1 プログラミング入門

1.1 定数と文字列の出力

コンピュタは単純な命令を多数組み合わせて様々な計算を行なうことができるが、計算して求まった結果を出力しなければ意味がない。この演習で学ぶFORTRAN語では、出力には“print”と言う命令を使う。[1] 例えばモニタの上に“Hello”と言う言葉を表示させたければ、次のようなプログラム

```
␣␣␣␣␣␣print␣*,␣'Hello'
␣␣␣␣␣␣end
```

を用意して、計算機に実行させれば良い。print 命令の前にある六個の“␣”は空白を六文字書く事を示している。この様に命令文の前に空白を置くのは古いFORTRAN言語の仕様なのだが、この演習では古い規格に従って各行の先頭に六個のスペースを必ず置くことにする。煩わしいので、以後この記号は書かないが、これから配るプリントの中のサンプルプログラムには、この空白が付いていることを頭に入れておいて欲しい。また、プログラムの命令は行の先頭の空白から数え始めて72文字目までに書くという古い約束もある。古い処理系では73文字目以降は無視されてしまうこともあるので、こちらの規則もこの演習では従うことにしよう。“print”の後ろの星印(*)は表示の書式を処理系に任せるという意味であるのだが、この演習ではこう書くものだと決めてしまって構わない。その後にカンマ(,)を置いてから単引用符(')で囲った文字列を書けば、書いた通りに画面に出力される。プログラムの最後には処理が終ったことをコンピュータに伝えるために必ず end 文を置く。

画面への表示は print 文一つにつき一行が出力される。同じ行に複数の文字列や数字を書きたければ、「print *, 'Hello. ', 'Glad to see you.'」のようにカンマで区切って並べればよい。一つの命令文が長くなり過ぎて72文字以内に収まらなくなったときは、次の行に続けて書くこともできる。その場合は次の行が前の行の続き(継続行)であることを示すために、行の先頭の空白文字を六文字書くのではなく、空白五文字の後、六文字目に“$”を書く。例を示すと

```
      print *, 'Hello.  Happy to meet you.  My name is Lucy.  ',
     $         'I lived in Africa about 3,000,000 years ago.'
      end
```

こんなプログラムが書ける。このプログラムを実行すると、我々の遠い祖先からの挨拶が表示されるであろう。この方法による文の継続は、print 文に限らず他の文でも使うことができる。

> 基本問題 1: 画面一行目に「Hello, 」、次の行に「I am a computer.」と表示するプログラムを作れ。作成したプログラムを提出せよ。(以下「提出せよ」の文は繰返さないが同様)

また print 文の後ろには文字列だけでなく、数字や式を書いても良い。式は計算されて答えが表示される。例えば、

```
      print *, 1 + 1
      end
```

というプログラムを書くと、画面に表示されるのは「1 + 1」という文字列ではなく、1 + 1 を計算した答えの2である。なお、式の中で使う四則演算の演算子は、“+”と“-”は普通の数式と同じだが、掛け算には × ではなく“*”、割り算には ÷ ではなく“/”を用いる。また掛け算の記号を続けて書くとべき乗の意味になる。例えば“3**4”は 3^4 の意味である。(“3**4**2”は“3**(4**2)”と解釈される。)

[1] 以下プリントの中では、プログラム中に書く命令やコンピュータに対するコマンドをタイプライター体の活字で示す。

さて、print 文の後ろには数字や式を書くことができると言ったが、実際にプログラムを書いて計算結果をよく見ると、文字が出力される位置が異なっていることに気付く。例えば、「print *, '1'」と「print *, 1」を比べると、前者は出力行の左端から一文字空けた位置に数字の "1" が印字されるのに対して、後者では沢山の空白が出力されてから数字の "1" の文字が印字される。そのため「print *, '1 + 1 = ', 1 + 1」の様に計算式と計算結果を並べて表示させるプログラムを書こうとすると、少々体裁の悪い出力が得られてしまう。FORTRAN 語には体裁を整える方法も用意されているのだが、この演習は覚えることを最小に止めるという方針なので、体裁の悪いことは気にしないことにしよう。

> 基本問題 2: 単引用符付きの "'1'"、引用符なしの "1"、引用符なしの "1.0" を一行ずつ出力するプログラムを作り、印字のされ方の違いを確認せよ。

引用符の有無で数字の "1" は印字のされ方が異なっていたが、上の問題の様に "1.0" を印字した場合は、指示した覚えのない余分な "0" がズラズラと出力されるであろう。今回の演習で用いる FORTRAN 言語は、小数点以下の部分を持たない整数とそれらを持つ実数を区別する計算機言語である。計算を効率的に行なうために整数と実数は異なる形式で記憶され、異なる方法で計算が実行される。文字としての'1' と数字の 1 を区別するだけでなく、整数と実数も区別されるのである。出力される時もこの二つは別々の方法で扱われる。そのため "1.0" の出力には余分な "0" が多数出力されたのである。この演習では文字列は余り取り扱わない方針なので、文字の'1' は深刻に考えなくて良いのだが、整数と実数が区別されることは覚えて置かなくてはいけない。例えば、整数同士の演算は答えの小数点以下を切り捨てて整数にして返すという約束が FORTRAN 語にはある。そのため不用意に割り算をすると、「1 / 3 → 0」となってしまう。「print *, 1 / 3 * 3」とプログラムに書くと、予想に反して画面には 0 が表示されることになるだろう。FORTRAN 語でプログラムを書くときには、物の数や回数を数えるための数—整数—と、量を測るための数—実数—を区別して考える必要がある。整数の 1 ではなくて、実数の 1 であることを計算機に伝えたければ、"1.0d+0" という具合に表現する。小数点を付けて小数点以下の部分を持つ数であることを示した上で、位取りを示す整数を付け加える。つまり "d" の後のプラス・マイナスの符号の付いた数は、実数を "(真数)×10^n" と表記したときの n である。[2] 実数同士の足し算、例えば "1.0d+0 + 2.0d+0" なんて表現は＋記号が沢山あって気持が悪いけれど、唯の 1＋2 を実数として計算することになる。この演習では実数としてはこの形式のものだけ覚えていれば良い。

1.2 変数の利用

print 文さえ知っていれば、円周率を 10 億桁表示するプログラムも原理的には書ける訳だが、[3] ソースコードに 10 億個の数字を並べてそれを表示するだけというのでは、計算機を使っていると言えるか微妙なところである。計算機を計算機らしく使うためには変数を使えるようにならなければいけない。計算機で用いる変数と言うのは、数値を記憶して置くメモ帳あるいは箱のようなものである。例えば整数の変数を一つ用意して、値を代入して、代入した中身を表示するサンプルプログラムを示すと

```
implicit none
integer :: i
i = 0
print *, i
end
```

一行目の「implicit none」は暗黙の型宣言を無効にする宣言だが、この演習ではプログラムの 1 行目

[2] "(真数)×10^n" という形式の通りに表現しようと考えて、"1.5*10**(-3)" などと書くと、整数同士の演算の答は整数という約束のため、「10**(-3) → 0」となるので注意が必要。"1.5d+0*10.0d+0**(-3)" なら多分大丈夫だけれど、"1.5d-3" と書く方がスマートだと思う。

[3] 「print *, '3.1415 92653 58979 32384 62643 ...'」。以下 10 億桁目まで print 文を並べる。

には必ずこう書くものだとして置こう。二行目の「integer :: i」という宣言で“i”が整数の変数であることを計算機に伝える。変数iが実数であるならば、“integer”の代りに“double precision”という宣言を用いる。また、変数名“i”のところにはアルファベット6文字までの任意の名前を書いて良いし、[4] カンマで区切って幾つもの変数名を並べても良い。以上の二行は計算機に具体的な計算を実行させる命令ではなく、変数の種類を指示するための文である。三行目の「i = 0」は、こうして作られた変数iに整数0を代入する命令であり、四行目で変数の中身を表示する。画面には変数iの内容である0が表示されるはずである。「i = 0」や「print *, i」のように代入や計算を具体的に指示する命令文を実行文、「integer :: i」のように変数の種類を指示するような文を宣言文と呼ぶ。FORTRAN言語には宣言文は全ての実行文よりも前に置かなければならないという規則がある。

ここで注意しなければいけないのは、「i = 0」という命令が普通の数式の意味するところとは微妙に異なることである。プログラム中に「i = 0」と書けば、この命令は右辺の数値0を左辺の変数iに代入しろと計算機に命令していることになる。だから「i = i + 1」という命令(変数iに記憶されている数値に1を加えて、その結果を再び同じ変数iに記憶する)は正しい命令であるが、「0 = i」という命令は有り得ない。また、引用符で囲われていない文字は変数名なので「print *, i」と書くと画面に表示されるのは“i”という文字ではなく、その時に変数iに記憶されている数値が出力される。

さて、変数を使うことができるようになると、計算に使うデータをキーボードから打ち込んでプログラムに教えることができるようになる。こうしたデータの読み込みにはread文を用いる。「read *, i」で変数iに代入すべき数値をキーボードから入力できる。例を示すと、

```
implicit none
integer :: i
print *, 'input an integer: '
read *, i
i = i + 1
print *, i
end
```

このプログラムを実行すると、計算機は「input an integer: 」というメッセージを表示して、そこで実行が一旦中断、ユーザが数値(整数)を入力するのを待つ。適当な数値を入力すると、その数値に1を加えた数が表示されるであろう。[5]

> 基本問題 3: 画面に「How old are you?」と表示した後、整数の入力を受けて、さらに読み込んだ数値を「You are ... years old.」と出力するプログラムを作れ。

1.3 条件による分岐

データの数値がある値を越えない時だけ何かの処理を施すといった事が必要になることは多い。こうした時には

```
if (条件) then
    条件が満されたときの操作
end if
```

という構文を使う。条件が満されたときの操作は複数行にわたっても良い。以下にサンプルプログラムを示す。

[4] 実は、変数名の先頭以外の場所には数字を入れても良い。

[5] 「print *, i + 1」と言う文は変数“i”の内容に1を加えて表示する命令になるが、「read *, i + 1」は無効な命令である。入力した数から1を引いたりはしてくれない。計算機は気が効かないのである。

```
      implicit none
      integer :: i
      read *, i
c
      if (i > 0) then
        print *, i
      end if
c
      if (i < 0 .or. i == 0) then
        print *, 'input value is not positive.'
        print *, 'I do not show you the number.'
      end if
c
      end
```

このプログラムは変数 i に整数の値を読み込んで、その値が正 (i > 0) ならばその値を画面に表示する。条件の指定には通常の不等号を用いた数式を用いて良い。等しくない (≠) は"/=" とタイピングする。ただし等しいかどうかを判定するときには、等号 "=" を一個だけ書くと前述の代入命令になってしまうので、"==" という風に二つの等号を続けて書く。また、".and." や ".or." で条件を組み合わせることもできる。"and" や "or" の前後にピリオド (.) を置くことに注意。

なお、このサンプルプログラムには、プログラムを読むときに判りやすいようにする工夫が幾つか施してある。条件が満された時の操作を記述する文、「print *, i」が他の命令文よりも少し右にずれて書いてあるのは「字下げ」と呼ばれる習慣で、if 文の始まりと終わりを判りやすくするものである。行頭の 6 文字分の空白文字を書きさえすれば、どこから文を書き始めようが計算機は全く気にしない。しかし人間にとっては、字下げにより if 文に対応する end if 文を探すのが格段に容易になる。また、"c" で始まる行が二つある。これはコメント行と呼ばれるもので、計算機は "c" で始まる行に何が書かれていても無視することになっている。プログラムが複雑になって来ると、変数の意味や用いたアルゴリズムを書き込んだり、プログラムの区切りを示したりしたくなる。こんな時は "c" で始まる行を用いる。[6] 字下げもコメント行も、どちらも計算機がプログラムを解釈するときには何の効果もないため、プログラミングの習得に必須ではないのだが、読み易いプログラムを書くために取り入れると良い。

この演習では覚えることを最小限に留めるように努力しているので、遠回りをしなければならないことがある。上のプログラム例で、変数 i が正でない時の分岐がその一つである。直前で変数 i の中身が正かどうかを判定しているので、i が負または 0 かどうかの判定は冗長である。さすがに煩わしいので「最小限の知識」の例外を一つ導入しよう。こんな時には "else" という命令を使うことができる。

```
      implicit none
      integer :: i
c
      read *, i
      if (i > 0) then
        print *, i
      else
        print *, 'input value is not positive.'
      end if
c
      end
```

命令 "else" は文字通り「そうでなければ」と言う意味である。「if (条件) then」の条件が満されなければ (つまり入力した値が正の数でなければ)、"else" から "end if" までの命令が実行される。else 文の仲間の命令として else if (条件) then という命令もあり、

[6] コメント行に文字 "c" を使うのは古い FORTRAN 言語の仕様。新しい Fortran90 以降では "!" を用いる。

```
if (i > 10) then
  ...  (i が 10 より大きい時の処理)
else if (i > 0) then
  ...  (i が 1 から 10 までの時の処理)
else
  ...  (i が 0 又は負の数の時の処理)
end if
```

のように使う。この様な場合、条件 “`i > 0`” と “`i > 10`” の順序は大切である。

> 基本問題 4: 整数の入力を受け、読み込んだ数値が正の数なら「... is positive」、負の数なら「... is negative」、と表示するプログラムを作れ。(... のところには入力した数値が入る。入力した数が 0 なら何も出力しない。)

1.4 関数の呼出しI (組み込み関数の利用)

FORTRAN 言語では四則演算の他に、しばしば使われる幾つかの関数が予め定義されており、プログラムの中で用いることができる。こうした組み込み関数には、初等関数の指数関数 (“`exp(x)`”)、e を底とした対数関数 (“`log(x)`”)、三角関数 (“`sin(x)`” など) があり、見ての通りの意味を持つ。逆三角関数 $\arcsin(x)(=\sin^{-1}(x))$ などは “`asin(x)`” などと書く。他に、整数や実数の絶対値を求める “`abs(x)`”、割り算の余りを求める “`mod(x, y)`” などがある。また、実数の小数点以下を切り捨てて整数に変換する “`int(x)`”、逆に整数を (倍精度) 実数に変換する “`dble(x)`” など、計算機特有の関数も存在する。

> 基本問題 5: 実数の平方根を計算する組み込み関数 “`sqrt`” を用いて $\sqrt{2}$ を計算するプログラムを作れ。(注意: 関数 “`sqrt`” は整数の引数を取れない。)

簡単な例として、一次方程式 $ax+b=0$ を解くプログラムを作ってみよう。係数 a と b を与えて、この式を満す未知数 x の値を求めるという問題である。こんなプログラムを書けばよいだろう:

```
      implicit none
      double precision :: x, a, b
c
      print *, 'input a and b:'
      read *, a, b
c
      if (abs(a) < 1.0d-15) then
        print *, 'absolute value of <a> is too small.'
      else
        x = - b / a
        print *, 'x = ', x
      end if
c
      end
```

まず変数の宣言をして、方程式の係数 a, b の値を二つの変数 “`a`”, “`b`” に読み込む。1.1 節で FORTRAN 言語では整数と実数を区別すると説明したが、この課題では個数や順番は登場しないので、変数は全て (倍精度) 実数で良い。一次の項の係数 a が 0 であると一次方程式は解くことができないので、入力された値 ‘`a`’ が 0 と等しいか否かに応じて場合分けをするのが次のステップである。ここで、計算機の中の演算は有限桁の小数として実行されることに注意しなければいけない。この演習で用いる “`1.0d+0`” の形式の数 (倍精度実数という) の有効桁数は、多くの計算機で十進数に換算して 15〜16 桁であり、普通の計算には十分な精度であるのだが、最後の方の数値は四捨五入などして丸めてある関係上、演算の順序に依存して計算結果が変ったりすることがある。例えば $\sqrt{3}/\sqrt{2}-\sqrt{6}/2$ を手元の計算機で計算して

みると、0ではなく“-4.62954328E-17”という数字が返って来た。[7] 整数であれば値は離散的なのでこうした問題は起こらないが、実数の演算では小さな計算の誤差が生じることがある。そのため係数aの評価で「`if (a == 0.0d+0) then`」のような比較をすると誤った判断になってしまうことがある。これを防ぐために、0と等しいかを調べる代りに、絶対値が十分小さい正の実数より小さいかどうかを調べるということが普通行なわれる。上の例で「`if (abs(a) < 1.0d-15) then`」などとしているのはそのためである。

さて上のプログラムを実行すると

```
$ ./a.out
 input a and b:
3.0d+0  1.0d+0
 x =     0.33333333333333331
```

のような結果が得られる。下線を引いた部分が自分で入力した部分であり、それ以外の部分が計算機からの出力である。数学的には解は$x = 1/3$なのだが、計算機が示してくれる答はその近似値に過ぎない。計算機は非常に高速かつ間違えずに演算をすることができるが、数学的な概念を扱うことができるわけではなく、有限桁の数の四則演算ができるだけなのだ。以下の問題で「解を求めるプログラムを作れ」というのは近似値を求められるプログラムを作成せよという意味である。

> 基本問題 6: 係数a、b、cを入力して、二次方程式$ax^2 + bx + c = 0$の解を表示するプログラムを作れ。虚数解を持つ時は“No solutions.”と表示すれば良い。また、$a \neq 0$を仮定して良い。(三つの実数a、b、cを読み込んで、判別式$D = b^2 - 4ac$の正負に応じて出力を分岐すれば良い。)

1.5 繰り返し

計算機を用いるメリットは計算が速く処理が正確なことである。たとえば表1のような三角関数の表を作ろうとするときに、電卓を叩いて答えを書き写す作業を人がしたとすると、遅いし、間違えもありそうだし、何よりもその仕事に幸せを感じることができないであろう。そこで計算機の出番なわけである。しかしながら、今まで説明して来たことだけしか使わないとすると、プログラムは、

表 1: 三角関数表

θ(°)	$\sin\theta$	$\cos\theta$	$\tan\theta$	θ(°)	$\sin\theta$	$\cos\theta$	$\tan\theta$
0	0.000000	1.000000	0.000000	50	0.766044	0.642788	1.191754
5	0.087156	0.996195	0.087489	55	0.819152	0.573576	1.428148
10	0.173648	0.984808	0.176327	60	0.866025	0.500000	1.732051
15	0.258819	0.965926	0.267949	65	0.906308	0.422618	2.144507
20	0.342020	0.939693	0.363970	70	0.939693	0.342020	2.747477
25	0.422618	0.906308	0.466308	75	0.965926	0.258819	3.732051
30	0.500000	0.866025	0.577350	80	0.984808	0.173648	5.671282
35	0.573576	0.819152	0.700208	85	0.996195	0.087156	11.430052
40	0.642788	0.766044	0.839100	90	1.000000	0.000000	—
45	0.707107	0.707107	1.000000				

[7] 平方根の割り算を正しく有理化できないとすると、この計算機の頭脳はできの悪い高校生並と言ったところか。

```
      implicit none
      double precision :: theta
c
      theta = (3.14159d+0 / 180.0d+0) * 0
      print *, 0, sin(theta), cos(theta), tan(theta)
      theta = (3.14159d+0 / 180.0d+0) * 5
      print *, 5, sin(theta), cos(theta), tan(theta)
      theta = (3.14159d+0 / 180.0d+0) * 10
 ...
      theta = (3.14159d+0 / 180.0d+0) * 85
      print *, 85, sin(theta), cos(theta), tan(theta)
      theta = (3.14159d+0 / 180.0d+0) * 90
      print *, 90, sin(theta), cos(theta), tan(theta)
c
      end
```

とこんな風になるだろう。FORTRAN 言語の三角関数は弧度法 (180° を π と表現する角度の大きさの表現法) を用いるので、度数法の 0°, 5°, 10°, ... を弧度法に変換した値を変数 "theta" に代入して、三角関数の値を計算している。コピー&ペーストをしたので全ての文字を打鍵したわけではないが、電卓を叩くよりも尚更手間がかかりそうである。こんな悲しいことにならないために、FORTRAN 言語には同じ処理を繰り返すように指示する命令がある。

```
do 整数変数名 = 初期値, 終了値
    繰り返す操作
end do
```

という構文を用いる。指定された変数名に初期値から始まって終了値まで、値を一つずつ増やしながら "end do" までに書かれた操作を繰り返す。ここでも if 文のときと同じように、繰り返される操作は複数行にわたっても構わない。プログラムを書いていてコピー&ペーストをしたくなるような問題には、この構文を用いるのが良い。

例えば、

```
      implicit none
      integer :: i
      do i = 1, 5
        print *, 'Good morning, honey.'
        print *, 'I love you.'
      end do
      end
```

とプログラムすると、愛する人への朝の挨拶を自動化できる。[8] 繰り返した回数を数えておくための変数 "i" は「個数や順番」にあたるので、「integer :: i」と整数型に宣言されている。上の例では同じ文面を 5 回、10 行にわたって出力する。

> 基本問題 7: "How many times shall I love you ?" と尋ねて回数の入力を求め、入力された回数だけ "I love you." を繰返し出力するプログラム。

また、繰り返しの回数を数えるための変数は、「繰り返す操作」のところで利用することもできる。これらの知識を用いて、三角関数表作りのプログラムを do 文を用いて書き直すと、

```
      implicit none
      integer :: i
      double precision :: theta
c
```

[8] こうした手抜きをしていると、定年退職した後に ...

```
      do i = 0, 18
        theta = (3.14159d+0 / 180.0d+0) * (i * 5)
        print *, (i * 5), sin(theta), cos(theta), tan(theta)
      end do
c
      end
```

となる。先程の例では、0, 5, 10, ..., 90 と定数が書いてあった部分が “`(i * 5)`” という変数を用いた式に置き換わっており、19 回繰り返されていた文が一回のみになっていることを確認して欲しい。ここでも `if` 文のときと同様に、繰り返しの範囲を判別し易いように字下げが行なわれている。

> 基本問題 8: 上の例を真似して、0 ° から 3 ° 刻みで 45 ° まで $\sin\theta$, $\cos\theta$, 及び $\sin^2\theta + \cos^2\theta$ を計算するプログラムを作れ。

> 基本問題 9: 1 から 1000 までの逆数の和
>
> $$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{1000}$$
>
> を求めるプログラムを作れ。(答えは 7.48547 に近い数になる。)

ヒント

単純に和を求めるなら

```
      Total = 1.0d+0 / 1.0d+0 + 1.0d+0 / 2.0d+0
     $  + 1.0d+0 / 3.0d+0 + 1.0d+0 / 4.0d+0 + 1.0d+0 / 5.0d+0 ...
      print *, Total
```

という手もあるのだが、1000 の逆数まで和を求めるとなるとプログラムの入力に少々手間がかかる。計算機言語を用いて数列の和を求める際の常套手段として

```
      T = 0.0d+0
      T = T + 1.0d+0 /    1.0d+0
      T = T + 1.0d+0 /    2.0d+0
      T = T + 1.0d+0 /    3.0d+0
         ...
      T = T + 1.0d+0 /  999.0d+0
      T = T + 1.0d+0 / 1000.0d+0
c
      print *, T
```

こんな方法がある。ポイントは、

i) 和を格納するための変数 “`T`” を用意して、0 で初期化

ii) 「`T = T + 1.0d+0 / ...`」というタイプの文を繰り返す

という二点である。最初に一回だけ行なう (i) の初期化については、FORTRAN 言語の仕様上、「`double precision :: T`」と宣言した時点では、変数 “`T`” の中にどんな値が入っているか不定なので、必ず行なうべきである。経験上は宣言されたばかりの変数には 0 が入っていることも多いのだが、いつもそうであることは保障されていないのである。二番目の繰り返しの部分については、「`T = T + 1.0d+0`」のような文が数学的な意味での等号ではなく、代入命令であるという点をもう一度確認しておこう。こちらの和の求め方だと、同じ形式の文が繰り返されているので、容易に `do` ループを用いた方法に書換えることができる。

漸化式「$a_{n+1} = a_n + a_{n-1}$　但し $a_0 = 0$、$a_1 = 1$」で与えられる数列を Fibonacci(フィボナッチ)数列と言う。最初の何項かを書けば $\{0, 1, 1, 2, 3, 5, 8, 13, \ldots\}$ である。漸化式の形は単純であるが、一般項を求めようとすると少し苦労する不思議な数列である。始めのうちは気付かないが、n が大きくなると急激に増加して膨大な数になる。例えば 100 項目は $a_{100} = 354,224,848,179,261,915,075$ である。

基本問題 10: Fibonacci 数列の第 n 項 (a_n) を求めるプログラムを作れ。n が大きいと問題が起こるので、n は 40 以下の数として良い。(n を入力して、必要な回数だけ「`c = b + a`」と「`a = b, b = c`」の操作を繰り返せば良い。)

ヒント

まず下のような「手際の悪い」プログラムを作ってみて、変数に記憶されている数値がどう変化して行くかを追ってみる。変数 `c` の中身は Fibonacci 列になりそうである。このプログラムは 5 ～ 7 行の処理を繰り返しているので、`do` 文を用いて書き直すことを考える。

行番号	プログラム	変数			
1	`implicit none`				
2	`integer :: a, b, c`	a	b	c	
3	`a = 0`	0	—	—	
4	`b = 1`	0	1	—	
5	`c = b + a`	0	1	1	
6	`a = b`	1	1	1	
7	`b = c`	1	1	①	$\leftarrow a_2$
8	`c = b + a`	1	1	2	
9	`a = b`	1	1	2	
10	`b = c`	1	2	②	$\leftarrow a_3$
11	`c = b + a`	1	2	3	
12	`a = b`	2	2	3	
13	`b = c`	2	3	③	$\leftarrow a_4$
14	`c = b + a`	2	3	5	
15	`a = b`	3	3	5	
16	`b = c`	3	5	⑤	$\leftarrow a_5$
17	`c = b + a`	3	5	8	
18	`a = b`	5	5	8	
19	`b = c`	5	8	⑧	$\leftarrow a_6$
20	`c = b + a`	5	8	13	
	...				

プログラムが思った通りに動いてくれないときには、この様に紙の上で変数の中身がどのように変って行くか追い掛けてみると良い。

計算を続ける内に、指定された回数だけ処理を繰り返す必要がないことがわかり、繰り返し処理を中止したいような場合もある。こんな時には `exit` 文を用いる。`exit` 命令に出会うと繰り返し計算はそこで終わり、計算の流れは `end do` 文の次の文へと向う。`exit` 文は、その役割を考えれば容易に理解できることだが、必ず先程述べた `if` ブロックの中に現れるはずである。もし "`if`" と "`end if`" の間以外に "`exit`" が出現したら、頭の中を整理した方が良い場合が多い。

基本問題 11: Fibonacci 列 $\{a_n\}$ が最初に 100000 を越えるのは n が幾つの時かを求めるプログラムを作れ。

基本問題 12: 基本問題 9 の級数は発散することが知られているが、各項の符号を交代に変えた交代級数

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \cdots$$

は $\log_e 2$ に収束する。整数 n を入力して、この級数の第 n 項までの和を求めるプログラムを作れ。

1.6 配列の利用

計算機を用いた計算には、行列の演算、それも巨大な行列を扱うものも多い。行列の要素に一つ一つ別々の変数名を付けようとすると、面倒だし do 文による繰り返し処理もできない。FORTRAN 言語には配列という仕組みがあって、変数を順番に処理することができるようになっている。プログラム先頭の変数の宣言 (integer 文や double precision 文) のところで、変数名の後ろに括弧でくくった整数定数を付けて配列であることを示し、利用 (値の代入や参照) するときは、変数名の後ろに括弧でくくった整数定数や整数変数を付けて利用する。言葉で説明しても分りずらいので、以下に例を示す。[9]

```
      implicit none
      integer :: i
      double precision :: x(10), y(10)
c
      do i = 1, 10
        x(i) = dble(i)
        y(i) = sqrt(x(i))
        print *, x(i), y(i)
      end do
c
      end
```

宣言文「double precision :: x(10), y(10)」によって、変数 x と y は 10 個の要素を持つ一次元の配列であることをプログラムに伝える。コンパイル時には配列のサイズが決まっていなければいけない決まりなので、宣言文中では (変数ではない) 定数の整数を用いる。計算処理の中で、i 番目の要素に値を代入したり、記憶されている値を読み出したりする必要が生じたときには、普通の変数を書くべきところに "x(i)" と書くことで、普通の変数と同じように扱うことができる。添字のところには定数を書いても良いし、上の例のように整数型の変数や "2 * i + 1" のような式を書いても良い。

行列の計算をするときのように 2 次元の配列が欲しいときには、宣言文のところで二つの整数の定数をカンマで区切って並べて要素の数を示す。例を示すと、「double precision :: a(10,10)」のように宣言するということである。値の代入をするなど変数として利用するときは、一次元の場合と同様に、"a(i,j)" や "a(2,i)" のように整定数か整数型の変数をカンマで区切って要素を特定する。

基本問題 13: 5 × 5 行列を宣言して、単位行列と零行列を作れ。用意した行列の要素を任意の形式で良いので表示するようにせよ。行列風に 5 × 5 の形で表示すると、小数点以下の桁が長過ぎて一行に収まらないなど書式に問題はあるかも知れないが構わない。

[9] サンプルプログラム中に現われる "dble(i)" というのは、整数を倍精度実数に変換する組込み関数である。人間は「4」と「4.0d+0」を区別しないが、計算機中では整数と実数は異なる内部表現を持つので、この変換を明示的に行なう関数である。ただし、この文脈では「x(i) = i」と書いたとしても、計算機は左辺の "x(i)" が実数で、右辺の "i" は整数であることを知っているので、型の変換は自動的に行なわれる。組込み関数 "dble" が必要になるのは、"sqrt(dble(3))" などのように関数呼出しのときである。ここでは整数と実数の区別 (基本問題 2) を思い出してもらうために敢えて用いた。

ヒント

零行列を用意するなら、「double precision :: a(5,5)」と 5×5 の二次元行列を宣言した後に、「a(1,1) = 0」, 「a(1,2) = 0」, 「a(1,3) = 0」, ... 「a(5,5) = 0」と配列要素に順番に値 0 を代入すればよい。ここで添字の変化の仕方を見ると、$(1, 1) \to (1, 2) \to (1, 3) \ldots$ という具合に前の添字が固定されたまま後ろの添字だけが1から5まで変化したのちに、前の添字が1増加するという過程を繰り返している。変化していく添字が二つあるので‘i” と “j” という具合に二つの整数変数を導入すると良いだろう。do 文を用いた繰り返しの構文は二段重ねにすることもできる。例を示すと、

```
do i = 1, 5
  do j = 1, 5
    a(i,j) = 0
  end do
end do
```

とすれば、25 個の配列要素に 0 を代入することができる。単位行列の場合は i == j の時には 1、それ以外の場合は 0 を代入するように、if 文を用いたプログラミングをすれば良い。

この演習では出力の体裁にはこだわらない方針ではあるが、行列風の出力をしたいという要望も根強くあるので、「最小限の知識」からの逸脱をもう一つ紹介しよう。行列の出力の部分を

```
do i = 1, 5
  do j = 1, 5
    print *, a(i,j)
  end do
end do
```

と書いてしまうと、行列の要素が縦方向に一列に出力されてしまって甚だ体裁が悪い。

```
do i = 1, 5
  print *, a(i,1), a(i,2), a(i,3), a(i,4), a(i,5)
end do
```

と書けば行列風にはなるけれど、プログラムのタイピングが面倒だし、任意の次元の行列に対応していないのも癪に触る。ミスタイピングも心配である。こんな時

```
do i = 1, 5
    print *, (a(i,j), j = 1, 5)
end do
```

と書くと、変数 “j” の所を 1 から 5 に順に置き換えた表現を横に並べたように出力してくれる (説明の仕方が絶望的に下手だが、要するにやりたいことをしてくれるということである)。並べたい変数の後ろにカンマ (“,”) を置いて、繰り返しを指示する do 構文の範囲を指定する部分の表現 (“j = 1, 5”) を付け加えて、全体を括弧でくくる。このような表記法は「do 型並び」による出力と呼ばれる。この書き方は最小限の知識からははみ出しているので、設問への解答には用いなくても構わない。

1.7 関数の呼出し II (外部関数の利用)

1.4 節では三角関数などの予め用意されている関数 (組込み関数) の利用法について述べた。この節のテーマは自分で定義する関数 (外部関数) の利用である。実はこれまでに述べて来たことを組み合わせると、計算機にできることは殆ど実行させることができるのだが、複雑なプログラムを書くようになったとき外部関数が使えると見通しが遥かに良くなる。

例として、実数xを与えるとx^2を返す外部関数squard(x)を定義して、利用するサンプルプログラムを以下に示す。

```
c
      function squard(x)
      implicit none
      double precision :: x, squard
      squard = x ** 2
      end
c
      implicit none
      integer :: i
      double precision :: x, squard
      do i = 1, 20
        x = dble(i)
        print *, x, squard(x)
      end do
      end
```

外部関数の定義はfunction文から始めて、end文で終わる。中身は普通のプログラムと同じであるが、関数が返す値を関数と同じ名前の変数に代入するという点に注意する。すなわち「double precision :: x, squard」や「squard = x ** 2」のように、外部関数の中では関数自身の名前“squard”を変数のように扱っている。

外部関数を呼出す側のプログラムでは、関数の名前“squard”が実数型であることを宣言してから“squard(x)”という形で利用している。組込み関数では型は宣言しなくても計算機が知っていたが、自分で定義する外部関数では型を教えてやらなければならないのである。それ以外の点では、外部関数は“sin(x)”や“log(x)”と変らない。[10]

関数の最後まで行かずに途中で帰りたくなることもある。平方根を計算する関数に負の数を与えてしまったようなときは、無理に計算をせずに不適切な入力があったことを報告してすぐ帰る方が良い。また、例えばGauss関数($G(x) = \exp(-x^2)$)を計算する外部関数を作るとき、引数“x”の値が20を越える数が渡されると、関数の値は$\exp(-20^2) \sim 10^{-174}$と、とてつもなく小さい数なので、本当に計算をせずに0を返して置こうと考えるような場合もある。そのようなときには“return”命令を用いる。

```
      function gauss(x)
      implicit none
      double precision :: x, gauss
      if (x < -20 .or. x > 20) then
        gauss = 0.0d+0
        return
      endif
...
```

xとして絶対値が20より大きい値を与えられると、関数の値gaussに0.0d+0を代入して、return文から先に記述してある命令は実行しないで、外部関数はすぐに終了する。

今までの例では外部関数に渡す引数[11]は変数“x”一つだけであったが、複数の変数を渡すこともできるし、定数や配列の名前を渡しても良い。また、上の例では呼出す側も呼ばれる側も同じ変数名“x”を用いていたが、型が同じ(配列の場合はサイズも同じ)でさえあれば、外部関数と呼出し側とで引数の変数名は異なっていても良い。逆に外部関数と呼出し側で同じ変数名を用いたとしても、その変数名

[10] このサンプルプログラムにも型変換の組込み関数“dble”が出て来た。ここでも代入により型変換は自動的に行なわれるので、「x = dble(i)」という代入文は「x = i」でも良い。しかしながら、変数“x”へ代入は無駄なものだと考えて、次の行にある「print *, x, squard(x)」を「print *, i, squard(i)」と書いてしまうと、整数型と実数型は内部表現が異なるので計算はうまく行かない。正しい型の引数を与えてやらなければいけないという点は、ここで説明した外部関数も組込み関数も同じである。

[11] 式“squard(x)”の中の“x”のように関数にデータを渡すための変数のことを「引数」と言う。

が引数に現われなければデータのやりとりは行われない。

```
      function f(x, y)
      integer :: x, y, f
      f = x + y
      end
c
      integer :: x, y, f
      x = 4
      y = 7
      print *, f(y, 1)
      end
```

このサンプルプログラムでは、関数 “f” に渡される引数は “y” と 1 であり、外部関数 “f” の中ではこれが “x” と “y” と言う名前の変数で引継がれて足し合わされる。呼出した側の主プログラムでは “x” と “y” には 4 と 7 が入っているが、外部関数の中では “x” と “y” には 7 と 1 がそれぞれ入ることになるので、出力される数字は 11 ではなく 8 である。

またFORTRAN言語では、引数に与えられた変数や配列の内容を関数の内部で書換えると、関数を呼び出した側にもその変更が伝わるという副作用がある。

```
      function inc(x)
...
      inc = 0
      x = x + 1
      end
c
      integer :: x, inc
      x = 4
      print *, inc(x)
      print *, x
...
```

この例では関数 “inc” 自体の値は 0 なので、まず 0 が印字される。変数 “x” の値は関数の内部で 1 増えるので、次に出力されるのは 4 ではなく 5 である。この性質を利用して、何度も繰り返される少々面倒な手続きを外部関数にしてしまうことができる。こうして定義された外部関数は更にまた別の関数を呼出すこともできるし、一つの外部関数を何度呼出しても構わない。難しそうな問題も小さい処理に分けて次々に解いて行くならば、いずれは解くことができるので、外部関数を上手に使えれば応用範囲が広がる。

> 基本問題 14: Fibonacci 列の一般項は α、β を二次方程式 $x^2 = x + 1$ の解として
>
> $$a_n = \frac{\alpha^n - \beta^n}{\alpha - \beta}$$
>
> と書ける。整数 n を与えると Fibonacci 列の一般項 a_n を返す関数 `Fbncc(n)` を作って、Fibonacci 列の最初の 20 項を表示するプログラムを作れ。

外部関数を用いたプログラミングの例として、組合せの場合の数、${}_nC_r$ について考える。${}_nC_r$ は統計学や確率論を勉強すると出会う記号であり、n 個の中から r 個を選ぶ場合の数は、

$$ {}_nC_r \equiv \frac{n!}{(n-r)!\, r!}$$

と計算できる。ここで、感嘆符の付いた $n!$ は n の階乗、則ち $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$ を表わす。階乗の計算は n が少し大きくなるとすぐにオーバーフローを起すので、${}_nC_r$ の計算には Pascal の三角形 (図 1) を使ってみよう。

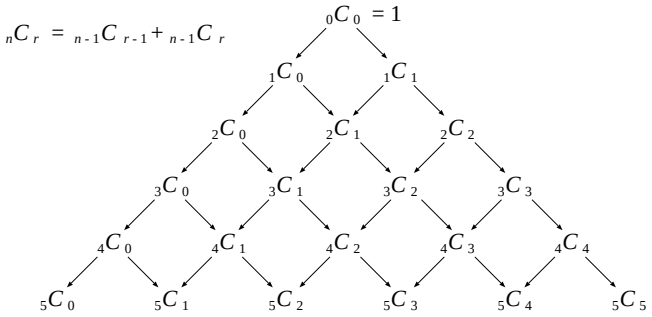


図 1: Pascal の三角形による ${}_nC_r$ の計算。漸化式 ${}_nC_r = {}_{n-1}C_{r-1} + {}_{n-1}C_r$ が成立するので、矢印の根本（ねもと）の数字を足し合せることで矢印の先の数を計算できる。${}_0C_0 = 1$ から始めて、順に ${}_1C_0$, ${}_1C_1$, ${}_2C_0$, ${}_2C_1$, ${}_2C_2$, ... を求めることで、全ての ${}_nC_r$ が得られる。

Pascal の三角形の 2 段目から始めて順に大きな n についての場合の数を求める関数は、例えば以下のように書ける。なお 33 はこの方法でオーバーフローを起こさない限界である。範囲外の入力をすると 0 を返す。

```
      function cmb(n, r)
      implicit none
      integer :: n, r, i, j, C(33,34)
c
      if (n > 33 .or. r > n) then
        cmb = 0
        return
      else if (r == 0 .or. r == n) then
        cmb = 1
        return
      end if
c
      C(1, 1) = 1
      C(1, 2) = 1
      do i = 2, n
        C(i, 1) = 1
        do j = 2, i
          C(i, j) = C(i - 1, j - 1) + C(i - 1, j)
        end do
        C(i, i + 1) = 1
      end do
      cmb = C(n, r + 1)
      return
      end
```

一番内側のループの計算式は漸化式と全く同じ形をしている。但し、FORTRAN 語の配列は 1 から始まるが、${}_nC_r$ の n や r は 0 の値を取ることもあるので、プログラム中の変数 "`C(n, r)`" は ${}_nC_{r-1}$ に対応している。それで値を返す時に 1 だけずれた場所の値を返している。美しくないので、最後の「最小限の知識」からの逸脱をしよう。配列宣言の際に「`integer :: C(0:33)`」のようにコロン (:) を用いて配列の添字の下限を指定できる。入力が面倒なので、上のサンプルの外部関数はダウンロードできるようにしておくが、その中ではこのテクニックを使っている。

> 基本問題 15: 整数 n を与えると和 ${}_nC_0 + {}_{n-1}C_1 + {}_{n-2}C_2 + \ldots + {}_{[(n+1)/2]}C_{[n/2]}$ を計算するプログラムを作れ。ここで $[x]$ は実数 x の小数部分を切り捨てて得られる整数である。答は Fibonacci 列になっているそうだ。

1.8 まとめ

以上でプログラミング入門を終わる。重要事項の復習をすると、

1. 整数と実数
2. 変数と配列
3. 組み込み関数
4. do 文と if 文
5. 入出力
6. 外部関数

これだけ知っていれば、あとは努力と根気で様々なことを計算機にさせることができる。ただし、途中でも述べたように、これだけ知っていれば一通りのことはできるという最低限の線を狙って説明をしたので、説明していないことも沢山ある。また「普通の」FORTRAN 言語の習慣と異なる所もある。意欲のある人はテキストを読んで勉強すると良い。

付録　よく使う関数

しばしば登場する組み込み関数をまとめておく。関数には、整数を引数として与えることのできないものもあるので、注意しなければならない。以下の一覧で "x" は実数限定であり、"a" と "b" は整数でも実数でもよい。

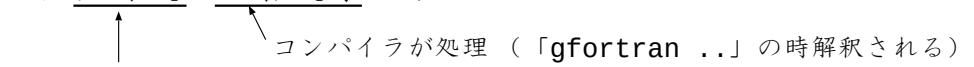
`dble(a)`	実数へ変換する
`int(x)`	整数へ変換する (小数部分は切り捨てられる)
`aint(x)`	実数の小数部分を切り捨てる
`nint(x)`	実数の小数部分を四捨五入する
`abs(a)`	絶対値 $\|x\|$
`mod(a,b)`	a を b で割った時の余り
`sqrt(x)`	平方根 $\sqrt{x}$ $(x \geq 0)$
`exp(x)`	指数関数 e^x
`log(x)`	自然対数 $\log x$ $(x > 0)$
`log10(x)`	常用対数
`sin(x)`	三角関数 $\sin x$ (引数はラジアン)
`cos(x)`	三角関数
`tan(x)`	三角関数
`asin(x)`	逆三角関数 $\arcsin x$ (sin(x) の逆関数, $-1 \leq x \leq 1$)
`acos(x)`	逆三角関数
`atan(x)`	逆三角関数 (円周率は `4.0d+0 * atan(1.0d+0)` で計算できる)

Fortran 文法のまとめ

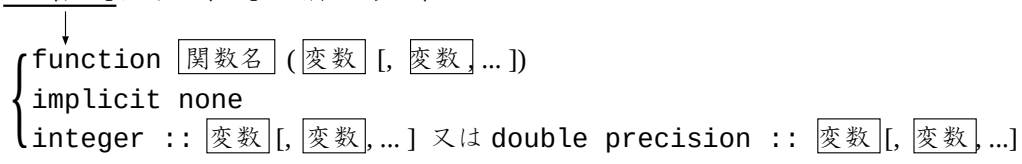
1) プログラムは「文」からなる

原則として一行に一つの文を書く。（ただし継続行という手はある）

2) 文には実行文と宣言文等がある

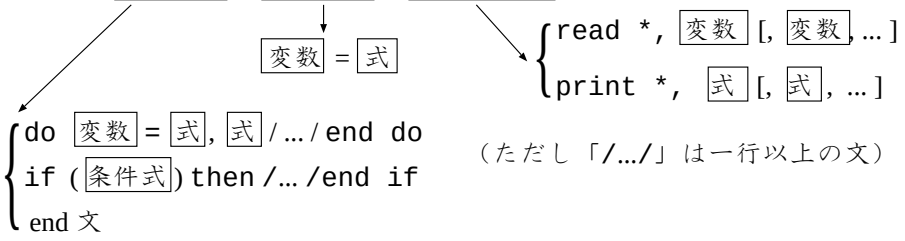


3) 宣言文は実行文の前に置く



ここで 関数名 や 変数 はアルファベットと数字をつなげて作る任意の名前
（但し名前の先頭は数字ではいけない）

4) 実行文には制御文、代入文、入出力文がある



（ただし「/…/」は一行以上の文）

ここで「式」と呼ぶのは（再帰的に定義すると）

- 定数または（配列要素を含む）変数 （例 「5」,「3.14」,「i」,「a(2)」）
- 式を(算術)演算子で繋いだもの （例 「1+1」,「sin(x)+2」）
- 式を括弧でくくったもの （例 「(1+3)」,「(sin(x)+1)」）
- 関数名 (式 [, 式, 式, …]) （例 「fn(pi)」,「mod(int(3*sin(x)),2)」）

のこと。つまり、代入文（例 「i=n-1」）はここで言う「式」ではない。
また print 文では、式 の代わりに「’ (シングルクォート)」で囲んだ
固定されたメッセージを表示させることもできる。

5) 実行文は制御文で指示されない限り、上から下へ実行される

6) 整数と実数は区別される

7) 配列は変数名の後ろに整数を括弧でくくって表現する