

# Fortran90/95の基礎

## 前半

担当： 臼井英之（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://exp.cs.kobe-u.ac.jp/wiki/comp\\_practice/2010/index.php](http://exp.cs.kobe-u.ac.jp/wiki/comp_practice/2010/index.php))

# 予定

---

1. イン트로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. 副プログラム
8. 配列
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の歴史

---

- **FORTTRAN66**

- 1966年に標準化

- **FORTTRAN77**

- 1977年に標準化

- if/then/else

- 広まった

- **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

F77は今でも健在。しかし、、、

- 歴史の中で改良されてきた

- 最新版を使った方が、将来の保守という観点からもよい

Fortran90はFORTTRAN77とは大きく違う。違う言語と考えるべき。

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

# なぜFortranか？

---

## 1. 計算速度が速い

- スーパーコンピューティングは速さが命
- 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

(言語としての自由度が高いために、コンパイラが困る。  
最適化ができなくなる。)

## 2. 数値計算プログラムを書きやすい

- 道具(言語)は目的にあったものを
- 数学的計算にはFortranが適している
  - \* For-mula Tran-slator
- 数値計算ライブラリの豊富な蓄積

数式をほぼそのままプログラムとして表現できる

# 作業に必要なもの

---

- **計算機** (本fortran演習ではscalaを使います)
- **エディタ** (emacs, vi, etc. 何でもよい)
- **f95コンパイラ** (本スクールではpgf95を使います)
  - 他のコンパイラ: g95, gfortran, ifort, ...

# 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

**% pgf95 aaa.f95** → 実行ファイル“a.out”ができる

- 実行ファイルは名前を変えた方が後でわかりやすい

**% mv a.out aaa.exe**

実行ファイルの拡張子に制限はない

- コンパイル時に実行ファイル名を指定してもよい

**% pgf95 -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

## • プログラムの実行

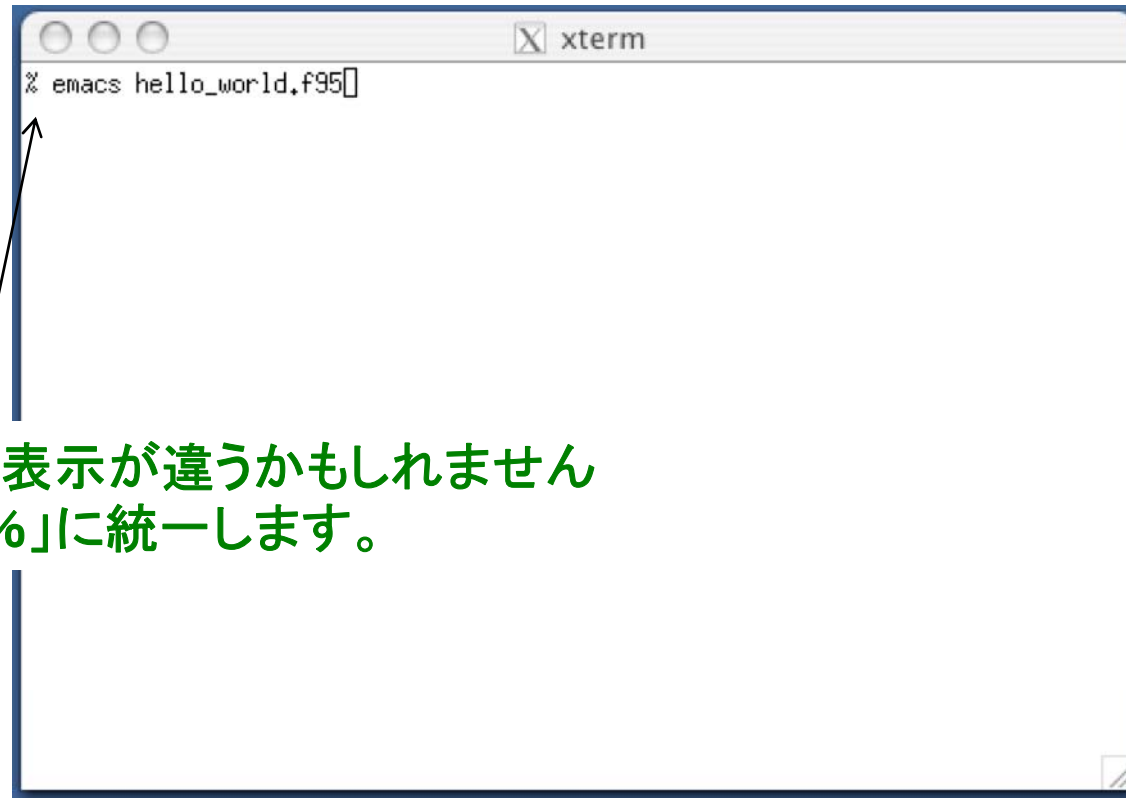
**% ./aaa.exe**

「./」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

---

- 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



```
xterm
% emacs hello_world.f95
```

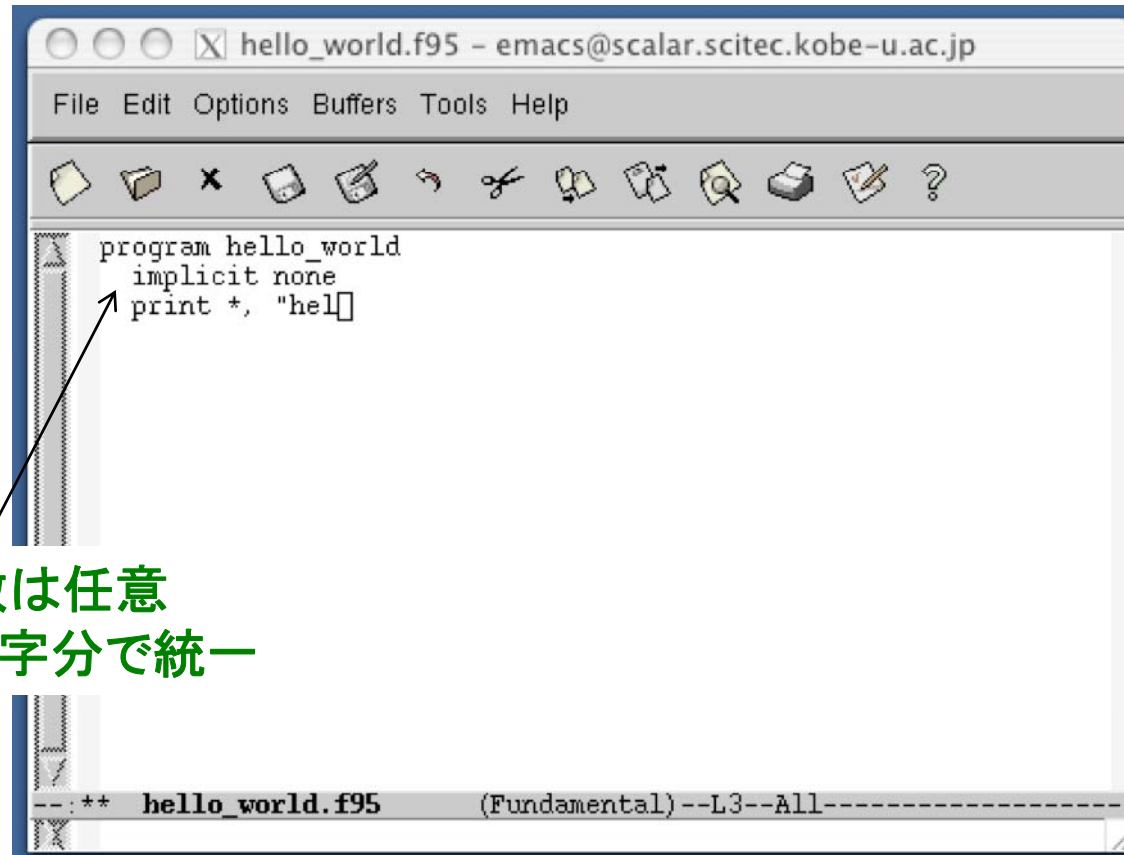
「プロンプト」の表示が違うかもしれませんが、ここでは「%」に統一します。

**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



```
program hello_world
  implicit none
  print *, "hel[]
```

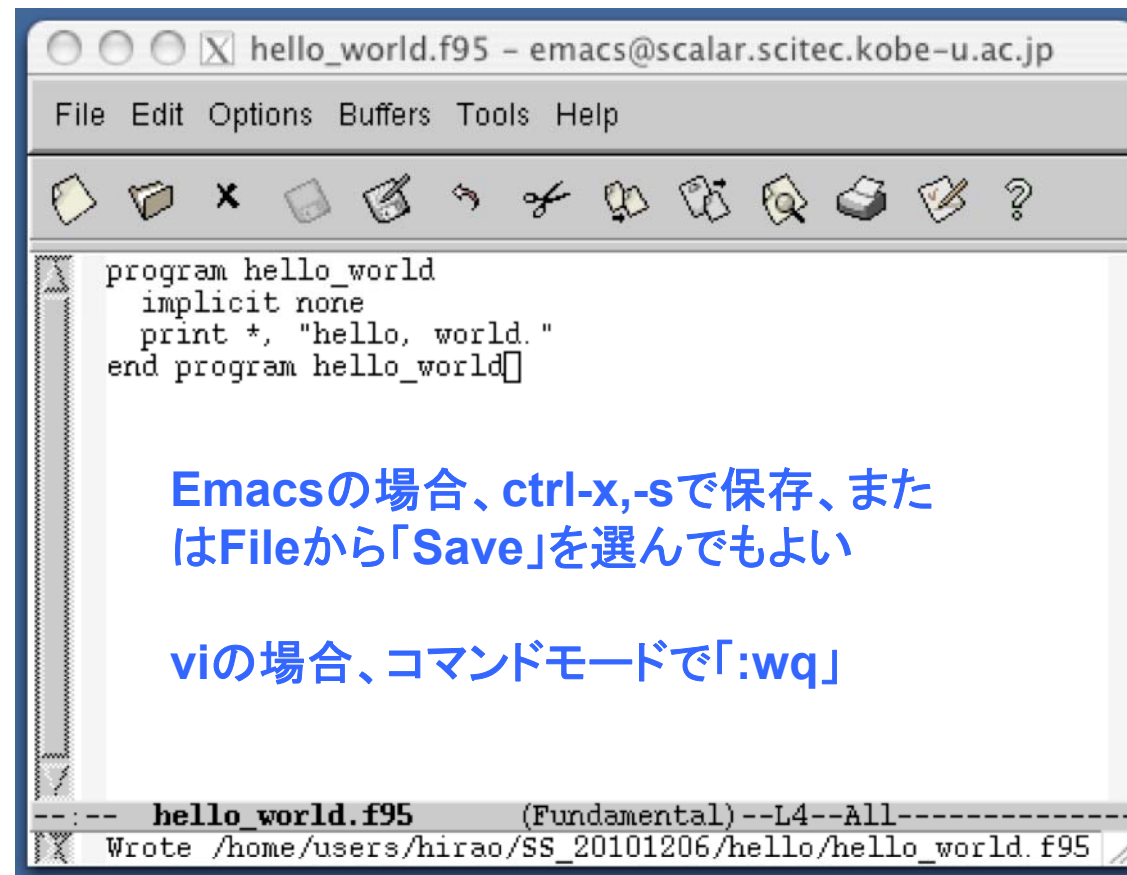
スペースの数は任意  
ここでは二文字分で統一

**演習: 以下の内容を打ち込みましょう!**

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

## Step 3. ファイルを保存する

---



The screenshot shows an Emacs editor window titled "hello\_world.f95 - emacs@scalar.scitec.kobe-u.ac.jp". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", and "Help". The toolbar contains icons for file operations. The main text area contains the following Fortran code:

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

Below the code, there are two lines of blue text providing save instructions:

Emacsの場合、ctrl-x,-sで保存、またはFileから「Save」を選んでもよい

viの場合、コマンドモードで「:wq」

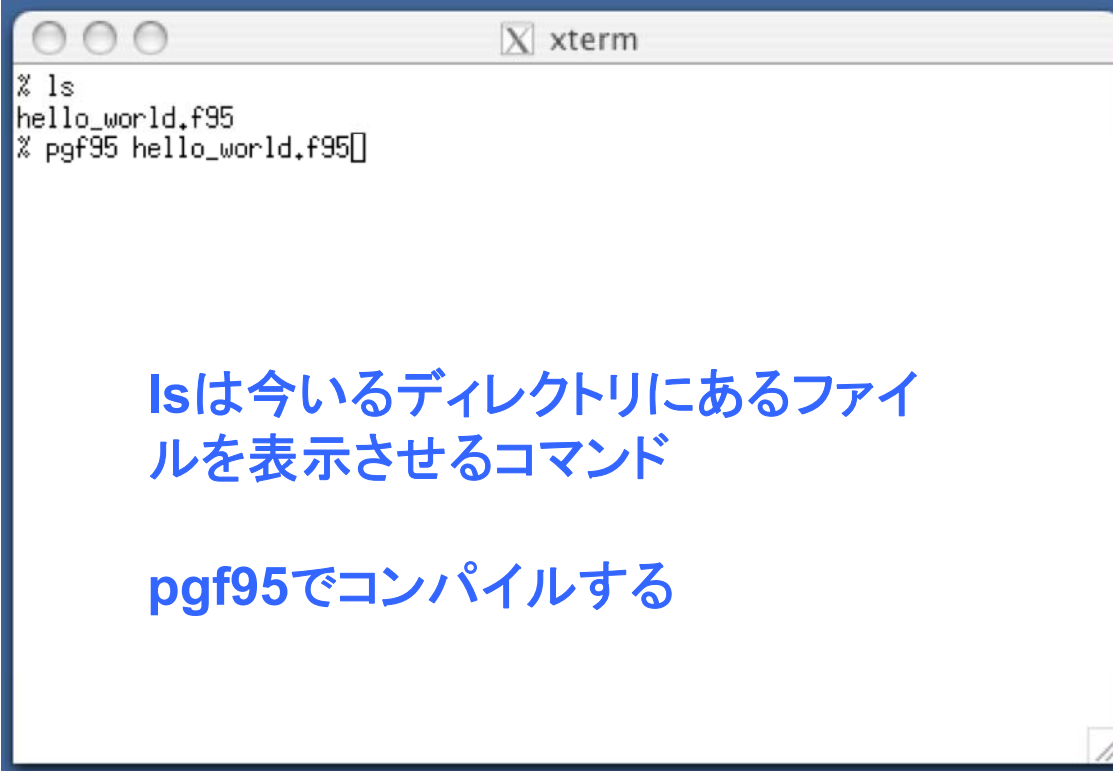
The status bar at the bottom shows "hello\_world.f95 (Fundamental) --L4--All" and a message "Wrote /home/users/hirao/SS\_20101206/hello/hello\_world.f95".

### 演習: 一緒にやってみましょう!

- 保存後、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 4. コンパイルする

---



```
xterm
% ls
hello_world.f95
% pgf95 hello_world.f95
```

lsは今いるディレクトリにあるファイルを表示させるコマンド

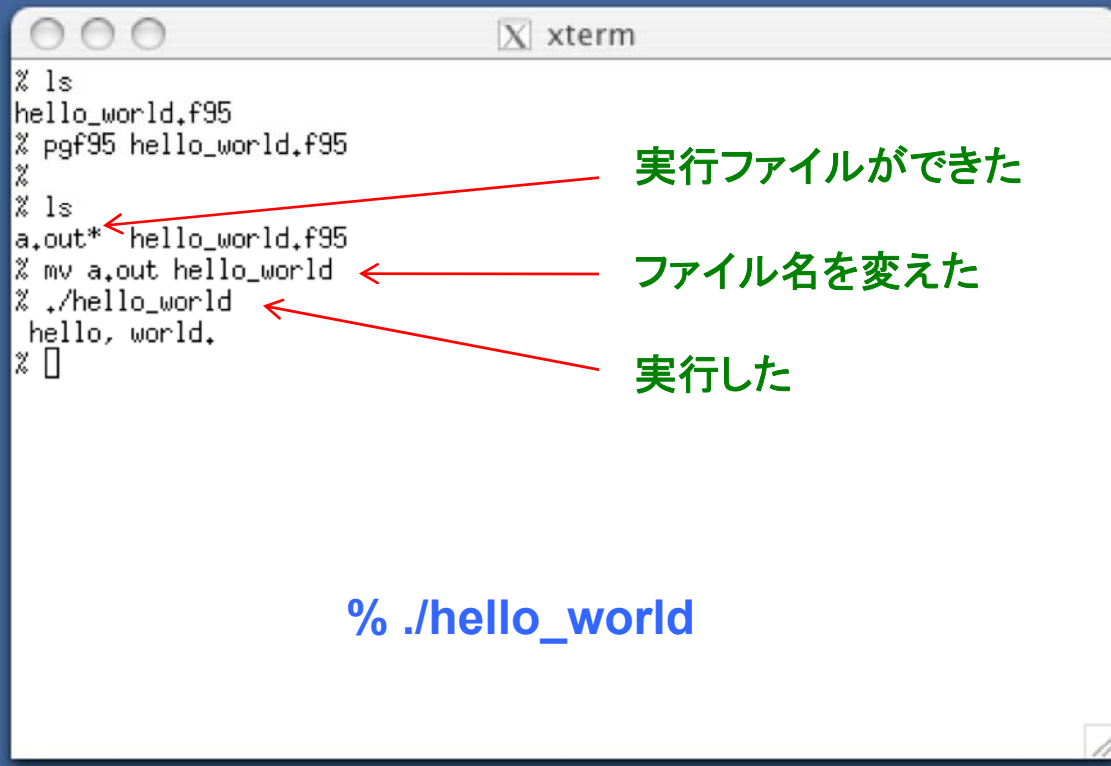
pgf95でコンパイルする

### 演習: 一緒にやってみましょう!

- プログラムをpfg95コマンドでコンパイル。
- lsを実行し、a.outというファイルができていることを確認せよ。
- mvコマンドを使って、a.outの名前を、hello\_worldに変えよ。

# Step 5. 実行する

---



```
xterm
% ls
hello_world.f95
% pgf95 hello_world.f95
%
% ls
a.out* hello_world.f95
% mv a.out hello_world
% ./hello_world
hello, world.
% 
```

実行ファイルができた

ファイル名を変えた

実行した

`% ./hello_world`

**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。

# コメントの挿入

---

## 例

```
!=====
! This is a program to say hello.
! Coded by Kobe Taro on April 28, 2011.
!=====
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
```

=

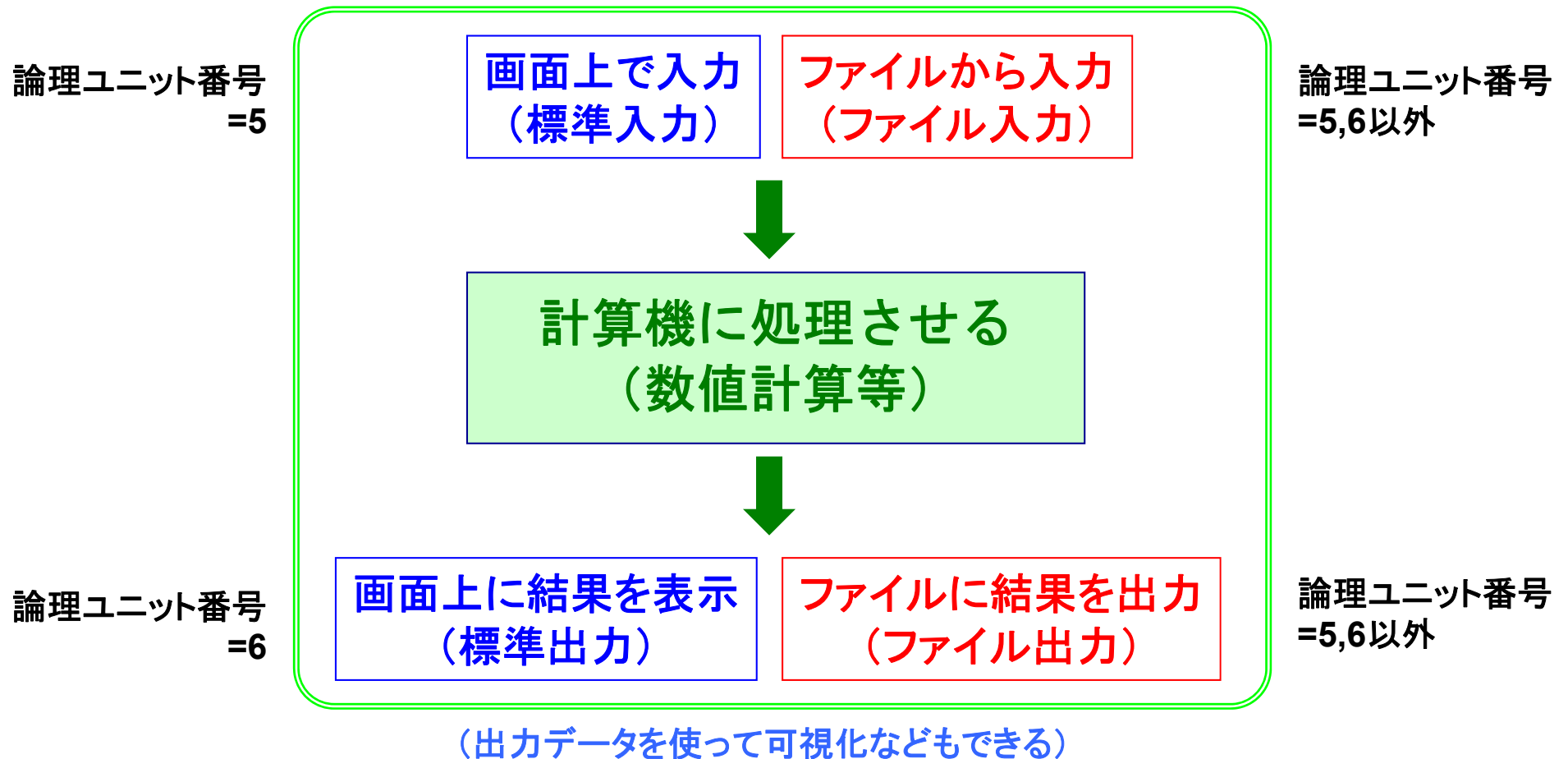
```
program hello_world  
  implicit none  
  print *, "I live in Kobe. &  
    I am going to Osaka tomorrow."  
end program hello_world
```

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



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(*,*) 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(*,*) "hello, world, 2."  
write(6,*) "hello, world, 3."  
end program sample_output
```

**実行結果**

```
hello, world, 1.  
hello, world, 2.  
hello, world, 3.
```

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(*,*) 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

**演習:** sample\_input.f95を作成、コンパイル、実行せよ。

# ファイル入力

## 5,6以外の論理ユニット番号とファイル名を指定

```
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”

```
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

**open文で定義した入力ファイルからread文で読み込む**

**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

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
```

“input”

100 200

実行

% ./a.out

“output”

100 200

11へ出力

あらかじめopen文で定義したファイルへwrite文で書き出す

# その他の入出力操作：リダイレクション

## 例

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

標準出力

## 実行例

```
% ./hello_world > output
```

```
% ./hello_world >> output
```

```
% ./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
- ② 古い内容の下に追加する形で出力
- ③ エラー出力(コンパイルのエラーメッセージ等)をoutputへ

**標準出力内容がファイルに書き出される**

演習: すでに作成したhello\_worldを使い、

1. ①→①の処理

2. ①→②の処理

を行ってそれぞれの場合のoutputの中身を確認せよ。

# その他の入出力操作：リダイレクション(2)<sup>22</sup>

---

## 例

```
program sample_input3
  implicit none
  integer :: n1, n2
  read(5,*) n1, n2
  write(6,*) n1, n2
end program sample_input3
```

標準入力  
標準出力

## “input”

```
100 200
```

## 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

---

# 変数の型

# 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
  write(6,*) "n?"
  read *, n
  write(6,*) n
end program test2
```

型宣言



## 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



# 様々な型

---

- 文字型
- 整数型
- 単精度実数型 有効桁数7程度
- 倍精度実数型 有効桁数16程度
- 複素数型
- 倍精度複素数型
- 論理型

# 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文字分の変数

k	o	b	e
---	---	---	---

s	c	h	o
---	---	---	---

a	b	c	
---	---	---	--

o |  
定義した領域に収まらない

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

# 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)

- **-2147483648～2147483647 の整数が扱える**

# 実数型（浮動小数点型）変数

## 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

```

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

```

integer, parameter :: SP = kind(1.0)

integer, parameter :: DP = selected\_real\_kind(2\*precision(1.0\_SP))

real(SP) :: a = 1.0\_SP

単精度(初期値も設定)

real(DP) :: b

倍精度

b = 2.0\_DP

値を代入した

write(6,\*) a

write(6,\*) b

**実行** 演習: sample\_jissu1.f95を作成、コンパイル、実行せよ。

```

1.000000
2.000000000000000000

```

- 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  
倍精度

$$2 + i$$

## 実行

```
% ./a.out
(2.000000,1.000000)
(2.0000000000000000,1.0000000000000000)
```

実部と虚部の二つの値からなる

# 論理型変数

---

## 例

```
program sample_logical
  implicit none
  logical :: I1 = .true.           真(.t. としてもよい)
  logical :: I2 = .false.        偽(.f. としてもよい)
  write(6,*) I1, I2
end program sample_logical
```

## 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

## 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を整数型に(四捨五入されない)  
iを実数型に変換

## 実行

7.700000	10
7	
10.00000	10.0000000000000000

最初の宣言とは異なる型を使いたい場合

演習: sample\_transform.f95を作成、コンパイル、実行せよ。

# 定数

## 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。

プログラム内で何度も使う定数をあらかじめ宣言



# 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: 文字の数を指定、左詰め
- 整数: 桁数を指定、右詰め
- 実数: 箱の数と小数点以下の桁数を指定
- 複素数: 実数と同様、ただし二数分必要

## 実行

```

hyogo
hyogo _ _ _ _ _ この場合、'(a10)'と同じ
_2010
_ _ 1.2300
2010 _ _ _ _ 1.2300
_ _ 1.2300 _ _ 1.2300
_ _ 1.0000 _ _ 2.0000

```

h	y	o	g	o					
---	---	---	---	---	--	--	--	--	--

		1	.	2	3	0	0
--	--	---	---	---	---	---	---

	2	0	1	0
--	---	---	---	---

2	0	1	0					1	.	2	3	0	0
---	---	---	---	--	--	--	--	---	---	---	---	---	---

# 構造体

## 例

```

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	Einstein	19
--------	----------	----

一つの変数を、いろいろな型を持つ要素の組み合わせとして定義できる

後で見る「配列」  
に多少似ている

# 演習 1 (構造体)

---

- ① 前述のプログラムをsample\_type.f95に保存し、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\_110428\_a.txt)にまとめ、臼井までメールで送る。

Scalarからは、

```
mail -s "exercise_110428_a" usui < result_110428_a.txt
```

---

# 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
b = 1.0_DP - 2.0_DP
c = 1.0_DP * 2.0_DP
d = 1.0_DP / 2.0_DP
e = 1.0_DP ** 2.0_DP
write(6,'5(4x,a)') " wa"," sa","seki","shou","beki"
write(6,'(5f8.4)') a, b, c, d, e
!-----
end program sample_enzan
```

足し算  
引き算  
かけ算  
割り算  
べき乗

## 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わせ  
→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
b = 1.0_SP/3.0_SP
c = 1.0_SP/3.0_DP
d = 1.0_DP/3.0_DP
write(6,*) a
write(6,*) b
write(6,*) c
write(6,*) d
!-----
end program sample_transform3

```

整/整→整→倍  
単/単→単→倍  
単/倍→倍→倍  
倍/倍→倍→倍

## 実行

0.0000000000000000  
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！

# 組み込み関数による演算

## 例

```
program sample_enzan2
!-----
  implicit none
  #JISSU#
  real(DP), parameter :: pi = 3.141592653589793238_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
0.0000000000000000
1.0000000000000000
1.0000000000000000
  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<sup>2</sup>)

## 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

## 組み合わ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

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

## 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
```

**else if(条件式)かelseで複数の条件が扱える**

# If文：論理型変数の使用

## 例

```
program sample_if5
!-----
implicit none
integer :: a = 100
logical :: large
!-----
if(a >= 50) then
  large = .true.
else
  large = .false.
endif
if(large) write(6,*) "a is large (>=50)."
!-----
end program sample_if5
```

論理型変数largeを宣言

50以上の値である場合は真

論理型変数largeが真ならば

## 実行

a is large (>=50).

If(〇〇)は、「〇〇が真ならば」を意味する

カッコ内に、真か偽を戻り値とする関数を置いてもよい

# Case文

## 例

```
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

整数型、論理型、または文字型  
(実数型は扱えない)

## 結果

Other season

一つの変数に関し、複数の条件分岐がある場合に便利



## 演習 2 (条件分岐)

---

- ① 前述のプログラムをsample\_case.f95に保存し、コンパイル & 実行せよ.
- ② sample\_case.f95を修正し、整数変数monthを標準入力から入力し、if文によって同じ判断ができるようにせよ。
- 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\_110428\_b.txt)にまとめ、臼井までメールで送る。

Scalarからは、

```
mail -s "exercise_110428_b" usui < result_110428_b.txt
```

---

# 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

```

*l,j,k,m,l,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も使っても気がつかない可能性がある → **implicit none**を使うべき

---

# 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 "stop now".
  write(6,*) "hello, world, 3."
end program sample_stop
```

表示内容は指定しなくてもよい

## 実行

```
hello, world, 1.
hello, world, 2.
stop now.
```

- **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)

```

付属的な作業を右の方に書いておく  
(あとですぐ消せる)

```

tmp = a; a = b; b = tmp

```

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