranか？**

1. 計算速度が速い（コンパイラが最適化しやすい）

- C/C++でもFortranと同じくらい速いコードは書けるが、遅いコードも書けてしまう

2. 数値計算プログラムを書きやすい

- 各種組込み関数や強力な配列操作など、数値計算に便利な機能
- 数値計算ライブラリの豊富な蓄積

Fortranの歴史

- **FORTRAN66**
  - 1966年に標準化
- **FORTRAN77**
  -1977年に標準化
  - if/then/else
  - 広く普及が進む
- **Fortran90**
  - 1991年に標準化
  - 大幅な改訂
- **Fortran95**
  - F90からのマイナーバージョンアップ
- **Fortran2003など**

Fortran90はFORTRAN77とは大きく違う。違う言語と考えるべき。

- | f95 - f90 | << | f90 - f77 |

作業に必要なもの

- **計算機**（今回は、π-Computerのフロントエンドマシンを使います）
  - 大規模な演算は「計算ノード」で→あとの演習
- **エディタ**（emacs, vi, etc. 何でもよい）
- **f95コンパイラ**（今回の演習では”gfortran”（GNU Fortran）を使います）
  - 計算ノードで実行するためのコンパイラ：frtpx

Fortranを使った作業の流れ

- **エディタを使ってプログラムを書く**
  - 大文字でも小文字でも構わない　　読みやすさが肝心
- **ファイルを保存**
  - “aaa.f95”などと名付ける　　“aaa”の部分は任意。「ソースコード」
- **コンパイル**

  % gfortran aaa.f95 → 実行ファイル“a.out”ができる
  - コンパイル時に実行ファイル名（例 aaa.exe）を指定してもよい

    % gfortran -o aaa.exe aaa.f95

    実行ファイルの拡張子に制限はない
  - 計算ノード実行用のコンパイル

    % frtpx -o aaa.exe aaa.f95
- **（リンク）**

  問題点、バグがあれば修正する（デバッグ）。コンパイルをやり直す
- **ログインノードでのプログラム実行**

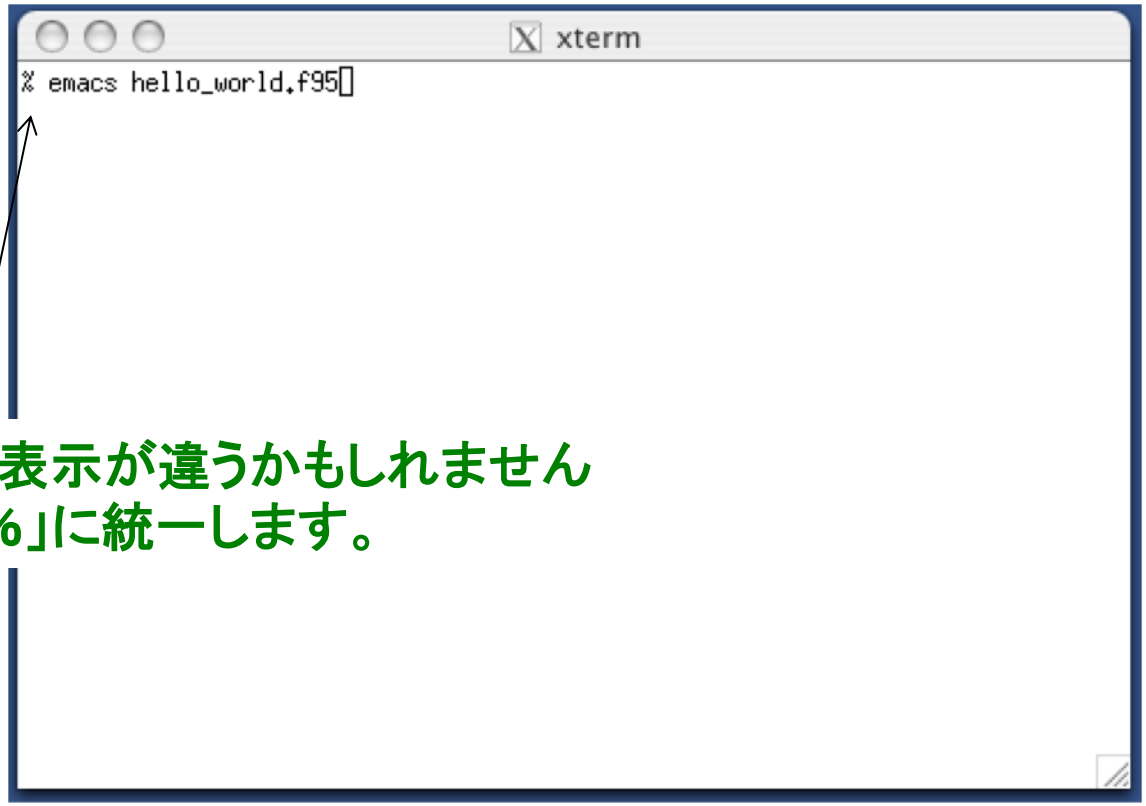
  % ./aaa.exe

  「./」はファイルの位置を指定するためにつけている

Step 1. エディタを起動する

演習a1

- 作業ディレクトリの作成（mkdir コマンド）
- 作業ディレクトリに入り、そこでファイル作成



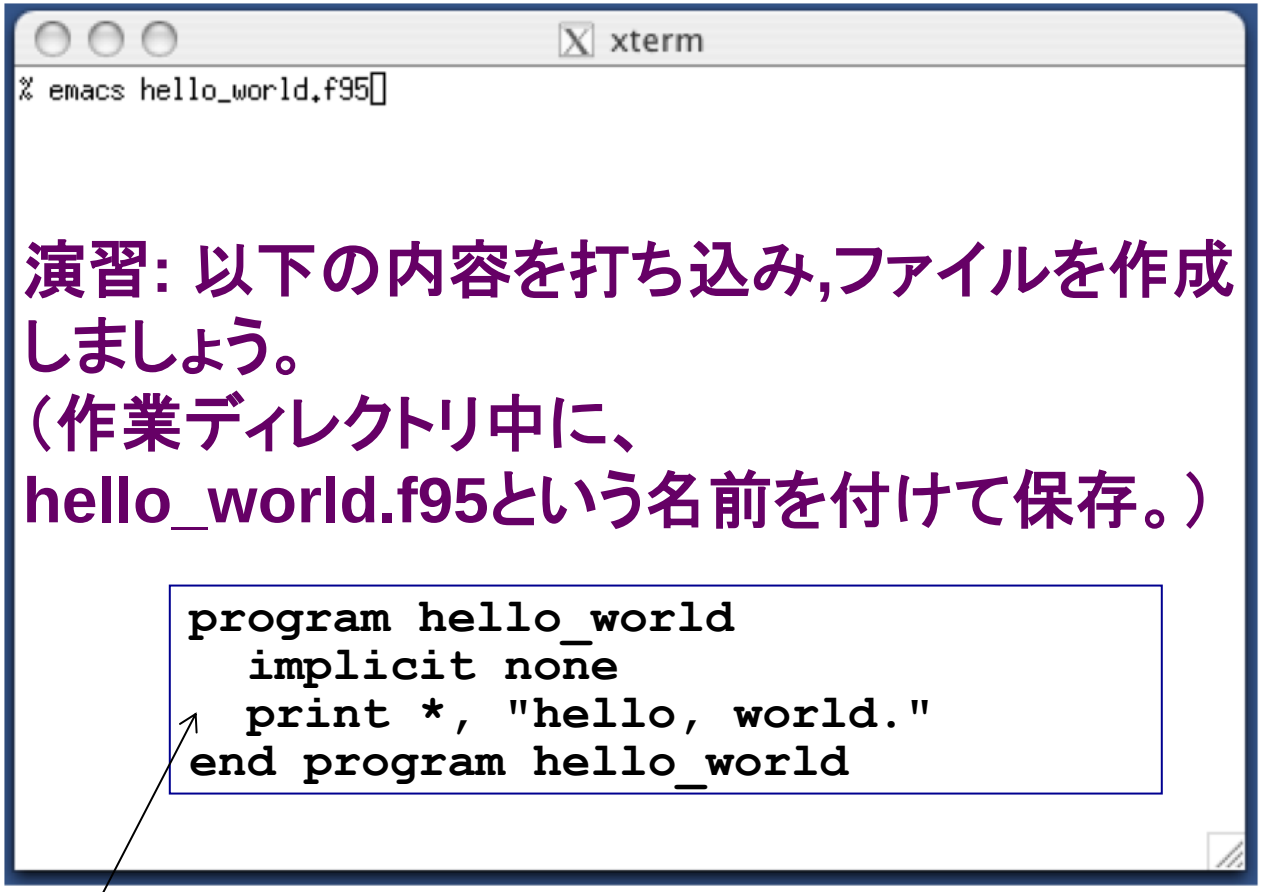
「プロンプト」の表示が違うかもしれません
が、ここでは「%」に統一します。

演習: 一緒にやってみましょう！

- ファイル名はhello_world.f95

Step 2. プログラムを書く,ファイル保存

演習a1



演習: 以下の内容を打ち込み,ファイルを作成しましょう。
（作業ディレクトリ中に、hello_world.f95という名前を付けて保存。）

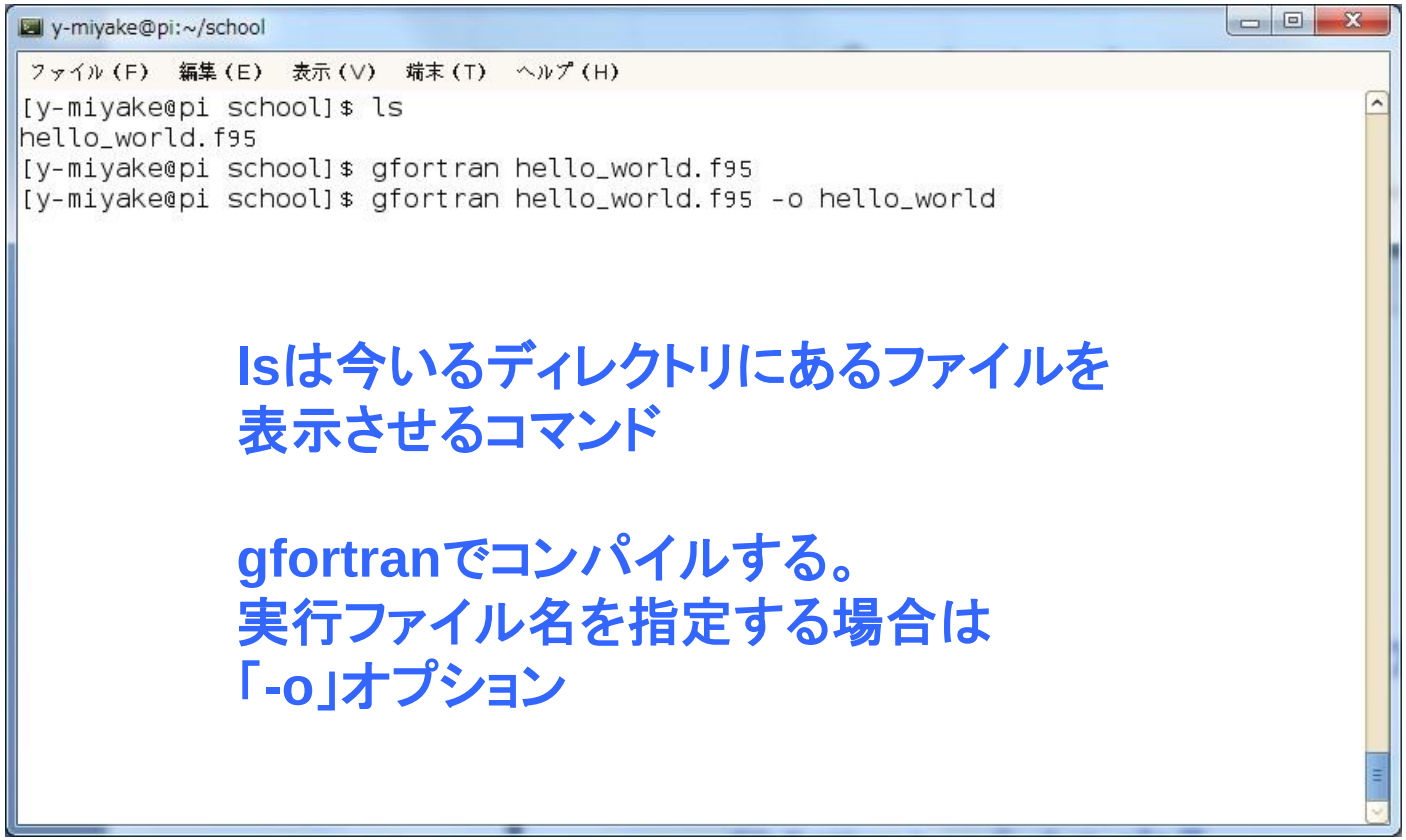
```
program hello_world
   implicit none
   print *, "hello, world."
end program hello_world
```

スペースの数は任意
ここでは二文字分で統一

保存後、ターミナル上でlsコマンドを実行し、現ディレクトリにあるファイルを確認せよ。

Step 3. コンパイルする

演習a1

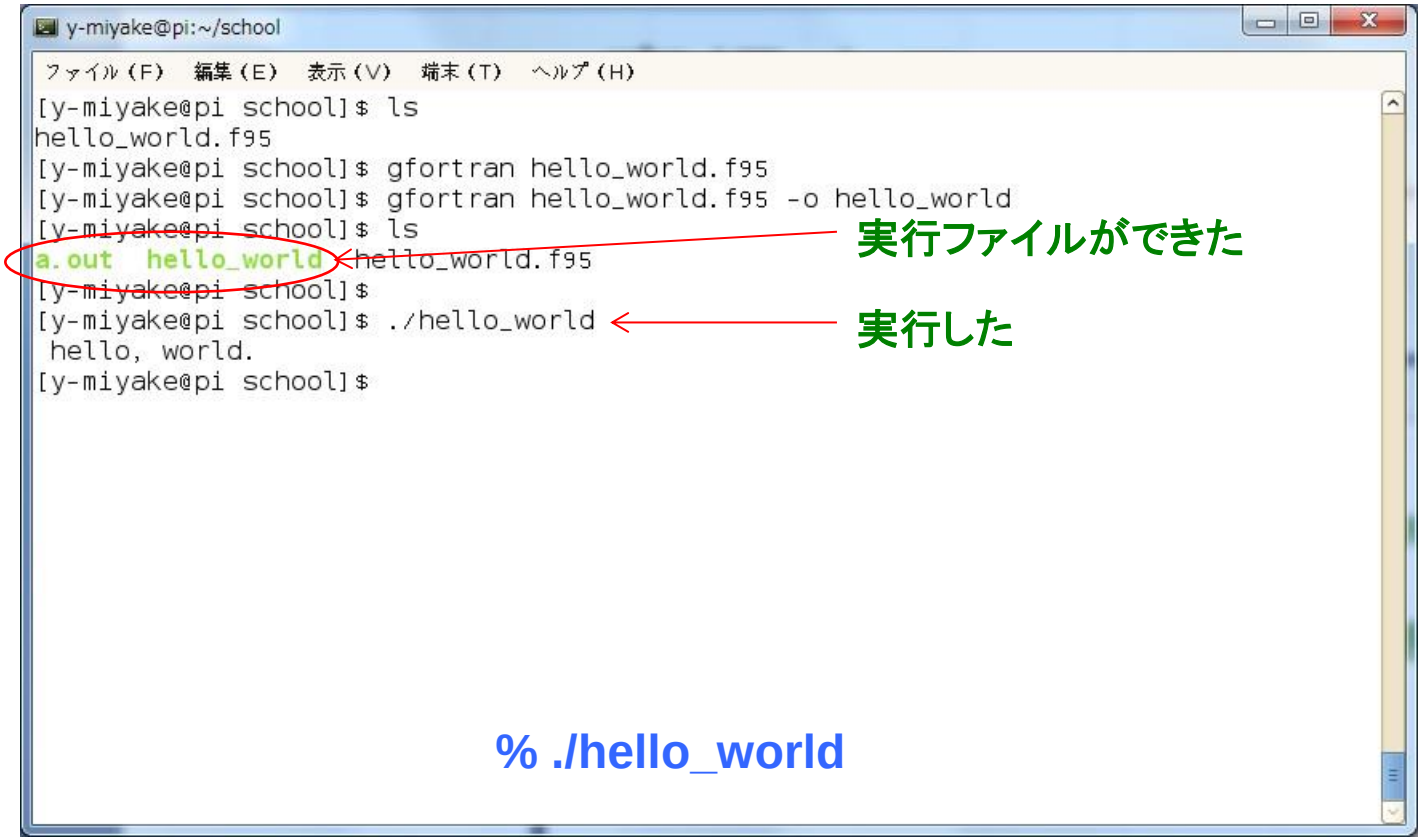


演習: 一緒にやってみましょう！

- プログラムをgfortranコマンドでコンパイル。
- lsを実行し、実行ファイル（a.outやhello_worldなど）ができていることを確認せよ。

Step 4. ログインノード上で実行する

演習a1



演習: 一緒にやってみましょう！

- 何が起こるかをみてみよう。

計算ノードで実行する場合

π-Computerの本来の使い方

```
#!/bin/sh
#------ pjsub option --------#
#PJM -L "rscgrp=school"
#PJM -L "node=1"
#PJM -L "elapse=10:00"
#PJM -j
#------- Program execution -------
#
./hello_world
```

ジョブスクリプト

```
y-miyake@pi:~/school/job0
ファイル(F)  編集(E)  表示(V)  端末(T)  ヘルプ(H)
[y-miyake@pi job0]$ ls
hello_world.f95  hello_world.sh
[y-miyake@pi job0]$ frtpx hello_world.f95 -o hello_world
[y-miyake@pi job0]$ ls
hello_world  hello_world.f95  hello_world.sh
[y-miyake@pi job0]$
[y-miyake@pi job0]$ pjsub hello_world.sh
[INFO] PJM 0000 pjsub Job 10245 submitted.
[y-miyake@pi job0]$
[y-miyake@pi job0]$ ls
hello_world  hello_world.f95  hello_world.sh  hello_world.sh.o10245
[y-miyake@pi job0]$ cat hello_world.sh.o10245
 hello, world.
[y-miyake@pi job0]$
```

frtpxでコンパイル

ジョブの実行を依頼

実行結果が格納されている
テキストファイル

- 本日の実習では、手続きの単純なログインノード実行を用います。

コメントの挿入

例

```
!=================================================
!  This is a program to say hello.
!  Coded by Kobe Taro on May 8, 2014.
!=================================================
program hello_world
  implicit none
  print *, "hello, world." ! saying hello here
end program hello_world
```

行内の、“!”以降は、コメントとして無視される

実行

```
% ./hello_world
 hello, world.
```

適宜コメントを入れるとプログラムがわかりやすくなる。
（他人だけではなく自分が後で解読できる事も重要）

行の継続

例

```
program hello_world
  implicit none
  print *, "I live in Kobe. I am going to Osaka tomorrow."
end program hello_world
```

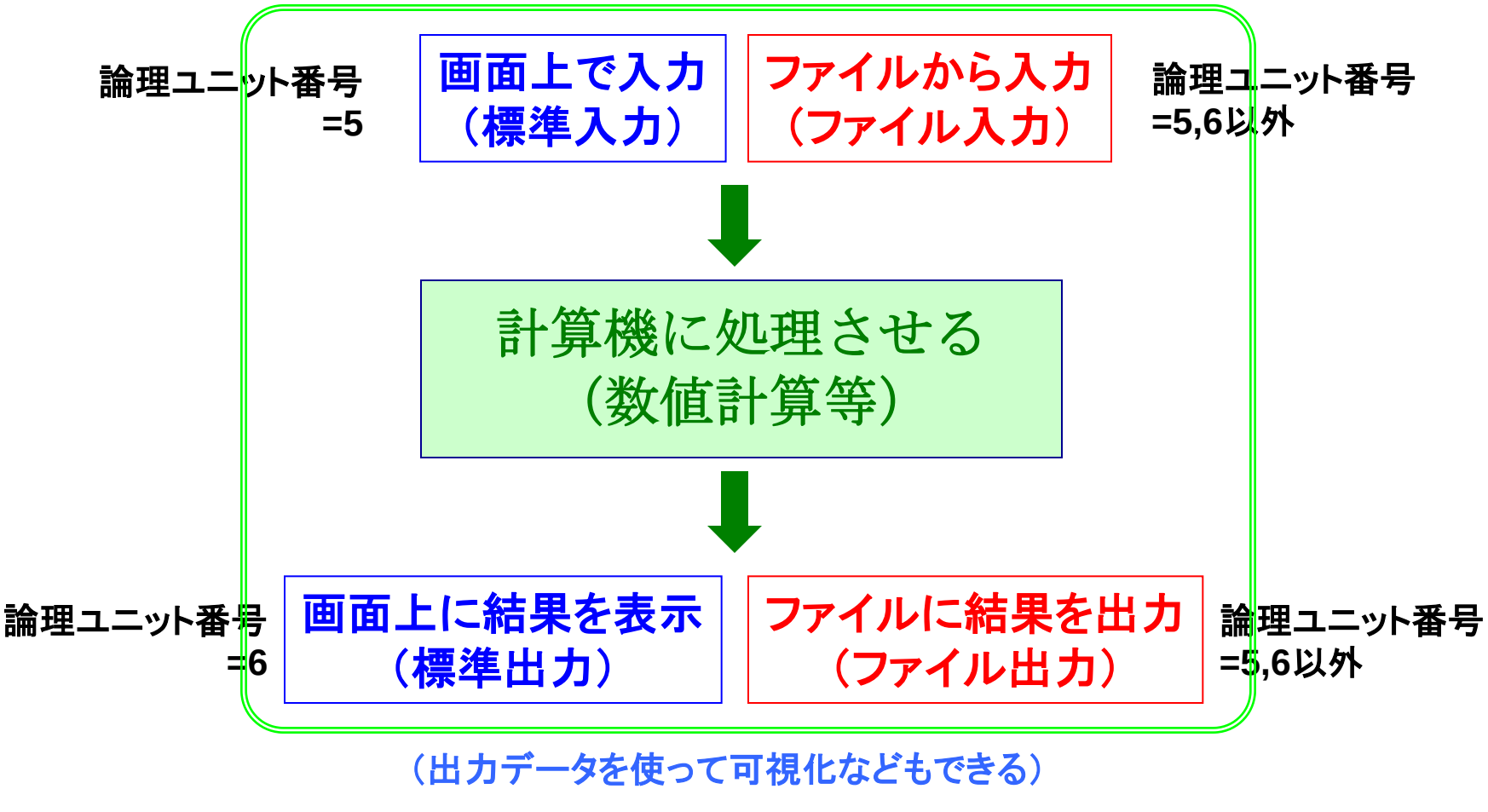
＝

```
program hello_world
  implicit none
  print *, "I live in Kobe. &
    I am going to Osaka tomorrow."
end program hello_world
```

行内の命令が長過ぎる時に便利

入出力

プログラムに入出力はつきもの



（出力データを使って可視化などもできる）

エラーの原因が入出力に帰着されることは多い。
正確な入出力は、正確な演算と同様に大事！

論理ユニット番号=各ファイルに割り当てられた背番号のようなもの

標準出力（ディスプレイ出力）

論理ユニット番号を“*” または “6”として、画面上に出力する

```
print *,    something
write(6,*)  something
```

```
print format
write(unit,format)
```

例　hello…と画面上に出力するだけの作業

```
program sample_output
  implicit none
  print *,    "hello, world, 1."
  write(6,*)  "hello, world, 2."
end program sample_output
```

実行結果

```
hello, world, 1.
hello, world, 2.
```

文字列は”〇〇”, ‘〇〇’のように囲うこと

formatを * とすると書式は指定されない
（書式については後述）

標準入力（キーボード入力）

論理ユニット番号“*”または“5”を使って画面上で入力する

```
read *,    something
read(5,*)  something
```

```
read format
read(unit,format)
```

例： ユーザが打ち込んだ整数データを画面上に出力するという作業

```
program sample_input
  implicit none
  integer :: n ←この意味は後述
  write(6,*) "n?"
  read(5,*) n
  write(6,*) n
end program sample_input
```

実行

```
% ./sample_input
 n?
2
                2
```

- 画面の待ち状態の意味が明確になるように工夫
- 大量データの入力には向かない

ファイル入力

演習a2

open文で定義した入力ファイルからread文で読み込む

5,6以外の論理ユニット番号（例：10）とファイル名（例：input）を指定

```
open(10,file="input")
read(10,*) something
```

← “背番号”を与える。ファイル名は任意

例：

```
program sample_input2
  implicit none
  integer :: n1, n2
  open(10,file="input")
  read(10,*) n1, n2
  write(6,*) n1, n2
  close(10)
end program sample_input2
```

“input”←予め作っておく

```
100 200
```

実行結果

```
% ./a.out
           100           200
```

- open文を実行せずにread(10,*)を行った場合、”fort.10”から読みこまれる
- 必要な処理が終わったら、closeすべき

演習: sample_input2.f95とinputを作成、コンパイル、実行せよ。

ファイル出力

あらかじめopen文で定義したファイルへwrite文で書き出す

5,6以外の論理ユニット番号。入力ファイル番号とも異なるように。

```
open(11,file="output")
write(11,*) something
```

← ファイル名は任意

例

```
program sample_output2
  implicit none
  integer :: n1, n2
  open(10,file="input")
  open(11,file="output")
  read(10,*) n1, n2
  write(11,*) n1, n2
  close(10)
  close(11)
end program sample_output2
```

11へ出力

“input” ←予め作っておく

```
100 200
```

実行

```
% ./a.out
```

“output”

```
        100             200
```

その他の入出力操作:リダイレクション

リダイレクション:標準の入出力先を別の入出力先に変更する
(FortranというよりはUNIXの知識)

例　標準出力をファイル(output)に書き込む

```
program hello_world
  implicit none
  print *,   "hello, world."
end program sample_output
```

標準出力

実行例

```
% ./hello_world > output
```
①

すでにoutputに何か書かれていた場合、今回の出力で上書きされる (outputファイルがない場合は、作成される。)

```
% ./hello_world >> output
```
②

古い内容の下に追加する形で出力

```
% ./hello_world >& output
```
③

エラー出力(コンパイルやプログラム実行中のエラーメッセージ等)をoutputへ

```
% (./hello_world > output) >& error
```
④

標準出力はoutputへ、エラー出力はerrorへ

標準出力内容がファイルに書き出される

その他の入出力操作:リダイレクション(2)

例　標準入力(キーボード)をファイルinputファイルに変更し、標準出力(モニタ)をファイルoutputへ書き込む

“input” ←予め作っておく

```
100 200
```

```
program sample_input3
  implicit none
  integer :: n1, n2
  read(5,*)  n1, n2          標準入力
  write(6,*) n1, n2          標準出力
end program sample_input3
```

実行例

```
% ./sample_input < input > output
```

標準出力内容をoutputへ

標準入力内容をinputから

変数

プログラム中で値を記憶しておくための機能

変数の使用

変数を使わないプログラムの例

```
program test1
  implicit none
  print *, 5
end program test1
```

実行

```
% ./a.out
                5
```

この作業しかできない、汎用性が無い

変数を使うプログラムの例

```
program test2
  implicit none
  integer :: n          型宣言
  write(6,*) "n?"
  read *, n
  write(6,*) n
end program test2
```

実行

```
% ./a.out
 n?
5
                5
```

変数nの入力次第で異なった結果が得られる

- 変数の使用でプログラムがflexibleになる
- ただし、処理を始める前に、変数の型宣言が必要
- 変数名としてのnとNは区別されない

様々な変数の型

- 基本型と派生型（構造型、ユーザーが定義）
- 基本型の種類
  - 整数型（integer）
    - 4バイト整数（-2147483648~2147483647）
    - 8バイト整数（-9223372036854775808~9223372036854775807）
  - 実数型（real）
    - 単精度実数（有効数字7桁程度）
    - 倍精度実数（有効数字16桁程度）
  - 複素数型（complex）：単精度、倍精度
  - 文字型（character）
  - 論理型（logical）
- 基本種別と拡張種別
  - 基本種別：処理系のデフォルト（上の赤であることが多い）
  - 拡張種別：種別選択子で個別指定

整数型変数

例

```
program sample_integer
  implicit none
  integer :: n
!-------------------------------
  write(6,*) "n?"
  read *, n
  write(6,*) n
end program sample_integer
```

実行

```
% ./a.out
 n?
5
                5

% ./a.out
 n?
2.6
                2
```

実数を入力しても整数と認識される（四捨五入はされない）か、コンパイラによっては”Bad integer”のエラーが出てプログラムが止まります

- -2147483648～2147483647　の整数が扱える

実数型(浮動小数点型)変数

演習a3

例

この二行は長いので、以後のスライドでは、#JISSU#と略します
皆さんは略さず、入力(もしくはコピー＆ペースト)してください

```
program sample_jissu1
!--------------------------------
  implicit none
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(SP) :: a = 1.0_SP
  real(DP) :: b
!--------------------------------
  b = 2.0_DP
  write(6,*) a
  write(6,*) b
end program sample_jissu1
```

単精度実数変数の型宣言(初期値として単精度の実数1)
倍精度実数変数の型宣言
倍精度の実数2を値を代入した

実行　　演習: sample_jissu1.f95を作成、コンパイル、実行せよ。

```
    1.0000000
    2.0000000000000000
```

- **単精度では、絶対値の範囲**:1.175494E-38～3.402823E+38
- **倍精度では、絶対値の範囲**:2.225074D-308～1.797693D+308

複素数型変数

例

```
program sample_complex
  implicit none
  #JISSU#
  complex(SP) :: i1 = (2.0_SP,1.0_SP)
  complex(DP) :: i2 = (2.0_DP,1.0_DP)
  write(6,*) i1
  write(6,*) i2
end program sample_complex
```

単精度複素数変数の型宣言（初期値2+i）

倍精度複素数変数の型宣言（初期値2+i）

$2 + i$

実行

```
% ./a.out
 (2.000000,1.000000)
 (2.000000000000000,1.000000000000000)
```

実部と虚部の二つの値からなる

文字型変数

例

```
program sample_character
  implicit none
  character(len=4) :: moji
!-------------------------------
  write(6,*) "moji?"
  read(5,*) moji
  write(6,*) moji
end program sample_character
```

4文字分の文字型変数の型宣言

k	o	b	e

s	c	h	o

o l

定義した領域に収まらない

a	b	c	

余裕がある、問題無し

実行

```
% ./a.out
 moji?
kobe
 kobe

% ./a.out
 moji?
school
 scho

% ./a.out
 moji?
abc
 abc␣
```

定義の際には必要な文字の数に注意

論理型変数

例

```
program sample_logical
  implicit none
  logical :: l1 = .true.
  logical :: l2 = .false.
  write(6,*) l1, l2
end program sample_logical
```

論理型変数の型宣言、真(.t. としてもよい)を初期値
論理型変数の型宣言、偽(.f. としてもよい)を初期値

実行

```
% ./a.out
  T   F
```

TまたはFと出力される(実践では、あまり出力することはない)

「真」か「偽」の二種類の論理値。条件の判定に使う。

型の変換

演習a4

最初に型宣言したある変数(定数)を、プログラム中で型変更する

例

```
program sample_transform
!--------------------------------
  implicit none
  #JISSU#
  real(SP) :: a = 7.7_SP
  integer  :: i = 10
!--------------------------------
  write(6,*) a, i
  write(6,*) int(a)
  write(6,*) real(i,SP), real(i,DP)
!--------------------------------
end program sample_transform
```

aを単精度実数から整数型に変換(四捨五入されない)

iを整数型から実数型に変換

実行

```
    7.6999998                  10
            7
    10.000000         10.000000000000000
```

演習: sample_transform.f95を作成、コンパイル、実行せよ。

定数

プログラム内で何度も使う定数（物理定数など）は、parameterとしてあらかじめ宣言しておくと便利

例1　定数を定義した場合

```
program sample_constant
!-------------------------------
  implicit none
  integer, parameter :: nx = 10
!-------------------------------
  write(6,*) 5 + nx
  write(6,*) 5 - nx
!-------------------------------
end program sample_constant
```

値を変えたいとき、ここだけ変えればよい

例2　定数を定義しなかった場合

```
program sample_constant2
!-------------------------------
  implicit none
!-------------------------------
  write(6,*) 5 + 10
  write(6,*) 5 - 10
!-------------------------------
end program sample_constant2
```

値を変えるには、プログラム内の全ての対応箇所を変える必要がある。プログラムの改良が大変。

フォーマット(書式)

例

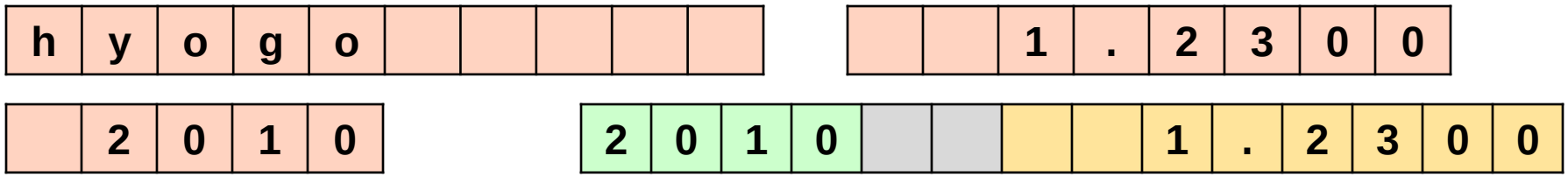
```
program sample_format
!-------------------------------
  implicit none
  #JISSU#
  character(len=10) :: moji = "hyogo"
  integer    :: i = 2010
  real(SP)   :: a = 1.23_SP
  complex(SP) :: x = (1.0_SP,2.0_SP)
!-------------------------------
  write(6,'(a5)') moji
  write(6,'(a)')  moji
  write(6,'(i5)') i
  write(6,'(f8.4)') a
  write(6,'(i4,2x,f8.4)') i, a
  write(6,'(2f8.4)') a, a
  write(6,'(2f8.4)') x
!-------------------------------
end program sample_format
```

- 文字列(a):文字の数を指定、左詰め
- 整数(i):桁数を指定、右詰め
- 実数(f):箱の数と小数点以下の桁数を指定
- 複素数(f):実数と同様、ただし二数分必要
- ブランク指定(x):ブランク数を指定

実行

```
hyogo
hyogo␣␣␣␣␣
␣2010
␣␣1.2300
2010␣␣␣␣1.2300
␣␣1.2300␣␣1.2300
␣␣1.0000␣␣2.0000
```

この場合、'(a10)'と同じ



構造体

整数型、実数型、複素数型、文字型等の基本型を複数組み合わせた複合データを構造体（構造型、派生型）として定義できる

例： 名前（文字型）と年齢（整数型）を組み合わせた構造型

```
program sample_type
!-------------------------------------------------
  implicit none

  type student
    character(len=20) :: first_name, last_name
    integer :: age
  end type student

  type(student) :: st
!-------------------------------------------------
  st = student("Albert", "Einstein", 19)
  print *, st%first_name, st%last_name, st%age
!-------------------------------------------------
end program sample_type
```

構造体の定義（文字列、文字列、整数）

stを構造体変数とした

“メンバ”に値を入れる

“メンバ”にアクセス

実行

```
 Albert                Einsten                19
```

後で見る「配列」に多少似ている

課題1

① 前述のプログラムを作成し、コンパイル＆実行せよ。

② sample_type.f95を修正し、student型の構造体変数をもう一つ（例えばst2という名前）を作り、stのデータをst2にコピーした上で、要素の一部（例えばage）を変更し、st2を出力せよ。

③ ②のプログラムとその出力結果をテキストファイル（result_170427.txt）にまとめ、
坪倉（tsubo@tiger.kobe-u.ac.jp）までメイルで送ってください。

```
% mail -s YourAccount_160512 kobeuniv.compra1@gmail.com < result_170427.txt
```

YourAccountNameには皆さん個別のアカウント名を

締切：5月2日23:59まで

演算の基礎

基本的な演算

例

```
program sample_enzan
!-------------------------------
  implicit none
  #JISSU#
  real(DP) :: a, b, c, d, e
!-------------------------------
  a = 1.0_DP + 2.0_DP          足し算
  b = 1.0_DP - 2.0_DP          引き算
  c = 1.0_DP * 2.0_DP          かけ算
  d = 1.0_DP / 2.0_DP          割り算
  e = 1.0_DP ** 2.0_DP         べき乗
  write(6,`(5(4x,a))') "  wa","  sa","seki","shou","beki"
  write(6,'(5f8.4)') a, b, c, d, e
!-------------------------------
end program sample_enzan
```

実行

```
      wa      sa     seki     shou     beki
  3.0000 -1.0000   2.0000   0.5000   1.0000
```

演算における(自動的な)型の変換

例1

```
program sample_transform2
!-------------------------------
  implicit none
  #JISSU#
  real(SP) :: a = 7.7_SP
  integer  :: i = 10
!-------------------------------
  write(6,*) a + i
!-------------------------------
end program sample_transform2
```

優先順位: 倍 > 単 > 整

実行

```
17.70000
```

単精度実数型と整数型の組み合せ
→iは単精度実数型として処理される。
a + read(i,SP)と同じ

例2

```
program sample_transform3
!-------------------------------
  implicit none
  #JISSU#
  real(DP) :: a, b, c, d
!-------------------------------
  a = 1/3                  整/整→整→倍
  b = 1.0_SP/3.0_SP        単/単→単→倍
  c = 1.0_SP/3.0_DP        単/倍→倍→倍
  d = 1.0_DP/3.0_DP        倍/倍→倍→倍
  write(6,*) a
  write(6,*) b
  write(6,*) c
  write(6,*) d
!-------------------------------
end program sample_transform3
```

実行

```
 0.000000000000000
0.3333333432674408
0.3333333333333333
0.3333333333333333
```

例2のa, bの様に、精度が落ちてしまうケースに注意！

組込み関数による演算

良く使われる関数は組み込み関数として予め用意されており、プログラム中で引用することで、自由に使うことができる

例

```
program sample_enzan2
!-------------------------------
  implicit none
  #JISSU#
  real(DP), parameter :: pi = 3.14159265358979238_DP
!-------------------------------
  write(6,*) sqrt(2.0_DP)            ルート
  write(6,'(f21.15)') sin(pi)        サイン関数
  write(6,*) exp(0.0_DP)             exponential
  write(6,*) log10(10.0_DP)          10を底とする対数
  write(6,*) mod(33,5)               あまり(33÷5=6余り3)
!-------------------------------
end program sample_enzan2
```

実行

```
    1.414213562373095
    0.000000000000000
    1.000000000000000
    1.000000000000000
                   3
```

演算の優先順位

例

```
program sample_priority
!-------------------------------
  implicit none
  #JISSU#
  real(SP) :: a, b, c, d
!-------------------------------
  a = (1.0 + 2.0)/2.0
  b = 1.0 + 2.0/2.0
  c = 1.0 + 2.0/2.0**2
  d = 1.0 + (2.0/2.0)**2
  write(6,`(4(7x,a))') "a","b", "c", "d"
  write(6,'(4f8.4)') a, b, c, d
!-------------------------------
end program sample_priority
```

カッコ内が優先 (3/2)
割り算が優先 (1+1)
冪乗が優先 (1+2/4)
カッコ内が優先 $(1+1^2)$

実行

```
       a       b       c       d
  1.5000  2.0000  1.5000  2.0000
```

条件の扱い

関係演算子と論理演算子

関係演算子

==	=（例：a==b）
>	>（例：a>b）
>=	≥（例：a>=b）
<	<（例：a<b）
<=	≤（例：a<=b）
/=	≠（例：a/=b）

論理演算子

.and.	かつ
.or.	または
.not.	否定

組み合せによる条件式の構築例

(a==b) .and. (b/=c)	a=b かつ b≠c
(a>=1 .and. a<=10)	1≤a≤10
.not. (a==b)	a=bでない場合（a/=b)

条件式は真または偽の値をとる

関係演算子（＋論理演算子）で条件式（IF）を構築し、様々な条件を判定するのに使う

If文の使用

if (条件式) で真／偽を判定する

パターン1

ある条件

```
if (条件1) 実行文
```

パターン2

ある条件

```
if (条件1) then
    実行文
    実行文
    …
end if
```

パターン3

ある条件

その他

```
if (条件1) then
    実行文
    …
else
    実行文
    …
end if
```

パターン4（何段階でもOK）

ある条件

別の条件

その他

```
if (条件1) then
    実行文
    …
else if (条件2) then
    実行文
    …
else
    実行文
    …
end if
```

If文：実行文が一つだけの場合

例　　パターン1

```
program sample_if1
!-------------------------------
  implicit none
  #JISSU#
  real(SP) :: a = 1.0_SP
!-------------------------------
  write(6,*) "a=", a
  if(a < 5.0) write(6,*) "a is smaller than 5.0."
!-------------------------------
end program sample_if1
```

条件を満たさない時は素通り

結果

```
 a=    1.000000
 a is smaller than 5.0.
```

Ifのすぐ後に実行文が書ける

If文：実行文が複数ある場合

例　パターン2

```
program sample_if2
!--------------------------------
  implicit none
  #JISSU#
  real(SP) :: a = 1.0_SP
  real(SP) :: b = 2.0_SP
!--------------------------------
  if(a < b) then
    write(6,*) "a=", a
    write(6,*) "b=", b
    write(6,*) "a is smaller."
  end if
!--------------------------------
end program sample_if2
```

- 実行文三つ
- 条件を満たさない時は何も起きない

結果

```
 a=     1.000000
 b=     2.000000
 a is smaller.
```

実行文は独立した行に書き、thenとend ifをつけること

If文：条件が分岐する場合

演習a5

例1　パターン3

```
program sample_if3
!------------------------------
  implicit none
  #JISSU#
  real(SP) :: a = 1.0_SP
  real(SP) :: b = 2.0_SP
!------------------------------
  write(6,*) a, b
  if(a > b) then
    write(6,*) "a is larger."
  else
    write(6,*) "a <= b."
  end if
!------------------------------
end program sample_if3
```

例2　パターン4

```
program sample_if4
!------------------------------
  implicit none
  #JISSU#
  real(SP) :: a = 1.0_SP
  real(SP) :: b = 2.0_SP
!------------------------------
  write(6,*) a, b
  if(a > b) then
    write(6,*) "a is larger."
  else if(a < b) then
    write(6,*) "b is larger."
  else
    write(6,*) "a = b."
  end if
!------------------------------
end program sample_if4
```

演習: sample_if3.f95を作成、コンパイル、実行せよ。

else if(条件式）かelseで複数の条件が扱える

Case文

例

整数型、論理型、または文字型
（実数型は扱えない）

```
program sample_case
  implicit none
  integer :: month

  month = 6

  select case (month)
  case (1)
    print *,'January'
  case (2)
    print *,'February'
  case (3:5)
    print *,'Spring'
  case default
    print *,'Other season'
  end select
end program sample_case
```

変数monthを判定

monthが1の場合

下限:上限（3~5 ）

その他の場合

結果

```
 Other season
```

```
case (1)
case (下限:上限)
Case (1,2,7:9,13)
…
```

一つの変数に関し、複数の条件分岐がある場合に便利

基礎事項

Implicit noneの意味

例1

```
program sample_implicit
  b = 2
  i = 3.5
  write(6,*) b
  write(6,*) i
end program sample_implicit
```

i, j,k,l, m, nで始まる変数は整数型

実行

```
     2.000000
                3
```

例2

```
program sample_implicit_no
  implicit none
  #JISSU#
  real(SP) :: b
  integer :: i
  b = 2
  i = 3.5
  write(6,*) b
  write(6,*) i
end program sample_implicit_no
```

全て明示的に宣言

実行

```
     2.000000
                3
```

Implicit noneをいつでも宣言する事

Implicit noneを書かないと、暗黙の型宣言をしたと見なされる

暗黙の型宣言の問題点

例　新鮮な肉の値段を整数型で定義したかった

```
program use_implicit_none
  integer :: fresh_meat
  flesh_meat = 100 ! yen
  print *, "today's price = ", fresh_meat
  print *, "today's price = ", flesh_meat
end program use_implicit_none
```

値を代入した（つもり）

結果

```
 today's price =   -1073743800
 today's price =     100.0000
```

- fresh_meatに（システムに依存する）default値が入っている
- flesh_meatは、暗黙の型宣言により、単精度実数型になった

どちらも望まない事

宣言したつもりのない変数を間違って使っていても気がつかない可能性がある → implicit noneを使うべき

変数の型宣言

- 基本種別の型では不十分なケース
  - 科学計算での単精度実数型
  - 超大規模計算での4バイト整数型
- 拡張種別を用いて有効桁数を上げたり利用可能整数範囲を広げる場合の問題点
  - 種別選択子（整数値）が処理系依存になっており、プログラムの汎用性が下がる
- 拡張種別の型宣言の例

  integer, parameter :: DP=倍精度実数の種別選択子

  real(DP) :: 変数名
- 拡張種別の定数定義の例

  a=1.0_DP　（定数_DPで、種別選択子DPの定数定義となる）

その他の便利機能

Stop文

例

```
program sample_stop
  implicit none
  write(6,*) "hello, world, 1."
  write(6,*) "hello, world, 2."
  stop
  write(6,*) "hello, world, 3."
end program sample_stop
```

実行

```
 hello, world, 1.
 hello, world, 2.
```

- Stopにより、プログラムの処理が完全に終了する
- デバッグ作業などで便利

セミコロンの利用

例

```
program sample_semicolon
!--------------------------------
  implicit none
  integer :: a = 1
  integer :: b = 2
!--------------------------------
  write(6,*) a; write(6,*) b
!--------------------------------
end program sample_semicolon
```

1行に命令を複数書く

実行

```
           1
           2
```

有効なケースの例

```
call sub1              ; call cpu_time(t1)
call sub2              ; call cpu_time(t2)

tmp = a; a = b; b = tmp
```

付属的な作業を右の方に書いておく（あとですぐ消せる）

aとbの交換（頭の中ではすぐ終わるが行数を費やす作業）を一行ですませる