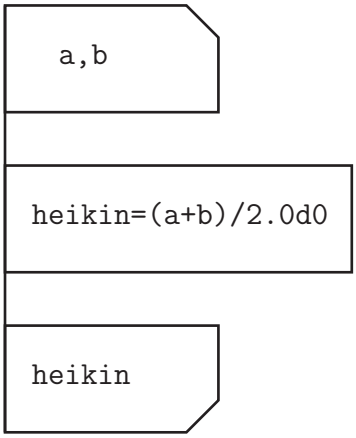
A　PADの描き方 (高野 宏)

ここでは、川合敏雄著 “PADプログラミング” (1985, 岩波書店) および二村良彦著 “プログラミング技法 –PADによる構造化プログラミング–” (1984, オーム社) を参考にPADの描き方を例を用いて説明します。なお、プログラム例中で、「宣言文」とあるのは、そこに使用全変数の宣言文を書き込むことを意味しています。紙面節約のため割愛しています。

A.1　処理箱、入力箱、出力箱、連接、ブロック

- 例題: 2個の数を読み込み、その平均を出力する
- PADの例



- PADの文法
  - ▷ 処理箱 
    処理する内容、計算式を中に書きます。
  - ▷ 入力箱 
    入力する変数を中に書きます。
  - ▷ 出力箱 
    出力する変数等を中に書きます。
  - ▷ 連接、ブロック 
    複数の処理を順番に行っていく場合は、箱を上から下へ処理の順番に従って並べ、箱の左側の辺を縦線でつなぎます。これを連接といいます。連接でつながれた複数の箱をブロックと呼びます。

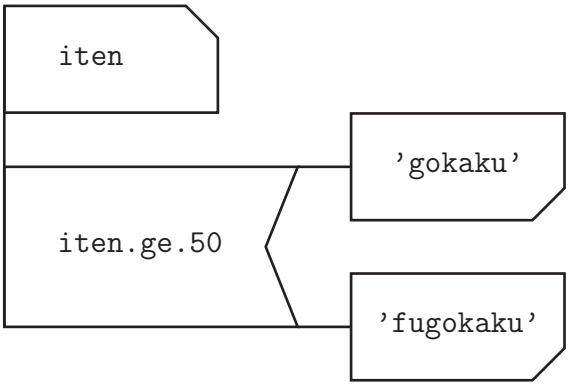
- FORTRAN への変換規則
  - ▷ 特に指定がなかったら、最初に`implicit none` を書きます。
  - ▷ 連接は上から順に変換します。
  - ▷ 処理箱はその内容をそのまま書き写します。
    処理箱の内容が、抽象的な処理を表していて、FORTRAN の文法に合わない場合は、FORTRAN の文法に合った処理に改めます。
  - ▷ 入力箱は、特に指定がなかったら、`read (*,*) 入力箱の内容`と書きます。
  - ▷ 出力箱は、特に指定がなかったら、`write (*,*) 出力箱の内容`と書きます。
  - ▷ 特に指定がなかったら、最後に
    ```
    stop
    end
    ```
    を書きます。

- FORTRAN プログラム

```
implicit none
宣言文 (例：real*8 a,b,heikin)
read(*,*) a,b
heikin=(a+b)/2.0d0
write(*,*) heikin
stop
end
```

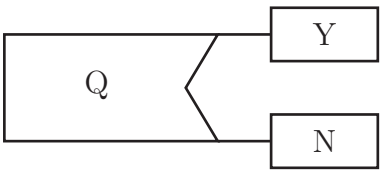
A.2 選択箱

- 例題: 点数を読み込み、50 点以上なら gokaku 、そうでなければ fugokauo と出力する
- PAD の例



- PAD の文法

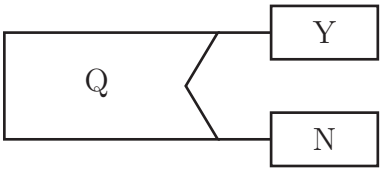
  ▷ 選択箱 
  もし条件 Q が成立すればブロック Y の処理を実行し、成立しなければブロック N の処理を実行する場合、PAD では次のように書きます。



  これを選択といいます。N の処理がない場合もあります。

- FORTRAN への変換規則

  ▷ 選択



  は
```
if (Q) then
   Y
else
   N
endif
```
  と変換します。

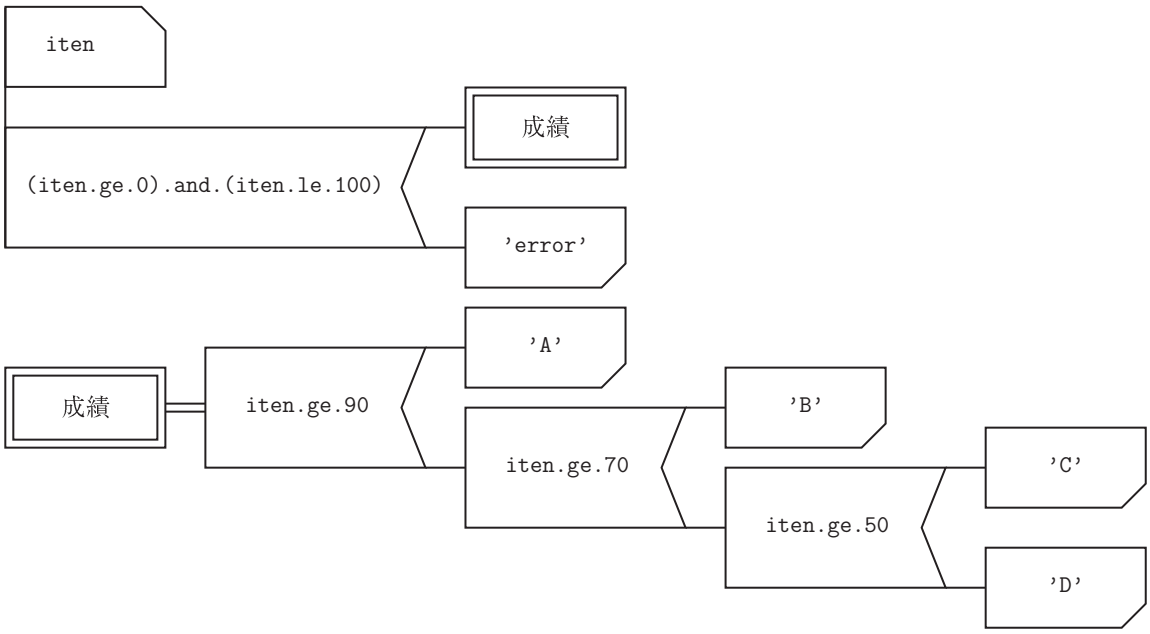
  N のブロックがないときは、
```
if (Q) then
   Y
endif
```
  と変換します。

- FORTRAN プログラム

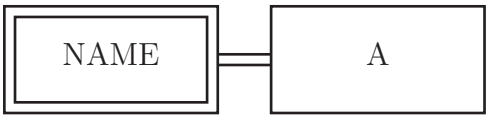
```
implicit none
宣言文
read(*,*) iten
if (iten.ge.50) then
  write(*,*) 'gokaku'
else
  write(*,*) 'fugokaku'
endif
stop
end
```

A.3 定義箱、参照箱

- 例題: 点数を読み込み、点数が 0 点以上 100 点以下のときに成績を出力する。点数がそれ以外の場合は error と出力する。成績は 90 点以上なら A、90 点未満 70 点以上なら B、70 点未満 50 点以上なら C、50 点未満なら D とする。
- PAD の例



- PAD の文法
  - ▷ 定義箱 
    ブロック A に名前 NAME を付ける場合、次のように書きます。



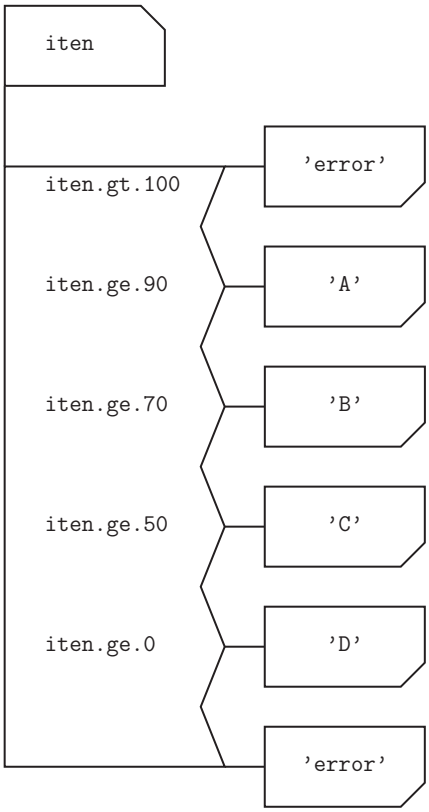
  - ▷ 参照箱 
    定義箱で別に定義したブロックを引用するときに使います。名前 NAME のブロックを引用するには次のように書きます。

- FORTRAN への変換規則
  - ▷ 参照箱は定義箱で定義されたブロックに置き換えてから変換します。
- FORTRAN プログラム

```
implicit none
宣言文
read(*,*) iten
if ((iten.ge.0).and.(iten.le.100)) then
  if (iten.ge.90) then
    write(*,*) 'A'
  else
    if (iten.ge.70) then
      write(*,*) 'B'
    else
      if (iten.ge.50) then
        write(*,*) 'C'
      else
        write(*,*) 'D'
      endif
    endif
  endif
else
  write(*,*) 'error'
endif
stop
end
```

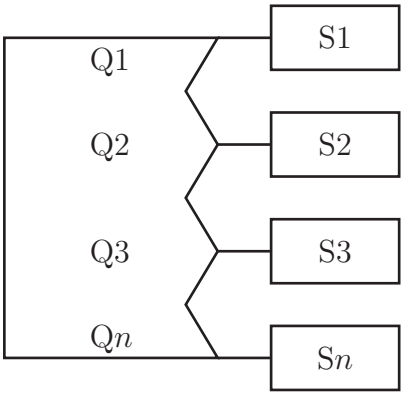
A.4 多枝選択箱

- 例題: 点数を読み込み、点数が 0 点以上 100 点以下のときに成績を出力する。点数がそれ以外の場合は error と出力する。成績は 90 点以上なら A、90 点未満 70 点以上なら B、70 点未満 50 点以上なら C、50 点未満なら D とする。
- PAD の例



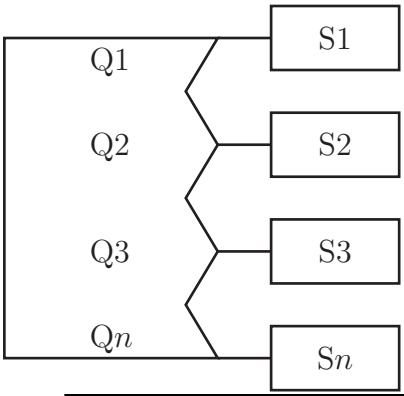
- PAD の文法
  - ▷ 多枝選択箱 

    n 個の条件 Q1, Q2, ···, Qn に n 個の処理 S1, S2, ···, Sn を対応させる。n 個の条件 Q1, Q2, ···, Qn を順番に調べていって、最初に成立した条件に対応した処理 1 つだけを選んで実行するには、次のように書きます。



最後の条件 Qn がその他の場合を意味するときは、Qn を省略できます。

- FORTRAN への変換規則
  - ▷ 多枝選択箱



は、

```
if (Q1) then
   S1
else if (Q2) then
   S2
      ⋮
else if (Qn) then
   Sn
endif
```

と変換します。

Qn が省略されたときは、

```
if (Q1) then
   S1
else if (Q2) then
   S2
      ⋮
else
   Sn
endif
```

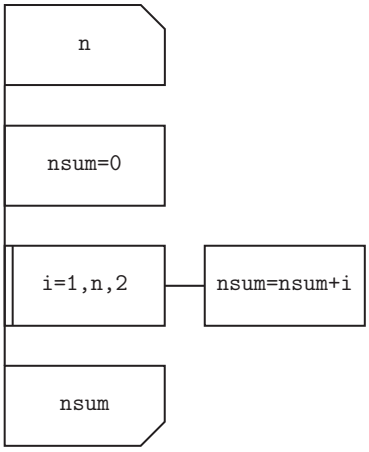
と変換します。

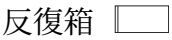
- FORTRAN プログラム

```
implicit none
宣言文
read(*,*) iten
if (iten.gt.100) then
  write(*,*) 'error'
else if (iten.ge.90) then
  write(*,*) 'A'
else if (iten.ge.70) then
  write(*,*) 'B'
else if (iten.ge.50) then
  write(*,*) 'C'
else if (iten.ge.0) then
  write(*,*) 'D'
else
  write(*,*) 'error'
endif
stop
end
```

A.5 反復箱

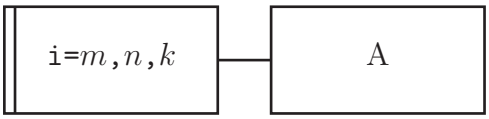
- 例題: 整数 n を読み込み、1 から n までの奇数の和を求める。
- PAD の例



- PAD の文法
  - ▷ 反復箱 
    i について m から n まで k おきにブロック A の処理を繰り返すには、次のように書きます。

    $k = 1$ のときは、,k を省略できます。

- FORTRAN への変換規則
  - ▷ 反復箱



    は、

```
        do 文番号 i=m,n,k
          A
文番号  continue
```

    と変換します。

    ,k が省略されたときは、

```
        do 文番号 i=m,n
          A
文番号  continue
```

    と変換します。

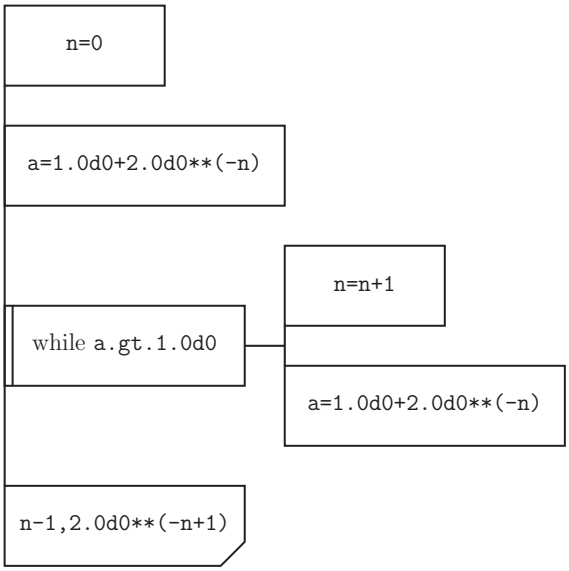
- FORTRAN プログラム

```
implicit none
宣言文
read(*,*) n
nsum=0
do i=1,n,2
  nsum=nsum+i
end do
write(*,*) nsum
stop
end
```

A.6 while 箱

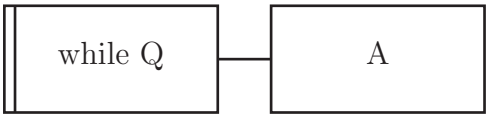
- 例題: 倍精度実数で $1+2^{-n}$ を計算し、それが倍精度実数の 1 より大きいような最大の整数 n を求め、n と 2^{-n}(計算機イプシロン) を出力する。

- PAD の例

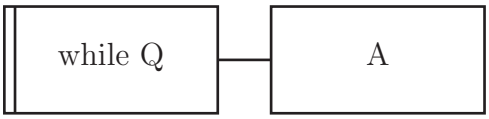


- PAD の文法

  ▷ while 箱 
  条件 Q が成立している間は、ブロック A の処理を繰り返すには、次のように書きます。



- FORTRAN への変換規則
  - ▷ while 箱



は、

```
文番号  continue
        if (Q) then
          A
          goto 文番号
        endif
```

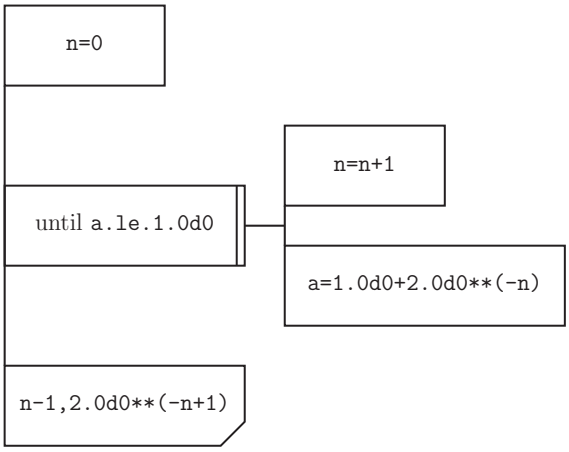
と変換します。

- FORTRAN プログラム

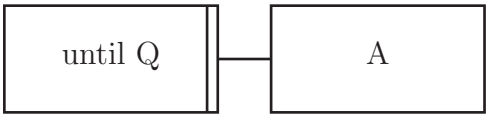
```
      implicit none
      宣言文
      n=0
      a=1.0d0+2.0d0**(-n)
 1000 continue
      if (a.gt.1.0d0) then
        n=n+1
        a=1.0d0+2.0d0**(-n)
        goto 1000
      endif
      write(*,*)n-1,2.0d0**(-n+1)
      stop
      end
```

A.7 until 箱

- 例題: 倍精度実数で $1+2^{-n}$ を計算し、それが倍精度実数の 1 より大きいような最大の整数 n を求め、n と 2^{-n}(計算機イプシロン) を出力する。
- PAD の例



- PAD の文法
  - ▷ until 箱 
    条件 Q が成立するまでブロック A の処理を繰り返すには、次のように書きます。



- FORTRAN への変換規則
  - ▷ until 箱

は、

```
文番号  continue
        A
        if (.not.(Q)) goto 文番号
```

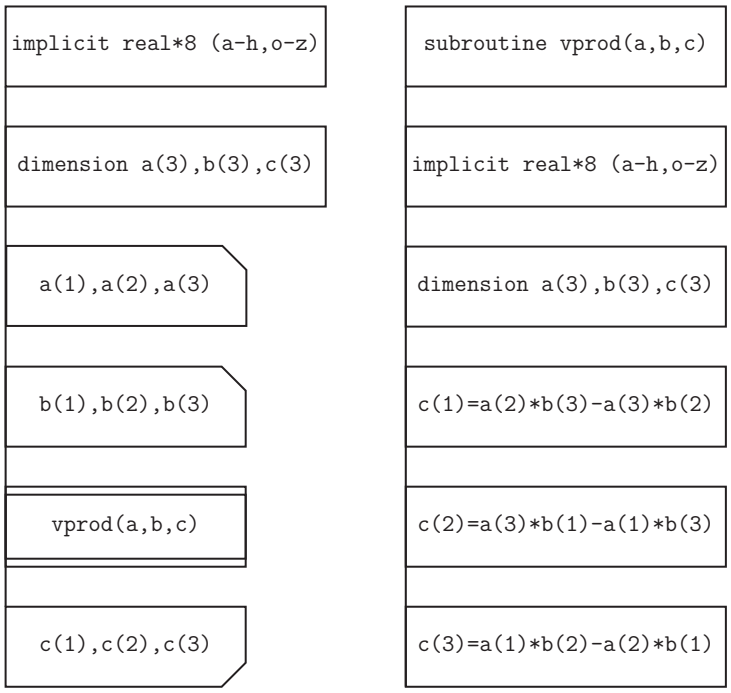
と変換します。

- FORTRAN プログラム

```
      implicit none
      宣言文
      n=0
 1000 continue
      n=n+1
      a=1.0d0+2.0d0**(-n)
      if (.not.(a.le.1.0d0)) goto 1000
      write(*,*) n-1,2.0d0**(-n+1)
      stop
      end
```

A.8 呼び出し箱

- 例題: 2 つの 3 次元ベクトルの x, y, z 成分を読み込み、それらのベクトル積の x, y, z 成分を出力する。
- PAD の例



- PAD の文法
  - ▷ 呼び出し箱 
    名前 NAME のサブルーチンを呼び出すには次のように書きます。

名前 NAME のサブルーチンは別に書いておきます。

▷ サブルーチンは最初の処理箱に subroutine 文を書きます。

- FORTRAN への変換規則

  ▷ 呼び出し箱

  は、`call NAME` と変換します。

  ▷ サブルーチンは、特に指定がなかったら、
  subroutine 文の次に`implicit none` を書きます。

  ▷ サブルーチンは、特に指定がなかったら、最後に
  ```
  return
  end
  ```
  を書きます。

- FORTRAN プログラム

```
      implicit none
      宣言文
      dimension a(3),b(3),c(3)
      read(*,*) a(1),a(2),a(3)
      read(*,*) b(1),b(2),b(3)
      call vprod(a,b,c)
      write(*,*) c(1),c(2),c(3)
      stop
      end
c
      subroutine vprod(a,b,c)
      implicit real*8 (a-h,o-z)
      dimension a(3),b(3),c(3)
      c(1)=a(2)*b(3)-a(3)*b(2)
      c(2)=a(3)*b(1)-a(1)*b(3)
      c(3)=a(1)*b(2)-a(2)*b(1)
      return
      end
```

B FORTRAN プログラムの書き方 (高野 宏)

ここでは、FORTRAN プログラムの書き方 (文法) を簡単に紹介します。詳しくは FORTRAN の文法書や教科書を参照して下さい。

- 使える文字

  英数字　英字　A から Z のアルファベット
  　　数字　0 から 9 の数字
  特殊文字　　空白　= + - * / ( ) , . ' : $

- プログラムは行を順番に並べたものとして書かれます。
  一行は次のように書かれます。

  プログラムを書く欄
  桁数

  1 桁目: 空白か C か*を書きます。ここが C か*の時は、
  その行は注釈行で、プログラムに関するメモ等を書きます。

  2–5 桁目: 文番号を書きます。

  6 桁目: [空白または 0] 以外の文字がある時、その行は継続行となります。
  継続行の 7–72 桁目は直前の行の 72 桁目の続きになります。
  継続行の 1–5 桁目は空白でなければなりません。

  7–72 桁目: 文を書きます。

- 文には、例えば次のようなものがあります。

  実行文: 代入文、GOTO 文、IF 文、ELSE 文、ENDIF 文、CONTNUE 文、
  STOP 文、DO 文、入出力文、CALL 文、RETURN 文等。

  非実行文: FUNCTION 文、SUBROUTINE 文、DIMENSION 文、
  IMPLICIT 文、PAREMETER 文、型宣言文等。

- プログラムは文と注釈行からなります。
  プログラムの一つの単位は、主プログラムまたは副プログラムと呼ばれます。
  プログラムの一つの単位の終りには END 文がきます。
  一つの実行可能なプログラムは、1 個の主プログラムと 0 個以上の副プログラムから構成されます。
  主プログラムは PROGRAM 文で始まります。
  副プログラムには FUNCTION 文で始まる関数副プログラムや、
  SUBROUTINE 文で始まるサブルーチン副プログラムがあります。

- 文は普通、プログラムの一つの単位の中で、次の順に並べます。

  主プログラムの場合は、PROGRAM 文
  副プログラムの場合は、FUNCTOIN 文や SUBROUTINE 文
  IMPLICIT 文
  型宣言文
  PARAMETER 文
  DIMENSION 文
  実行文
  END 文

- 普通は、プログラム中の実行文が、プログラム中で現れる順に実行されていきます。GOTO 文、IF 文、DO 文等により、この実行順が変更されます。

- FORTRAN で扱うデータには型と長さがあります。
  一つのデータが計算機の中で何ビットで表わされているかをデータの長さといいます。
  データの型とその長さには、次のようなものがあります (1 バイトは 8 ビット)。

  整数型: 4 バイト (標準)
  実数型: 4 バイト (標準、単精度)、8 バイト (倍精度)
  複素数型: 8 バイト (標準、単精度)、16 バイト (倍精度)
  文字型: 1 バイト (標準)、バイト数として正の整数を指定可能
  論理型: 4 バイト (標準)

  それぞれのデータの型と長さをもつような、定数、変数、配列を使うことができます。

- 定数は、例えば次のように表わします。

  整数型の定数:

```
 16
+25
  0
-36
```

  実数型の定数:

```
0.16D+2      0.16 × 10^+2 を表わします
1.0D0        1.0 × 10^0 を表わします
+2.5D1       2.5 × 10^1 を表わします
-360.0D-1    −36.0.0 × 10^−1 を表わします
```

  これらは、倍精度実数を表わします。
  D を E に置き換えたものは単精度実数を表わします。

複素数型の定数:

二つの実数型の定数をカンマで区切って括弧で括ったもので、
`(0.16D+2, -360.0D-1)`
は、 $0.16 \times 10^2 - 360.0 \times 10^{-1}\mathrm{i}$ を表わします。

文字型の定数:

アポストロフィで文字列をくくったものです。
文字列中に一つのアポストロフィを入れる時は、
二つのアポストロフィを書きます。
`'DON''T'`

論理型の定数:

`.TRUE.` 真を表わします。
`.FALSE.` 偽を表わします。

- 変数は、先頭が英字で他が英数字であるような 1–6 文字の名前 (英字名と呼びます) をもっています。
  その英字名を変数名と呼びます。
  変数に値を代入したり、変数の値を使って計算をしたりすることができます。
  変数の型と長さは、IMPLICIT 文や型宣言文であらかじめ決めておきます。

- 配列は、いくつかのデータを順序をつけて並べたものです。
  配列は英字名をもっています。
  その英字名を配列名と呼びます。
  配列内の一つのデータを配列要素といい、配列の名前に添字をつけたもので表わします。
  添字は、整数や整数を表わす式 (添字式といいます) をカンマで区切って並べ括弧でくくったものです。
  例えば、`A(2)` は `A` という配列の中の添字 `(2)` に対応した配列要素を表わします。
  配列が添字の中にいくつの添字式をもつかを、配列の次元といいます。
  例えば、`B(3,4)` は `B` という 2 次元配列の中の添字 `(3,4)` に対応した配列要素を表わします。
  配列の型と長さは、IMPLCIT 文や型宣言文であらかじめ決めておきます。
  配列の次元や、添字式のとりうる値の範囲は、DIMENSION 文であらかじめ決めておきます。

- PARAMETER 文で定数に英字名をつけることができます。
  その英字名を定数名と呼びます。
  定数名の型と長さは、IMPLCIT 文や型宣言文であらかじめ決めておきます。
  定数名を定数のかわりに使うことができます。

- 変数名、配列名、定数名はプログラムの一つの単位の中でのみ有効です。
- 数値の計算およびその結果を表わすのに、算術式を使います。
  算術式は、定数、変数、配列要素、関数等に対して

  `+` 加算
  `-` 減算
  `*` 乗算
  `/` 除算
  `**` べき乗

  等の算術演算子や () 括弧を使って計算の式を書いたものです。算術演算子の優先順位 (一つの式の中で演算を行う順序) は数学の普通の表記法と同じです。

  例:

  ```
  A(2)+B*(C-1.0D1)/SIN(E)**2
  ```

- 演算子を含んだ算術式の型と長さは、演算結果の型と長さになります。
  演算結果の型は、演算されるものの型の内、優先順位の高いものになります。
  優先順位は複素数型、実数型、整数型の順です。
  演算結果の長さは、長い方になります。
  例えば、C8 を単精度複素数、R8 を倍精度実数、I4 を整数とするとき、

  `I4+R8` は倍精度実数に
  `R8+C8` は倍精度複素数になります。

- 算術式と算術式の比較等をするために、算術関係式を使います。
  算術関係式は、算術式と算術式の間に関係演算子を書いたもので、その結果は論理型です。
  関係演算子には次のようなものがあります。

  `.LT.` Less Than
  `.LE.` Less than or Equal to
  `.EQ.` EQual to
  `.NE.` Not Equal to
  `.GT.` Greater Than
  `.GE.` Greater than or Equal to

  例:

  ```
  1.0D+1.LT.20.0D0    結果は.TRUE.
  A.GE.B
  ```

- 論理演算子およびその結果を表わすのに、論理式を使います。
  論理演算子には、例えば、次のようなものがあります。

  .NOT.
  .AND.
  .OR.

  論理演算子の優先順位はこの順です。

  例:

  (A.GT.B).OR.(A.EQ.B)　これはA.GE.Bと同じです。

- 算術演算子、関係演算子、論理演算子の順に高い優先順位をもっています。

- 変数や配列の型と長さは、IMPLICIT文や型宣言文で指定しない限り、次の規則で決まります。
  英字名の最初がI, J, K, L, M, Nのものは4バイトの整数型になります。
  英字名の最初が他の文字のときは4バイトの実数型になります。

- IMPLICIT文は、変数や配列の型と長さをまとめて指定するための文です。

  書き方:

  IMPLICIT 型名*バイト数（英字や英字-英字をカンマで区切って並べたもの）
  英字-英字は、アルファベット順でその範囲にある英字をカンマで区切ってならべたものと同じです。
  型名は 整数型は　INTEGER
  実数型は　REAL
  複素数型は　COMPLEX
  文字型は　CHARACTER
  論理型は　LOGICAL　です。
  これによって、()内の英字で始まる英字名の型と長さが指定されます。

  例:

  IMPLICIT REAL*8 (A-H,O-Z)
  英字名の最初がAからH迄、またはOからZ迄のものが8バイトの実数型になります。

- 型宣言文は個々の変数や配列の型と長さを指定するための文です。

  書き方:

  型名*バイト数 英字名をカンマで区切ってならべたもの

  例:

  CHARACTER*20 STRING
  英字名STRINGが20文字(20バイト)の文字型になります。

型宣言文の方が、IMPLICIT 文より優先します。

- PARAMETER 文は定数に英字名を与えるために使います。

  書き方:
  PARAMETER(英字名=定数 をカンマで区切って並べたもの)

  例:
  PARAMETER(ZERO=0.0D0,ONE=1.0D0)
  これによって、定数名 ZERO を定数 0.0D0 のかわりに、定数名 ONE を定数 1.0D0 のかわりに使うことができます。

- DIMENSION 文は配列の次元と添字式の取りうる範囲を指定します。

  書き方:
  DIMENSION 配列名 (添字式の最小値:添字式の最大値 をカンマで区切ってならべたもの)
  添字式の最小値が 1 の時は、添字式の最小値:は省略できます。

  例:
  DIMENSION A(-10:10,-10:10)
  これによって、A が 2 次元の配列になります。
  DIMENSION B(20)
  これは、
  DIMENSION B(1:20)
  と同じ意味で、B が 1 次元の配列になります。

- 代入文は、変数や配列要素に値を代入するために使います。

  書き方:
  変数名や配列要素名=式

  例:
  A=A+1.0D0
  これにより、現在の A の値に 1.0 を加えたものが新しい A の値となります。

- CONTINUE 文は、何も行いません。
  CONTINUE 文は、文番号をつけた何もしない実行文として、例えば、DO 文や GOTO 文と組み合わせて、使われます。

  書き方:
  CONTINUE

  例:
  ␣␣100␣CONTINUE
  ␣は空白を表わします。100 は文番号です。

- GOTO 文はプログラムの実行の順番を変えるために使います。

  書き方:

  GOTO 文番号

  これが実行された後、文番号で指定された文から実行が始まります。

  例:

  ␣␣␣␣␣␣GOTO␣100
  ␣␣200␣CONTINUE
  ␣␣␣␣␣␣A=B
  ␣␣100␣CONTINUE
  ␣␣␣␣␣␣A=C

  GOTO 100 が実行されると、A=B は実行されず A=C から実行が始まります。

  注:

  GOTO 文の飛び先の文は、CONTINUE 文にしておくと便利です。

- ブロック IF 文は、条件によってプログラムの実行する文を変えるために使います。

  書き方:

  IF（論理式 1）THEN
  　ブロック 1
  ENDIF

  これにより、論理式 1 が.TRUE. の時、ブロック 1 が実行されます。

  IF（論理式 1）THEN
  　ブロック 1
  ELSE
  　ブロック 2
  ENDIF

  これにより、論理式 1 が.TRUE. の時、ブロック 1 が実行され、それ以外の時、ブロック 2 が実行されます。

  IF（論理式 1）THEN
  　ブロック 1
  ELSE IF（論理式 2）THEN
  　ブロック 2
  ELSE
  　ブロック 3
  ENDIF

  これにより、論理式 1 が.TRUE. の時、ブロック 1 が実行され、論理式 1 が.FALSE. で論理式 2 が.TRUE. の時、ブロック 2 が実行され、それ以外

の時、ブロック 3 が実行されます。

例:

```
IF (I.EQ.0) THEN
  A=0.0D0
  B=0.0D0
ELSE IF (I.LT.0) THEN
  A=1.0D0
  B=1.0D0
ELSE
  A=2.0D0
  B=2.0D0
ENDIF
```

- DO 文は、ある範囲の文を繰り返し実行するために使います。

書き方:

DO 文番号 変数名=初期値, 終値, 増分

この文の次の文から、文番号の文までを、DO ループの範囲といいます。
これにより、まず変数に初期値が代入されます。
(終値 − 初期値 + 増分)/増分 を切り捨てて整数にしたものと 0 とを比較し、大きい方を繰り返し回数とします。
繰り返し回数が 0 でないあいだ、
　DO ループの範囲を実行し、
　変数の現在の値に増分を加え
　繰り返し回数を 1 減らす
ということを繰り返します。
増分が 1 の時は、, 増分 は省略できます。

例:

```
␣␣␣␣␣␣DO␣100␣I=1,10
␣␣␣␣␣␣␣␣A(I)=B(I)
␣␣100␣CONTINUE
```

注:

DO 文の文番号で指定する文 (DO ループの最後の文) は、普通 CONTINUE 文にします。

- 入出力文は、変数に外部から値を与えたり (入力)、変数の値を書き出したり (出力) するために使います。

  書き方:

  READ（*,*）変数名や配列要素名をカンマで区切って並べたもの
  これにより、変数や配列要素に値が読み込まれます。

  WRITE（*,*）変数名や配列要素名をカンマで区切って並べたもの
  これにより、変数や配列要素の値が出力されます。

  例:

```
READ（*,*）A,B
WRITE（*,*）'A=',A
WRITE（*,*）'B=',B
```

- ファイルから値を読み出したり、ファイルに値を書き出したりする場合には、次のようにします。
  まず、OPEN 文で入出力を行うファイルを指定します (開きます)。

  書き方:

  OPEN（0 から 99 迄の整数式，FILE=ファイル名を表わす文字型データ）
  これにより、指定したファイル名のファイルが、指定した整数式の番号の装置として開かれます。

  次に、READ 文や WRITE 文で、この番号の装置に対して読み書きを行います。

  書き方:

  READ（装置の番号を表わす整数式,*）変数名や配列要素名をカンマで区切って並べたもの
  これにより、指定した番号の装置から、変数や配列要素に値が読み込まれます。

  WRITE（装置の番号を表わす整数式,*）変数名や配列要素名をカンマで区切って並べたもの
  これにより、変数や配列要素の値が、指定した番号の装置に出力されます。

  最後に、CLOSE 文で使い終わったファイルを指定します (閉じます)。

  書き方:

  CLOSE（装置の番号を表わす整数式）
  これにより、指定した番号の装置に対応したファイルが閉じられ、そのファイルとその装置番号との対応もなくなります。

注:
普通、READ (*,*) は READ (5,*) を、WRITE (*,*) は WRITE (6,*) を表わしています。

ファイルは文字を並べたものと考えられます。
次に読み書きする文字の位置は、読み書き行うごとに順に動いて行きます。
REWIND 文は、次に読み書きする文字の位置を、ファイルの先頭に戻します。

書き方:
REWIND 装置の番号を表わす整数式

例:
```
OPEN (10,FILE='TEST')
WRITE (10,*) A,B,C
REWIND 10
READ (10,*) X,Y,Z
CLOSE (10)
```

- STOP 文はプログラムの実行を終了するための文です。

書き方:
STOP

- FORTRAN には、あらかじいくつかの関数が、組み込み関数として、用意されています。

例:

Y=ABS(X)	絶対値
Y=SIN(X)	正弦
Y=COS(X)	余弦
Y=TAN(X)	正接
Y=ASIN(X)	逆正弦
Y=ACOS(X)	逆余弦
Y=ATAN(X)	逆正接
Y=SQRT(X)	平方根
Y=EXP(X)	指数関数
Y=LOG(X)	自然対数

- プログラム中で何度も同じ形の計算をする部分等は、副プログラムとしてまとめておくと便利です。
  副プログラムには、関数として値を返す関数副プログラムや、サブルーチン副プログラムがあります。

- RETURN 文は副プログラム中にのみ現れます。

  書き方:

  RETURN

  これによって、副プログラムの実行が終わり、副プログラムを引用した次の文から実行が始まります。

- FUNCTION 文は関数副プログラムの最初にくる文です。

  書き方:

  型名 FUNCTION 関数名*バイト数（変数名や配列名をカンマで区切って並べたもの）

  FUNCTION 文に現れた関数名は関数副プログラムのなかで変数として現れます。

  RETURN 文を実行した時のその変数の値が、関数副プログラムの返す値となります。

例:

```
REAL FUNCTION F1*8 (X,Y)
IMPLICIT REAL*8 (A-H,O-Z)
Z=X*(1.0D0-X)*Y
RETURN
END
```

関数副プログラムは

関数名（変数名や配列名をカンマで区切って並べたもの）

という形で呼びます。() 内は FUNCTION 文に出てきたものと、同じ型、同じ長さ、個数、順序でなければなりません。

例:

```
Y=F1(X, 1.0D0)
```

- SUBROUTINE 文はサブルーチン副プログラムの最初にくる文です。

  書き方:
  SUBROUTINE サブルーチン名（変数名や配列名をカンマで区切って並べたもの）

  サブルーチン副プログラム中で () 内に現れた変数に値を与えることにより、その値をサブルーチンを呼んだ側で使うことができます。

  例:
  ```
  SUBROUTINE S1 (X,Y,Z)
  IMPLICIT REAL*8 (A-H,O-Z)
  Z=X*(1.0D0-X)*Y
  RETURN
  END
  ```

  サブルーチン副プログラムは CALL 文を使って、
  CALL サブルーチン名（変数名や配列名をカンマで区切って並べたもの）
  という形で呼び出します。
  () 内は SUBROUTINE 文に出てきたものと同じ型、長さ、個数、順序でなければなりません。

  例:
  ```
  CALL S1 (X,1.0D0,1.0D0)
  ```

- プログラム文は主プログラムの最初にくる文です。

  書き方:
  PROGRAM プログラム名

  例:
  ```
  PROGRAM MAIN
  ```

C　FORTRAN90 機能 (抜粋) 紹介 (山内 淳)

FORTRAN90(F90) 規格には、free format(自由形式) によるプログラム表現の自由度の拡大、オブジェクト指向を意識した module、変数 scope の導入等のプログラム拡張がなされています。しかしながら、ここではそれらには触れずに F90 文法の中から、FORTRAN77(F77) と整合性が良く便利なものについてのみ簡単に紹介します。詳しくは F90 の文法書や教科書を参照して下さい。

C.1　自由形式と固定形式

プログラムソースの書き方あるいは形式としては、FORTRAN77 以前からある固定形式 (fixed source form) と FORTRAN90 以降で使用可能になった自由形式 (free source form) があります。

固定形式は、プログラムは 7 桁目から 72 桁目に書かなければならない、継続行は 6 桁目に空白または 0 以外の文字が有るときには継続行となる、等、プログラムの行における桁の役割が厳密に決まっていました。

自由形式では、これらの制限が無くなり、より自由にプログラムを書くことができます。以下に固定形式からの変更点を列挙します。

- 1 行の内、プログラムを書く桁は 132 桁目までならどこに書いても構いません。実質的に自由にプログラムのレイアウトができます。
- コメントは、“!” より右側がコメントとして扱われます。つまり、“!” で始まる行は行全体がコメントとして扱われ、プログラム中のある行の途中に “!” があるときには、“!” より右側がコメントとして扱われます。
- 文を継続するときには、行の終わりに”&” を書くと、その次の行に継続することができます。但し、“!” より右側に書くとコメントとして無視されます。

さて、プログラムを実行するためにはコンパイラによって、バイナリファイル (実行ファイル) にコンパイルしなくてはなりません。コンパイラは与えられたソースファイルが、自由形式もしくは固定形式のどちらであるか区別できませんので、以下のように人間が教えて上げる必要があります。

- コンパイラオプションで教える。これはコンパイラによってオプションが異なることがありますが、例えば、-fixed, -free 等で、それぞれ固定、自由形式を指定することができます。
- 暗黙のファイル suffix で教える。これも厳密にはコンパイラ依存ですが、概ね次のような形式のものが多いようです。ファイル名が.F, .f で終わっていれば固定形式、.f90, .F90, (.f95, .F95) 等で終わっていれば、自由形式。この方法が便利なので、通常はこちらを使います。

- 自由形式のプログラムと固定形式のプログラムを混ぜてコンパイルすることも出来ますが、一つのファイルに両方の形式で混ぜてはいけません。必ず、一つのファイルはどちらかの形式に統一して、自由形式と固定形式のプログラムソースは別々のファイルとしてコンパイルするようにします。

C.2 動的メモリー割り当て

F77 の規格では、subroutine 内での配列変数は subroutine に入った段階で配列の次元、サイズが確定している必要があります。更に、配列サイズが変数で与えられている場合には、配列そのものも仮引数として上位の subroutine もしくは main program から引き渡す必要があります。

例えば、

```
subroutine ex02(a,n)
integer n
real*8 a(n)
```

は許されますが、

```
subroutine ex03(n)
integer n
real*8 a(n)
```

```
subroutine ex04()
integer n
real*8 a(n)
read(5,*)n
```

等は許されません。

これは program 実行開始時に必要なメモリーを全て割り当て、プログラム実行中は新たなメモリー割り当て、解放を許さない F77 の規格により定められています。このような記憶領域 (メモリー) の割り当て方を、「静的メモリー割り当て (static memory allocation)」と呼びます。

一方で、ある subroutine で作業配列が必要になり、そのサイズが入力により決定するというケースには頻繁に出会います。その際に便利なのが、必要な時に必要なだけのメモリーをシステム (OS) が割り当ててくれ、必要がなくなればメモリーを解放してくれる「動的メモリー割り当て (dynamical memory allocation)」です。

このように動的メモリー割り当ては便利ではありますが、プログラムが実行開始しても、実行中の他のプログラムのメモリー使用状況や、途中でメモリー割り当てがシステム上限を越えてしまう等によって、実行途中でプログラムが落ちてしまう可能性があります。一方で、静的メモリー割り当ての場合には、予めプログラムが使うメモリー領域を確保してから実行開始するので、メモリー割り当てが原因で途中でプログラムが落ちる心配がありません。このような性質を踏まえた上で両者を使い分けてください。

F90 規格では動的メモリー割り当てを利用できる次のような配列を用意しています。以下では、subroutine や function などを「手続き」と総称しています。

- automatic array

  手続きの最初に宣言され、サイズは仮引数として与えられている配列。
  つまり、

```
subroutine ex03(n)
integer n
real*8 a(n)
```

  を許しています。

- allocatable array

  これはプログラマーが配列の確保、解放を自由に設定できるので大変便利な配列です。使い方は、宣言、領域確保、領域解放の 3 段階からなります。

  宣言 手続きの最初に指定したい変数が allocatable array であると宣言する。例えば、

```
integer,allocatable:: i(:)
real(kind=8),allocatalbe:: a(:,:)
```

  この宣言書式は F90 独自のもので規定については後述します。allocatable と書く部分と変数のサイズが定数や変数ではなくて':' になっていることが特徴です。

  領域確保 次に実際に配列を使う前に領域を割り当てます。

```
allocate(i(n),a(n1,n2),stat=ier)
```

  'allocate' が配列に実際のメモリー領域を割り当てる命令で、引数には配列サイズを明記した変数を渡します。ここで ier は整数変数で allocate 命令実行後に、終了結果 (0 が正常終了) が入ります。

  'stat=' は省略できますが、メモリー確保に失敗した場合には通常メッセージはほとんどなく異常終了します。明示的に'stat=' の変数をチェックすることで、どの部分のメモリー確保が原因で落ちるかを知ることができるので、省略せずにチェックすることをすすめます。

  領域解放 確保したメモリーを使用し終わって、メモリーを解放します。

```
deallocate(i,a)
```

この場合には通常エラーは起きませんが、解放し忘れると使用しないメモリー領域がシステム中に溜って行き、使えるメモリーが減ってしまいますので、不要になったメモリーは必ず明示的に解放しましょう。このように使えないメモリー領域が増えていくことをメモリーリークと呼ぶことがあります。

以上、automatic array 並びに allocatalbe array について紹介しましたが、領域確保に失敗したときの処理が可能であることから、大きな配列を扱うときには allocatalbe array の方が適していると言えます。

C.3 配列の vector 記法

F90 では配列を一まとめにして扱う記法を使うことが出来ます。これをうまく利用するとプログラムがコンパクトになり、読みやすい code が書けます。

これについて説明する前に、配列の指定方法を例をあげて説明します。

```
a(n1:n2:n3)
```

は、一次元配列 a() の成分の内、n1 番目から始まり、n3 個置きに n2 番目までの成分からなる配列を意味します。':n3' は省略可能で省略された場合には 1 を指定したことと同じです。n1 を省略すると宣言された最小の添字、n2 を省略すると宣言された最大の添字を指定したことになります。a(:) とすると元の配列全体を指します。二次元以上の配列の場合にも同様です。また、配列の一部分 (section) を指定することもできる。例えば、b(10,10) と宣言した場合に b(2:5,5) は'b(2,5),b(3,5),b(4,5),b(5,5)' を成分とする 1 次元の配列を表す。特に、配列変数の次元を指定せずに記述すると配列全体を指定したことになります。つまり、a(10,10) で、'a' は、'a(:,:)' または'a(1:10,1:10)' と同じ意味になります。

以下にこれを利用した vector 記法について述べます。

- 配列の各成分毎の演算が可能。その際、scalar(配列ではない変数) は、任意の次元、サイズの配列と適合する。

```
integer:: i,j,a(10,10),b(10,10),c(10,10)
....
a=b+c
```

```
do i=1,10
  do j=1,10
    a(j,i)=b(j,i)+c(j,i)
  end do
end do
```

```
a=b*2.d0
```

```
do i=1,10
  do j=1,10
    a(j,i)=b(j,i)*2.d0
  end do
end do
```

```
a=b*c
```

```
do i=1,10
  do j=1,10
    a(j,i)=b(j,i)*c(j,i)
  end do
end do
```

左右の code は等価である。2 番目は行列の積ではなく、各成分毎の積であることに注意。

配列の演算では、演算の前後で配列の次元、サイズが一致している必要がある。例えば、a(1:10:2,10)=b(1,1:5) は許される。

- 算術関係の組込み関数に対しても成分毎の演算が許される。

```
integer:: i,j,a(10,10),b(10,10)
....
a=cos(b)
```

```
do i=1,10
  do j=1,10
    a(j,i)=cos(b(j,i))
  end do
end do
```

このような配列をそのまま扱う関数はユーザーが定義することも可能であるが、そのためには module と interface の知識が必要である。

- 配列専門に扱う関数が多数用意されている。例：

DOT_PRODUCT 内積を出力。

MATMUL (成分毎の積ではなく) 行列の積を出力。

TRANSPOSE 転置行列を出力。

MAXVAL,MINVAL 配列成分の中の最大、最小値を出力。

C.4 変数宣言について

- 変数の宣言

  変数の型 (精度), 属性:: 変数のリスト

  と言う形で宣言する。

  変数の型は integer,real 等が指定でき、精度は kind 指定子により'kind=xx' の形で指定される。xx と実際の精度に関しては厳密には処理系依存だが、変数に割り当てられる byte 数になっていることが多い。例えば、real(kind=8) は倍精度実数 (1 変数は 8 byte) を意味する。

  属性 (attribute) には色々な特性を割り当てることができる。例えば、parameter(パラメタ)、allocatable(allocatable array)、intent 等である。この中で、intent は変数の subroutine からの入出力条件を指定するもので、intent(in) とすると subroutine 外から与えられた変数で、subroutine 内での値の書き換えが禁じられる。逆に intent(out) とすると、subroutine 外では値が設定されていないので、その subroutine で値を設定せずに引用しようとすると compile 時にエラーになる。デフォルトは intent(in,out) で、特に何も影響しない。バグの少ないプログラムを書く上で利用して欲しい。

- 'implicit none' の奨め

  F77 を使ったプログラムでは subroutine,function,program の先頭付近に

  implicit real*8 (a-h,o-z)

  という statement が見られることが多い。これは暗黙の型宣言を倍精度に拡張したもので、a-h,o-z から始まる変数は全て倍精度実数として、それ以外は単精度整数として扱うことを意味する。

  この利点 (?) は新しい変数を使用する度にいちいち変数を宣言しなくても良い点にあるが、まさにその点が落し穴になっている。例えば、a1(イチ) という変数を書くつもりで、al(エル) と書いたとしても特に問題なくプログラムはコンパイルされてしまい、画面上や、プリントアウトした紙を眺めてもミスプリには非常に気づきにくい。

  上の暗黙の型宣言の代わりに'implicit none' を指定すると、一切の暗黙の型宣言は無効化され、全ての変数に対して型宣言をしなければならない。一見大変面倒であるが、前述のような typo によるバグの少ないプログラムを作成できる利点がある。

  F77 の段階では、比較的多くの処理系に採用されていたものの、規約ではなく一種の方言であったので、積極的に奨め辛かったが、F90 からは正式に規約に採用されたので、是非使用して欲しい。

C.5 その他

- 'end do' の使用。

  F77 の規格では do 文は

  ```
  do 文番号 i=1,n
  ....
  文番号 continue もしくは実行文
  ```

  のように書かれていましたが、F90 では

  ```
  do i=1,n
  ....
  end do
  ```

  と'end do' 文を使用することが推奨されています。文番号が減ってプログラムがすっきりします。例外として、indent を使用しても do-loop の終わりがわかりにくいような極端に長い do-loop の場合には、F77 のように文番号をつける場合もあります。

- 多次元配列の実装。

  F90 に限らず、FORTRAN では多次元配列の左の添字が memory の連続した領域へ割り当てられます。連続した領域で割り当てられた変数は、vector 機 (スーパーコンピュータ) でのアクセス速度も向上し、PC や EWS 等の cache にもヒットしやすくなり実効速度が向上します。従って、大規模なプログラムを書く場合には、最長の loop を一番内側のループとして、変数の一番左の次元に来るように coding するのがコツです。

  因みに、C 言語では FORTRAN とは逆に右の添字が memory の連続した領域に割り当てられます。

- Compiler のデバッグオプションの利用。

  配列を使用しているときに多いバグの一つは、配列の添字が宣言の上下限を越えて配列を参照、もしくは書き換える場合である。参照している場合には、明らかに意図した結果にならないし、書き換えている場合には高い確率で'segmentation fault' 等の message を残してプログラムが死んでしまう。このような場合に、プログラム実行時に配列の添字が、宣言された範囲を越えていないか、チェックするオプションが多くのコンパイラに用意されている。その他にも subroutine の実引数と仮引数の整合性をチェックするオプション

等デバッグに役立つオプションが用意されている。詳しくは man コマンドで調べてみて欲しい。

但し、これらのオプションは本来は必要ない演算を行うためにプログラムの実行時間は極端に遅くなることに注意が必要である。

D　デバッグ入門 (山内 淳)

プログラムを作成するにあたって、書いたコード (code) が、そのまま正しく走ることは非常に希です。通常は、何回もテストを行い、間違いを修正して、正しく動作することを確認して初めてプログラムが完成したと言えます。また、この場合にも、多くの場合、全ての可能なテストを網羅していると言うわけにはいかないので、100%正しいと言う保証は無いことを心に留めておくべきです。さて、ソフトウエアの分野では、このような間違いのことをバグ (bug:虫) と呼び、この間違いを取り除くことをデバッグ (debug) と呼んでいます。

デバッグは、バグがあるかどうかを調べるコードのテスト、バグが存在する場合にプログラムの論理構造を理解した上でバグと判断したテスト結果の原因を遡って調べていく作業、などからなり、いずれも非常に高度な論理的思考を要する作業になっています。理科系の学生の良い能力として論理的思考ができることがよく取り上げられていますが、まさにデバッグ作業は論理的思考を鍛える非常に良い課題になっているので、積極的に取り組んで下さい。

ここでは、基本的なデバッグの方法について簡単に解説します。

D.1　バグの分類

プログラムを作成する段階別に見つかるバグについて述べます。通常は、プログラムを書いた後、コンパイルして、実行し、一応動くようなら、テストを行い、問題ないと分かった時点で完成します。その各段階で、期待と異なる結果（これをエラーと呼ぶことにします）が出てくるので、これを修正 (デバッグ) するわけです。このエラーを段階毎に次のように分類してみます。

コンパイル時エラー プログラムをコンパイルした時に、コンパイルができず、エラーメッセージを出して止まる。これはプログラムに文法ミスがあることが原因です。通常、このタイプは、非常に初歩的なのでバグとは言わないことも多いです。

実行時エラー 無事コンパイルができた後に、実行させてみると、エラーメッセージを出力して、途中で停止する。

論理エラー プログラムは実行は最後まで終了するが、出力に明確におかしな点があったり、期待するテスト結果と異なる結果が得られる。つまり、プログラムを構成する論理が間違っています。これが一番やっかいな、つまり、取りにくいバグです。

上記のそれぞれの段階のエラーの基本的なデバッグ方法について以下に述べます。

D.2 コンパイル時エラー (compile error)

プログラムをコンパイルした時に、コンパイルができず、エラーメッセージを出して止まる。これはプログラムに文法ミスがあることが原因です。この解決には、コンパイラの出力するエラーメッセージをよく読むにつきます。例えば、

```
program test
  implicit none
  real(kind=8):: a,b,c
  do i=1,10
    a=b+c
  end do
end program test
```

というコードをコンパイルすると、

```
$ gfortran test.f90
test.f90:4.6:                                          ←ファイル名、エラーの
場所

  do i=1,10                                            ←エラーのある行
     1                                                 ←エラー箇所の番号
Error: Symbol 'i' at (1) has no IMPLICIT type          ←エラーメッセージ
```

というメッセージを出してコンパイルが停止します。

最初の行の “test.f90:4.6:” は、ファイルが”test.f90” で、エラーの場所が 4 行目の 6 桁目にあることを示しています。次の” do i=1,10” は、エラーがある行のコードです。その次の行の”1” はエラーの番号です。ここでは 1 番目のエラーということで 1 となっています。次のエラーメッセージ”Error: Symbol ’i’ at (1) has no IMPLICIT type” は、直訳すると「エラー：(1) にある’i’ というシンボルは、暗黙の型を持っていません」ということです。これは変数 i の明示的な宣言を行っていないので、通常は暗黙の型宣言として整数が割り当てられますが、`implicit none` の指示があるので、暗黙の型宣言の割り当てが禁止されているため、“no IMPLICIT type”(暗黙の型を持っていない) とエラーになっているわけです。

ここで、シンボル (symbol) というのは分かりにくい言葉で、FORTRAN では変数、関数、サブルーチンなど名前の付けることができるものを一般的に指します。なお、シンボルの意味は言語によって違うことがあるので注意して下さい。コンパイラには i が変数とは分からないので、シンボルと言っているわけです。

一般に、プログラム用語の英語、時には日本語のメッセージも直訳で何を言っているのか分からない場合があるので、いくつかのキーワードと、そのキーワードが関連するバグについてこの節の終わりに、簡単にコメントしますので、適宜

参照してください。これらの語は次の実行時エラーのエラーメッセージの解読にも有効です。

D.3 実行時エラー (run-time error)

プログラムのコンパイルが成功し、a.out などの実行ファイルが作られるが、この実行ファイルを実行するとエラーメッセージを出力してプログラムが停止する。この場合の出力 (標準エラー出力) に出される内容が実行時エラーです。その他に、出力に”NaN”、到底ありえない、大きな数、でたらめな数、などが出力されるのも、実行時エラーと言えなくもないが、一応最後までプログラムは実行されるので、対処法の分類上、これらは後述の論理エラーに含めます。この場合も、エラーメッセージを注意深く読んで意味を意味を理解することが、デバッグへの近道です。

実行時エラーの主なものは、入出力関連と「Segmentation fault(セグメンテーション　フォールト)」の２つです。

「入出力関連エラー」

入出力関連エラーの原因としては、例えば、読み込むファイルのデータが足りない、書き出す変数の型と書式が合わないなど、非常に多くのバリエーションがあります。この様な場合には大抵、”I/O”, “read”, “write”, “format” と言った入出力関連の用語がメッセージとして出されるので、丹念にメッセージを読むとエラーの場所の検討がつきます。

プログラムに問題が無くても、入出力関連エラーが起こる場合があります。これもいくつも原因が考えられます。例えば、入出力するファイルの属性 (rwx など) が不適切である場合です。経験上、意外と多いのが、quota による制限に引っかかる場合です。これは一人当たりの所持できるファイルの総容量に上限があるためです。この情報については、“quota” コマンドで調べてみて下さい。特にプログラムに問題がなさそうなのに、ファイルを書き出すときにエラーになる場合には疑ってみて下さい。

「Segmentation fault(セグメンテーション フォールト)」

ここでの segment(セグメント) というのは、物理的なメモリー上での変数の区切りのことです。プログラム上で宣言された変数、配列変数は、プログラム実行時に物理的なメモリ上に配置されます。この内、配列の場合は、配列の先頭から連続的に物理メモリ上に割り当てられます。「セグメンテーション フォールト」は割り当てられた変数の区切りが、プログラム上で期待されているものと異なる場合にでるエラーです。

例えば、「real(kind=8):: a(10)」と宣言すると、a(1) の先頭アドレスから、8byte 毎に 10 個の変数が物理メモリ上に連続して配置されます。ここでアドレスというのはメモリ空間上の位置 (住所) のことです。今、「integer:: ib(10)」で宣言された変数 ib が、a(10) の次のアドレスから配置されたとします。a(10) の隣は ib(1) なわけですが、何かの間違いで、ib(-1) に整数を代入したとします。とすると、integer は

4byte なので、a(10) のメモリ領域の後半に実数ではなく、整数が書き込まれます。その後、a(10) を参照すると、8byte の前半は実数の半分で、後半は integer なので、おかしなことになります。この様な不整合を検知して、変数の区切りが間違っているという「Segmentation fault(セグメンテーション フォールト)」(SF) メッセージが出力されるわけです。

さて、上記の ib(10) は整数だったから、システムの監視によりエラーが検出されたわけです。これが倍精度実数「real(kind=8):: b(10)」だったら、どうなるでしょうか？a(10) と同じ倍精度実数が書き込まれたので、システムはこの誤りを感知しません。つまり、プログラムは間違った計算を続けて行って、最終的に間違った結果を出して「正常」終了します。これは、SF メッセージを表示して終了するよりも、間違っていることに気づかないために遥にたちの悪いケースになります。またシステムによって物理メモリ上への変数の配置が異なるため、コンピューターによってエラーがでたりでなかったりします。

上記のように、SF は宣言されている index の上下限を越えて配列を扱う場合に起こりますが、subroutine, function の呼び出し時の仮引数と、実引数の不整合によっても起こります。

「Segmentation fault(セグメンテーション フォールト)」の対処法

このように、Segmentation fault 関連のバグは、基本的には、実行時に配列の引数 (index) が配列宣言時の上下限を越えていないか、subroutine, function の呼び出し時に、仮引数と実引数が整合しているかをチェックすれば発見できます。幸いなことに、ほとんどの処理系 (コンパイラ) には、このようなこと (実行時チェック) をするオプションが提供されています。この授業で使う gfortran では、配列の index に関するチェックを行うオプション”-fbounds-check” が提供されています。コンパイル時に指定することによって、実行時に配列のチェックを行います。

```
$ gfortran -fbounds-check (ソースコード)
```

注意点としては、実行時に配列引数のチェックを行うわけですから、その分実行がかなり遅くなります。テストして問題がないことを確認したら、本番の実行時にはこのオプションを外して下さい。

前述のように、実行時エラーが出力されずに最後までプログラムが実行している場合でも、配列が壊れていることがありますので、上記のオプションを使ってテストすることを進めます。

D.4 論理エラー (logic error)

プログラムは最後まで実行されて正常終了するが、テストデータが期待通りの結果を出さない、出力に”NaN”、到底ありえない、大きな数、でたらめな数、などが出力される、など想定と異なる結果が得られる。このような場合は、プログ

ラム中にバグが存在し、論理的に間違った code となっているので、論理エラーと呼ばれます。

論理エラーのデバッグにはいる前に、前節のデバッグオプションを付けてコンパイルした実行ファイルでのテストをしておくようにして下さい。前節で述べたように、範囲の上下限を越えた配列 index が、プログラムの途中停止を引き起こさずに、誤った結果を出力している場合が多々あります。

論理エラーに対するデバッグの基本的な方法は、「バックトレース (back trace)」と「分割統治法 (divide and conquer)」の二つです。

「バックトレース」は、誤った結果を検出した出力部分から、原因を上流に遡って追跡 (back trace) してゆく方法です。例えば、”NaN” という出力をデバッグする場合には、最初に、”NaN” を出力している write 文とその書式 (format)、変数について調べます。これらが正常な場合には、プログラムの上流に”NaN” となっている変数に式を代入している部分があるはずですから、この代入式に使用されているる変数を write 文などで出力させて調べます。代入式が複数の変数からなっている場合には、それぞれの変数を書き出して調べてゆきます。調べた変数の中で”NaN” の原因となっていると考えられる変数について、更に遡って追跡していきます。これを繰り返すことによってデバッグを行います。

バックトレースを効率的に行うツールとしては、デバッガ (debugger) と呼ばれるものがあります。システムによっては GUI 環境の非常に使いやすいデバッガが提供されていることもあります。この授業で使うシステムでは、gdb というデバッガが使えますが、テキストベースであまり使いやすくはありません。興味がある人は調べてみて下さい。

「分割統治法」の考え方は、大きな (文字数の多い) プログラムのデバッグは難しいので、全体のプログラムを小さい部分に分けて、分割したそれぞれの部分に対してデバッグを行い、最終的に全体を統合してデバッグしたプログラムを作成することです。分割した各部分毎にテストデータを作成してテストする必要があります。論理エラーのデバッグ時に適用するには、バグを示すなるべく小さい部分に限定してデバッグを行うと効率的です。

D.5 用語集

allocate, allocation メモリなどを「割り当てる」、「割り当て」。

array 「配列」。

declaration 「宣言」。この語が出てきたら `integer`, `real` などの変数の型宣言に関連した文法ミスを最初に疑うこと。

divide by zero 「ゼロ割」。ゼロで割ること。処理系によるが、大抵メッセージを出して停止する。

dimension　配列の「次元」。

end-of-file/ENDFILE　「ファイルの終わり」。例えば、プログラムで必要とする入力数値が全部入力される前に、入力ファイル中の数値 (レコード) がなくなってしまった場合には、「ファイルの終わりが検出されました」のようなメッセージが出力される。

error　「エラー」、「誤り」。

exception　「例外」。計算中に通常想定する以外の結果を検出した。大体停止する。

explicit　形容詞「明示的な」。implicit と対比。

floating point　「浮動小数点」。

format　「書式」、「フォーマット」。入出力時に指定する書式。まずは入出力時の書式をチェックすること。

function　「関数」。

index　「添字」。配列の添字。

integer　「整数」。

I/O　「Input/Output」、「入出力」。open,read/write 文あたりに原因がありそう。

implicit　形容詞「暗黙の」。明示的に指定しない場合に割り当てられる属性。explicit と対比。FORTRAN では、「暗黙の」型宣言ととして使われることが多い。

inconsistent　形容詞。不整合で、矛盾があること。

memory　「メモリ」。

NaN　Not a number. 数字があるべきところに、数字以外のものがあること。

real　「実数」。

record　「記録」、「レコード」。ファイル中の、文字、数字などのデータのこと。

segmentation, segmented　データの「区切り」のこと。例えば、4byte 整数では、4byte が区切り。単 (倍) 精度実数では、4(8)byte が区切り。

signal　「信号」。通常、取り決められた状況が発生した時に「信号」が送られ、割り込み処理などを行う。

SIGSEGV エラーメッセージの略語。多分、SIGnal of SEGmenation Violation の略. segmentation fault, stack overflow など、メモリ関係のエラーを意味する。

stack 「スタック」。メモリの一種。足りなくなると、“stack overflow” のようなメッセージが出る。この場合、ulimit コマンドなどを調べてみると良い。

statement プログラムを構成する「文」のこと。通常プログラム 1 行分、継続行がある場合はそれも含める。

subroutine 「サブルーチン」。

subscript 配列の「添字」。`a(i,j)` の `i,j` のこと。“Subscript out of range” は、配列の添字が配列で宣言されている添字の上限、もしくは下限を越えていることを意味する。

symbol シンボル (symbol) というのは分かりにくい言葉で、FORTRAN では変数、関数、サブルーチンなど名前の付けることができるものを一般的に指します。言語によって意味が違う場合があるので注意して下さい。

syntax error 「文法エラー」。この語がでてきたら、FORTRAN 文法に従わない記述があるということ。

unit 「ユニット」、「装置」。FORTRAN では入出力時の (論理) 装置に番号を指定します。その装置番号をさすこともある。例えば、unit 5, 6 はそれぞれ、標準入力、標準出力に割り当てられ、read(5,*) のように使われます。

variable 「変数」。

violation 「違反」。規則に従わないこと。