データ解析のための Fortran 90 / 95

```
subroutine plinec_xy(xy_array, n, fillcolor, hatch)
  type(coord_xy), intent(in)    :: xy_array(:)
  integer, intent(in), optional :: n
  integer, intent(in), optional :: fillcolor
  integer, intent(in), optional :: hatch

  integer                       :: fillcolor_
  real                          :: x_array(size(xy_array)), y_array(size(xy_array))

  x_array(:)=xy_array(:)%x
  y_array(:)=xy_array(:)%y

  if(present(fillcolor)) then
    fillcolor_=fillcolor
  else
    fillcolor_=-99
```

■　は　じ　め　に　■

1. この冊子の対象とする読者と目的

この冊子は、FORTRAN77 およびそれ以前の FORTRAN について知識のない初心者を対象として、データ解析を行うために必要な、Fortran90/95 の初歩を解説するためのものである。この方針により、最初から Fortran90/95 の様式に沿ったプログラミングを行い、FORTRAN77 からの移行については考慮していない。

また、この冊子は、Fortran90/95 によるデータの解析を主な計算の目的として想定している。そこで、初心者の習得への負担を軽くするため、データ解析に必ずしも必要でないと判断した項目は、省略する、ごく簡単にしか触れない、後回しにする、などの対応を行っている。たとえば以下の項目が該当する。

○ Fortran90/95 の廃止予定事項 (文関数、算術 `if` 文、割り当て型 `goto` 文 etc.)
○ 暗黙の型宣言、`format` 文
○ `pointer` 文、`target` 文、ポインタ代入文
○ `entry` 文、`namelist` 文
○ `rewind` 文、`endfile` 文

2. 適用する環境

この冊子は、IBM 互換 PC 上で稼働する Fortran90/95 の規格に従った Fortran コンパイラを対象として記述されている。そしてその中でコンパイラに依存しない汎用性を保つために、Fortran90/95 の規格に沿った記述を原則とする。すなわちそれぞれのメーカの提供する個々のコンパイラの持つ、`include` などの便利な機能は使用しない。また、コンパイラによっては省略できる記述も多いが、これらも省略しない。

これにより多くの場合プログラムは冗長になるが、可搬性を保つためにあえてそのように記述する。ただしこの冗長性は、演習問題のような短いプログラムでは目立っても、実際に長いプログラムを組むときにはほとんど問題にならない。さらにいえば、冗長性はソース・コードのみの問題であり、実行プログラムの効率はほとんどの場合低下しないであろう。

3. 冊子の性格

この冊子は、読者が演習問題を解くことにより、Fortran を習得することを目的とした自習書である。そのため、Fortran のすべての構文について解説しているわけではないし、対象となる構文のすべての事項について記述しているわけでもない。これらの文法の詳細について知りたいときは、Reference に紹介する文法書を参照していただきたい。

4. 冊子の構成

冊子は三部に分かれる。

・第一部の「導入編」では、プログラムの基本操作と、共通的なプログラミング規則について述べる。
・第二部の「基礎編」では、それぞれの文法について解説していく。
・第三部の「応用編」(予定) では、いくつかの応用問題について詳説する。

5. 冊子中の記号

`[...]` で囲まれた部分は、省略することができる (省略することによって、意味が変わることがある)。

`{...|...}`となっている部分は、いずれかを選択しなければならないことを意味する。

■　目　次　■

□　導入編

第 1 章　Fortran90/95 について

第 2 章　Fortran プログラミングの基本規則

□　基礎編

第 3 章　プログラム構造

第 4 章　配列

第 5 章　サブルーチンとモジュール

第 6 章　データの型と内部表現

第 7 章　入出力

第 8 章　ユーザ定義関数

□　応用編

■　第 I 部　導入編　■

第 1 章　Fortran90/95 について

§ 1.1 計算機の変化とプログラム

1. 機械語とプログラミング言語

現代の計算機は、CPU に対するバイナリー・コードで書かれた命令 (インストラクション) を主メモリに格納して、逐次実行していく。[*1] この命令を言語になぞらえて、**機械語** という。

機械語は個々の計算機の能力を最大限に発揮できるが、日常言語や数学記号とはかけ離れているので、人間が直接に機械語を記述するには非常な修練を要する。さらに、命令と動作の対応は CPU ごとに異なるので、機械語も CPU ごとに異なる。したがって計算機を更新するたびに書き直さなければならない。これはヒューマン・コストの点で大きな無駄である。

このため、人間が日常言語に近いコードを記述し、それを CPU ごとに異なる機械語に翻訳するという方式が生まれた。これが**プログラミング言語** である。この方式によって、それぞれの計算機について一旦プログラミング言語を開発すれば、他のユーザは機械語を学ぶ必要はなくなった。

2. オペレーティング・システム

プログラミング言語の普及について、大きな役割を果たしたのがオペレーティング・システム (略称 OS) の導入である。これはプログラミング言語の下位に位置し、システム・コール (Windows) あるいはカーネル・コール (Unix) という命令でハードウェアを制御する。OS によりそれぞれの計算機のハードウェアの相違は吸収され、プログラミング言語はそれぞれの OS ごとに開発すればよいということになった。

初期の計算機の OS はメーカごとに異なっていたが、UNIX などの汎用 OS が開発されたことにより、同系統の CPU で動作する計算機ならば、一つの OS で稼働することになった。

3. パーソナル・コンピュータの登場

1970 年代に登場したパーソナル・コンピュータ (PC) は、コンピュータの利用層に根本的な変化をもたらした。すなわち計算機は一部の技術者や研究者が使用するためのものではなく、誰でも使え、また使わされるものとなった。これに対応して、ソフトウェアの内容も変化した。すなわち、特定の目的のためにベンダーがプログラム (アプリケーション) を製品として販売し、それをユーザが購入して PC 上で利用するという形態が主流になった。もはやワープロなどのアプリケーションを使うだけの一般ユーザは、プログラミング言語を学ぶ必要はなくなったのである。

4. 現在のプログラミング言語の必要性

それにもかかわらず、現在でもプログラミング言語を学ぶ必要がある状況は依然として存在する。アプリケーションを開発する開発者自身は当然であるが、それに加えて、研究者も場合によってはプログラミングを行う必要があるのである。その理由は、研究はその性格上、一般ユーザの使わない特殊な計算を要することが多いが、それはベンダーの商業ベースに乗らないからである。

[*1] この方式をはじめて採用したのはフォン・ノイマンであり、プログラム内蔵方式という。

§1.2 Fortran 小史

1. IBM 系 FOTRAN

FORTRAN は 1957 年に、計算機メーカである IBM 社のジョン・バッカス (John Backus) が中心になって開発された、最初の科学技術計算用プログラミング言語である。その後も IBM 社によって改良が加えられ、1962 年の FORTRAN IV により完成をみる。この FORTRAN IV が事実上の業界標準となり、他社の汎用機でもこれに準じた FORTRAN が作成されるようになる。

2. FORTRAN66

このように複数の FORTRAN が供給されるようになると、その互換性が問題になる。そこでユーザの便宜を図るため、1965 年に ISO(国際標準化機構) がその最低限満たすべき規格を定めた。この規格に従う FORTRAN を **FORTRAN66** という。また、これを受けて日本でも 1967 年に JIS(日本工業規格) により、JIS-FORTRAN が定められた。*2

3. FORTRAN77

次の FORTRAN の改定は、1977 年に ANSI(アメリカ規格協会) によって行われ、それを受けて ISO、JIS の規格も改定された。この改定では、文字型と論理型、一部の構文の構造化などが導入された。この規格に従う FORTRAN を **FORTRAN77** という。

FORTRAN77 は FORTRAN66 に対して基本的に上位互換である。すなわち FORTRAN66 で書かれたプログラムは、手直しすることなく FORTRAN77 でも動作する。それゆえに、ユーザは改定によりプログラムが動作しなくなるという問題を回避でき、膨大な科学技術系算用のプログラムが FORTRAN77 で書かれ、ライブラリとして蓄積された。それらは現在でも有用である。

4. Fortran90 *3

FORTRAN77 では、旧バージョンとの互換性がその普及に有利に働いたが、これはよいことばかりではない。計算機のハード・ソフトの技術進化につれてプログラム言語もそれに対応して変化する必要があるが、互換性を保つという要求は、その変化を阻害するからである。FORTRAN77 でも、プラットフォームが汎用計算機から PC に移行するにつれて、その弊害が目立つようになり、ユーザ数は漸減した。

このような状況の変化に対応し、1991 年に ISO による改定が行われ、それを受けて ANSI、JIS の規格も改定された。この規格に従う FORTRAN を **Fortran90** という。Fortran90 では構造型、ポインタ型、配列計算導入などの大規模な改定が行われ、もはや FORTRAN77 との互換性は完全には保たれていない。とはいえ、FORTRAN77 の資産を捨て去るわけにはいかないので、FORTRAN77 で書かれたプログラムは、Fortran90 のプログラムとは別途に処理して、結合することにより利用できるようになっている。

5. Fortran95

Fortan90 のマイナー改定である。わずかな構文と関数が追加され、いくつかの旧バージョンの構文の使用が廃止された。とはいえ、廃止された機能も独自でサポートしているメーカも多く、Fortran90 との差はきわめて小さい。Fortran90 という名前でも、Fortran95 をサポートしているものが多い。

6. Fortran2003

いくつかのコンパイラ (g95、Intel for windows ver.11) がようやく提供され始めた段階であり、まだ解説書のたぐいは出ていない。2008 年現在では、急いでこちらに移行するほどのメリットは感じられない。

*2 FORTRAN66 の規格を見たことはないが、おそらく内容的には FORTRAN IV の追認に近いものと思われる。FUJITSU や HITACHI の汎用機でも、FORTRAN IV 準拠と表記していた。また、水準 3000 と言うような言い方もあった。

*3 FORTRAN77 以前は大文字で「FORTRAN」、Fortran90 以降は先頭だけ大文字で「Fortran」と表記することになっている。

§1.3 他のプログラム言語について

プログラミング言語には、Fortran 以外にも多くの種類がある。そのうち理科系の計算に使われるものについて、独断と偏見に基づいて、軽く触れることにする。

1. ALGOL(1960)

ALGOL は元々プログラム言語ではない。FORTRAN を開発したジョン・バッカスが 1959 年に発表したバッカスーナウア表記法というアルゴリズム表記法である。翌年言語化されたが、ALGOL を実装した計算機は全く普及しなかった。しかしその論理性により、他の多くの言語の基本となっている。

2. BASIC(1964)

もともとは、FORTRAN になかなかなじめない学生のために Dartmouth 大学で開発された、FORTRAN の簡易版である、会話型のインタープリタ言語である。利用のためのしきいは低かったが、所詮は初心者のための練習用言語であり、できることには限られ、また大規模な計算や厳密な計算は不可能だった。

事情が変わったのは、IBM が発売する PC のシステム言語として、新興の Microsoft 社の BASIC を採用してからである。その後 Microsoft 社の発展に伴い、BASIC は MS-BASIC → Quick-BASIC → Visual BASIC と進化した。その最終版の Visual BASIC6 では、BASIC 本来の特徴であるしきいの低さを保ちつつ計算機能は上昇した。またアプリケーション開発言語としては、プリンタや RS232C などの周辺機器をドライブする事ができるようになり、変形版である VBA は、Excel などのアプリケーションで使われた。

しかし、このような Visual BASIC の成功にもかかわらず、Microsoft 社は突如としてこの系列の提供を放棄し、アプリケーション開発指向の Visaul BASIC 2003 → 2005 → 2008 へと路線変更した。これらは同じ Visual Basic という名でも以前のシリーズとは大きく異なっている、この路線変更により利用のためのしきいが高くなり、計算言語としての利点が失われた一方、アプリ開発言語としては後述する VC++ 等に及ばなかったために、かつてほどは使われないようになった。

3. PL/I(1965)

IBM が、科学技術系の FORTRAN と事務処理系の COBOL とを統合し、一つの言語を学ぶだけですべての仕事ができるようにと開発した、汎用言語である。しかし、もともと一人ですべての仕事をこなす者などきわめてまれだったので、当然のごとく商業的に失敗した。

4. PASCAL(1970)

バッカスーナウア表記法を取り込んだ、ALGOL 直系のプログラミング言語である。開発直後から、アルゴリズム研究者の間では歓迎されたものの、数年間は一般にはなじみのない言語だった。この状況が変わったのは、Borland 社が PC-DOS(MS-DOS) 用に、TURBO PASCAL の販売を開始してからである。

70 年代の終わりに PC の普及が始まったが、当初は手頃なプログラム言語がなく、プログラムは BASIC インタープリタか機械語で組まれていた。BASIC は遅く、機械語は習得が簡単でない。そこで PC-DOS の導入により (一本道ではなかったが)、このような状況が打開されることが期待されていた。そこに現れたのが、コンパイル言語である TURBO PASCAL である。そのアルゴリズム記述の明確さ、学習のしやすさ、およびその実行の速さにより、文字通り一世を風靡するに至った。

しかし、TURBO PASCAL はモジュール化に難があり、大規模なプログラムの開発には向いていなかった。またグラフィックや周辺機器のドライブのような、機械語に近い低レベル操作にも向いていなかった。PC が高性能化し、アプリケーションの需要が多様化するにつれてこのような欠点が不利に働くようになり、C 言語の PC への導入によりその地位を譲った。

現在では TURBO PASCAL の後継として、Delphi が Borland 社から提供されているが、ユーザはきわめて少ない。

5. C(1972)

もともとは Bell 研究所で開発されていたオペレーティング・システム UNIX を記述するための言語である。当然ながら UNIX 系 OS(Solaris, FreeBSD, Linux) との相性はきわめてよい。開発者であるカーニハンとリッチーが「プログラミング言語 C」という本 (通称 k&R) を著し、それをいわば規格書のようにして、PC に C 言語が移植された。

C もまた ALGOL 系のプログラミング言語であり、PASCAL とは親戚 (兄弟というより、いとこ程度) にあたる。PASCAL に比べて、機械語に近い低レベルの操作が可能であり、モジュール化もすぐれ大規模なプログラムも組める。これらの利点から、PC のアプリケーション開発の主流言語となった。ただし多機能な分、使いこなすには時間がかかるという弱点があった。

6. C++(1983)

アプリケーション開発の手法として、オブジェクト指向 (Objecive Oriented Program, 略称 OOP) が提唱され、1972 年に最初の OOP 実用言語、Smalltalk が生まれた。しかし OOP を PC に実現するに当たっては、まずは新しい言語ではなく、C の上位互換として OOP 機能を付け加える形で導入された。それが C++ であり、C++ の出現により、PC、UNIX 機とも K&R の C は完全に C++ に置き換えられた。[*4]

PC の C および C++ については、Window98 の頃までは Microsoft 社の MS-C と、Borland 社の TURBO C(++) が競争していたが、統合環境 Visual Studio で動作する Visual C++(略称 VC++) が Microsoft 社から提供されると、自社 OS の有利を生かした Microsoft 社の一人勝ち状態になった。[*5]

C++ は強力なアプリケーション開発ツールであるが、低レベル言語 (機械語に近い言語) であるので、FORTRAN のように計算を主とする仕事もできる。そこでもう FORTRAN をやめて C++ で数値計算を行おうと言う動きもある。[*6] ただし、かなり習得の難しい言語であることがネックになって、主流にはなっていない。

7. awk(1977) および Perl(1986)

これらはプログラミング言語ではなく、スクリプト言語である。スクリプト言語とは、主メモリに常駐し、コマンドで行うには複雑すぎるが、わざわざプログラミング言語を呼び出すまでもない処理を行うものである (たとえば「行の 1 字目がブランクならば 999 に変える」)。awk より Perl の方が複雑な処理を行えるが、その分覚えなければならないことは多い。データのプロプロセスに有効であるが、UNIX 系列の OS でしか使えない。これを覚えるだけの価値のあるかどうかは、ユーザの行いたい仕事による。

8. Ruby(1986)

(ほぼ唯一) 日本で開発された、軽めのプログラム言語である。特徴としては、

○ OOP 言語である。
○ 個人により非商業ベースで開発され、無料で提供されている。
○ Windows、Linux、MAC-OS といろいろなプラットフォームで動作する。
○ 多バイト文字 (漢字) を標準でサポート。

個人の開発した言語であるから、最初はできることは限られていた。しかしユーザが協力することにより次第にライブラリは充実してきている。現在では fedora のディストリビューションでも標準で提供されている。これらのことから、データのプリプロセスには、大いに有効であると思われる。

[*4] それ以後は、単に「C 言語」といえば、ほとんどの場合「C++」のことを指す。

[*5] Visual C++ 2008 および Visual Basic 2008 は、express エディションが Microsoft 社のサイトから無料でダウンロードできる。興味があれば、試してみるのもよいだろう。

[*6] そのために、FORTRAN のようなライブラリを C++ で構築しようとする試みも行われている。

§1.4 Fortran90/95 コンパイラの準備

1. コンパイラの提供形態

Fortran90/95 のコンパイラは、HP、Intel、NAG などの多くのベンダーより販売されている。すでにこれらが利用可能であれば問題ないが、そうでない場合にはこれから Fortran を学ぼうとするものがすぐに購入するのは得策ではない。高価な上、毎年メインテナンス料がかかるからである。

一方、フリー (無償) のコンパイラには、以下の点で不利がある。

◦ 一般に、利用目的が個人的なものに限られる。[*7]

◦ 説明書が充実していない。バグがあっても、サポートを受けられない。

◦ ベンダーが提供するものは、提供側の都合で無償提供が急に停止されるおそれが常にある。

◦ ベンダーの提供する試用版以外のものは、一般に性能が低い。

使い分けとしては、とりあえずはフリーのコンパイラで Fortran を学習しておき、ある程度習熟した後に、本格的な仕事は有償のコンパイラで始めるのが得策であろう。

2. Windows 上のフリーの Fortran90 コンパイラ

Windows で動作する Fortran90 コンパイラは、現在のところ SilverFrost 社の Ftn95 が唯一と言ってよい。[*8] 次の URL から ftn95_personal をダウンロードしてインストールする。これは個人利用に限られる。

`http://www.silverfrost.com/32/ftn95/ftn95_personal_edition.asp`

統合環境としては VS(Visual Studio) に対応しているが、これを所持していない場合は付属の「`Plato`」を使うとよい。Fortran に関してはほぼ VS と同様の機能が使えるが、日本語の表示はできない。

3. Linux 上のフリーの Fortran90 コンパイラ

一方、Linux で動作する Fortran90 コンパイラは何種類かある。

Intel Fortran90 for Linux

Intel 社が開発する、他の商用コンパイラに比較しても見劣りがしない高性能なコンパイラである。実際フリー版、商用版という区別はなく、営利目的に使うときはこのコンパイラに対してライセンス料を支払うことになる。また、Windows 版はすべて有償である (試用版があるときもある)。[*9] [*10] [*11]

GNU Fortran

GNU プロジェクトで次の二種類の Fortran コンパイラが提供されている。

g95　　: (URL) `htttp://g95.org/`

gfortran : (URL) `http://gcc.gnu.org/wiki/GFortran`

これらは研究目的にも利用できる。また提供の中止など、メーカーの都合に左右されることはない。ただし、日本語の文法解説書がないなど、Fortran の学習者向きではない。

[*7] 他の多くの言語と異なり、教師が授業に使うこと、研究者が研究に使うことも、商業利用とみなされる。

[*8] 実はもう一つあるのだが、試していない。

[*9] Intel Fortran90 はたびたびバージョン・アップを行っており、その際にダウンロード先の URL がわずかに変わる。また、インストール手順も微妙に変わることがある。したがってここでは URL を示さないので、ネットで「Intel Fortran インストール」などをキーワードとして、インストール方法の掲載された Web 記事を検索し、それに従ってインストール作業を行っていただきたい。

[*10] インストールに当たって注意点がいくつかある。まず、ダウンロードすべきファイルは何カ所かのサーバに置かれているが、一時間以内にダウンロードしなければならない。もし、「アジア」のサーバだとタイムアウトするときは、指示を無視して「ワールド」のサーバを使えばダウンロードできることがある。

[*11] さらに、ディストリビューションを REDHAT 系の fedora とした場合、fedora のシステム (カーネル) の更新が頻繁に行われるので、最新のカーネルには対応できなくなることがある。したがって、fedora の旧バージョンをプラットフォームとした方が安全である。それでも、たとえば fedora8 ならば「yum install libstdc++.so.5」として yum で追加インストールしておかなければならない。

§1.5 Fortran プログラムの実行までの流れ

Fortran プログラムを作成し、実行させるまでの一般的流れを以下に示す。

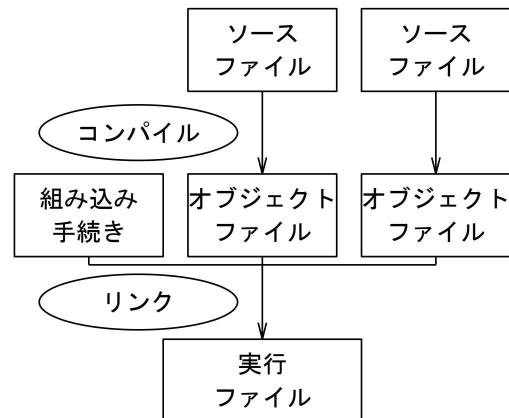


Fig.1.1

1. プログラミング

計算機に実行させるための手順を記述した文書を**プログラム** といい、プログラムを作成する作業を**プログラミング** という。プログラムの実体はテキスト・ファイルであり、テキスト・エディタを使って作成する。Linux なら、KDE または GNOME の標準エディタで十分であるが、Windows の場合 notepad ではなく、Terapad 等のフリーの多機能エディタか、統合環境を使った方がよい。

プログラムを記述したファイルを、ソース・ファイル (ソース・コード等の他の名前で呼ばれることもある) といい。Fortran のプログラムであることを、「`.f90`」などの拡張子をつけることによって区別する。

一般には、プログラムは複数のソースファイルから構成される。また、他者の作成したライブラリ・プログラムをその一部として組み込むこともできる。

2. コンパイル

作成したソース・ファイルを、計算機が実行できる機械語に翻訳することを**コンパイル** といい、そのためのアプリケーションを**コンパイラ** という。コンパイルは一つのソース・ファイルごとに行い、オブジェクト・ファイル (オブジェクト・コードあるいはオブジェクト・モジュール) という機械語ファイルを出力する。オブジェクト・ファイルの拡張子は、Windows では「`.obj`」、Linux では「`.o`」である。

コンパイル時に、ソース・ファイルの種類によっては、適切なオプションを指定しなければならないことがある。また、プログラムを構成する複数のソース・ファイルをコンパイルする場合、コンパイルの順序が重要になることがある。

3. リンク

コンパイラにより生成されたオブジェクト・ファイル (群) に、Fortran の標準手続きを加え、一つのアプリケーションとしてまとめることを**リンク** といい、そのためのアプリケーションをリンケージ・エディタ略して**リンカ** という。リンカは OS 上で実行可能な実行ファイルを出力する。実行ファイルは Windows では「`*.exe`」という拡張子をもつファイルであり、Linux では「`a.out`」という名のファイルである。

かつての Fortran 処理系では、コンパイラとリンカが別々のアプリケーションとして提供されていたが、現在では一つのアプリケーションでコンパイルとリンクを同時に行うものが主流である。したがって単に「Fortran コンパイラ」と言えば、リンク機能も含んでいるものを指すことが多い。本冊子も以後「コンパイラ」と言う語をこの意味で使用する。

4. 実行

Windows2000、Xp の場合は DOS 窓を開き、コマンド・プロンプトから実行ファイル名を、

```
c:\*> *.exe
```

のように入力して実行する。「`*`」の部分には、実際には具体的な名前が入る。

Linux ならは、端末モードでプロンプトの後に、次のようにタイプして実行する (シェルが bash の場合)。

```
localhost$ ./a.out
```

§1.6 実行までの操作手順

Fortran プログラミングの具体的な操作方法は、使用するコンパイラ、OS、統合環境 (これらを合わせて **処理系** という) によってそれぞれ異なる。ここでは例として、
∘ Windows 上で Silverfrost 社の ftn95 を Plato3 統合環境で使う場合
∘ fedora8(Linux) 上で intel Fortran90 for Linux を Gnome 開発環境上で使う場合
の二つの場合について説明する。

実際に以下の手順を実行して、操作方法を覚えていただきたい。 なお、† のある操作は今は行う必要はないが、あとで必要になる操作である。

1. windows(NT,2000,XP) + ftn95 + Plato3 環境の場合

<u>ソース・ファイルの入力</u>

(i) まず、必要ならばあらかじめ作業用フォルダを作成しておく。適当な既存のフォルダがあれば不必要。
(ii) Plato3 のアイコンをクリックして起動し、File → New → Free Format Fortran File を順にクリックして、入力画面を open する。
(iii) 以下のテスト・プログラムを入力する。

テスト・プログラム

```
program test_program
    implicit none

    write(*,*) "Hello"

    stop
end program test_program
```

入力はすべて半角であり、p が 1 カラム (1 文字目) にあたる。また「_」は「-」(マイナス) ではなく、キーボード右下のアンダーバーである。また、第 2,4,6 行の先頭の空白は、Tab キーを使わずに半角スペースを挿入する。全角スペースは不可 (これは発見しにくいエラーを引き起こす)。第 3,5 行は、何も入力せずにすぐに Enter キーを押す。
(iv) File → Save As.. で目的のフォルダに移動し、ファイル名を `test_program.f90` として保存する。

(注) 今は該当しないが、注釈等で日本語を使いたいときには、Plato3 は日本語の表示ができないので、Terapad などの日本語エディタで該当部分を修正する必要がある。

<u>コンパイル + リンク + 実行</u>

統合環境では、このような全体が一つのファイルに収まる短いプログラムならば、一つの操作で連続してコンパイル、リンク、実行を行うことができる。
(i) Build タブ → Start Run を実行する。あるいは Run のアイコン (黒い三角) をクリックする。
エラーがなければ、実行ファイル `test_program.exe` が生成される。この場合オブジェクト・ファイルはいったん生成された後に消去される。
(ii) 引き続いて実行が行われ、DOS 窓が現れ、最初の行に Hello と表示される
(iii) 同時に下の行に「Press RETURN to close window...」と表示されるので、Enter キーを押すと DOS 窓が消えて、元の Plato3 の画面へと戻る。

この場合、コンパイル、リンク、実行の段階でそれぞれエラーが起きる可能性がある。

○ コンパイル段階で致命的な (fatal) エラーがあれば、下部の window にエラー・メッセージが出力され、コンパイラは停止する。一方エラーが致命的でなければ、エラー・メッセージは出力するが、リンクに進む。むしろ気をつけなければならないのはこの場合であり、たとえ正常に実行が終了しても、結果は期待したものと異なることが多い。
○ リンク段階でのエラーはほとんどの場合、エラー・メッセージを出力するだけで実行ファイルが生成されて実行段階へと進み、そこで実行時エラーを引き起こす。
○ 実行段階でエラーがあれば、エラー・メッセージが Plato3 ではなく DOS 窓側に現れる。このとき実行エラーを起こした行の番号が表示されるので、これを参考にエラーの場所を探す。[*12]

○ プログラムが暴走したときには、**ctrl+C で強制的に実行を停止させる。**

エラーがあれば、この場合どこかに打ち間違いがあるので、ソース・ファイルを修正してもう一度 (i) の手順からやり直す。その際、修正後にいったん save する必要がある。(File タブ →Save、またはフロッピー・ディスクのアイコン)。

コンパイルのみ †

複数のファイルからなるプログラムから実行ファイルを作成する場合は、メインのファイルをコンパイル・リンクする前に他のファイルをすべてコンパイルして、それぞれオブジェクトファイルを作成しておく必要がある。[*13] それらのファイルについてはリンク・実行はせずにコンパイルのみを行う。また、ソースファイルに間違いがあることが期待される場合も、コンパイルのみを行うことがある。

以下の操作でオブジェクト・ファイル `test_program.obj` が生成される。
(i) Build タブ → Compile を実行する。

コンパイル + リンク †

後述するリダイレクションによる入出力を行う場合は、実行せずにリンクまでで止めておく必要がある。

なお、リンクに際して他のオブジェクト・ファイルが必要な場合でも、それらを同じ作業フォルダ内に置いておけば、統合環境が探し出してリンクしてくれるので、特にそれらを指定する必要はない。
(i) Build タブ →Build を実行する。エラーがなければ実行ファイル `test_program.exe` が生成される。

実行 †

作成された実行ファイルの実行は DOS 窓で行う。統合環境でも可能ではあるが、かえって面倒である。
(i) Windows のスタートボタンから、プログラム → アクセサリ → コマンド プロンプト で DOS 窓を開く。
(ii) カレント・ディレクトリを実行ファイルのある作業用ディレクトリに移動する。具体的な手順は、
· 作業フォルダのファイルの一つを DOS 窓にドラッグする。
· 絶対パスでのファイル名が表示されるので、後ろから backSpace キーでファイル名までを消す。
· カーソルを先頭まで移動し、`cd␣` (␣は空白を意味する) を挿入し Enter キーを押す。
· 作業フォルダが C ドライブでない場合には、ドライブの変更が必要。たとえば D ドライブなら「`>D:`」
(iii) `test_program` と入力し (`.exe` は不要)、Enter キーを押して実行する。

後処理

プログラミングを行うと、すぐにファイルの数が増える。そして時間がたつとその内容を忘れてしまい、しばしば整理がつかなくなる。これを避けるために、不要なファイルはすぐに消去すべきである。この場合ならば、ソース・ファイル以外の `BuildLog` や `ErrorLog`、実行ファイルなどは消去する。

[*12] このような単純なプログラムでは、エラーを起こした行にその原因があると考えてよい。しかし、もっと込み入ったプログラムでは、エラーの原因は該当行でになく、以前に実行された他の行にあることの方が、むしろ多い。
[*13] モジュールの場合は、モジュール情報・ファイル `*.mod` も必要である。以下同様

2. Linux + intel Fortran + Gnome 統合環境の場合

ソース・ファイルの入力

(i) ftn95 の場合と同様に、適当なディレクトリ (フォルダ) を作成あるいは選択し、ブラウザで開く。
(ii) ブラウザのタブから、ファイル → ドキュメントの生成 → 空のドキュメントを選択し、作成されたファイルの名前を `test_program.f90` に変更する。
(iii) ファイルをダブルクリックして、エディタを起動する。あとは ftn95 と同様に入力し、保存する。

コンパイル + リンク

(i) 上部のバーの一番左から アプリケーション → システムツール → 端末 を選択し、プロンプトを出す。
(ii) 端末 window のカレント・ディレクトリを、ソース・ファイルのあるディレクトリに移動する。この操作は、Windows の場合とにているが、いくらか異なる。

· ソースファイルのあるディレクトリのファイルのどれか一つを、端末 window にドラッグする。
· 両端に「'」がついたフルパス名が表示されるので、後ろから最後の「'」とファイル名を消す。
· ctrl+A キー、または矢印キーで先頭に移動して最初の「'」を消す。
· 先頭に「`cd␣`」を挿入し、Enter を押す。ドライブの変更は必要ない。

(iii) 端末 window に、「`ifort test_program.f90 -check -warn`」と入力して、Enter を押す。成功すれば、実行ファイル「`a.out`」が生成される。
(iv) エラーメッセージが出力された場合、どこかに間違いがあるのでソース・ファイルを修正してもう一度 (iii) の手順を行う。エディタと端末は同時に開けるが、修正後に save することを忘れないように。

コンパイルのみ †

(i) 端末 window に、「`ifort test_program.f90 -c -check -warn`」と入力して、Enter を押す。成功すれば、オブジェクトファイル「`test_program.o`」が生成される。

リンク †

メインのプログラムをコンパイル、リンクする際に、他に必要なオブジェクトファイルがあれば、そのファイル名をすべて、メインのファイル名の後に指定しなければならない。

実行

端末 window に「`./a.out`」と入力してエンターを押す[*14]。成功なら次の行に Hello と表示される。

後始末

Linux ではサイズの大きい実行ファイルが増えるのを避けるために、実行ファイルはすべて `a.out` という名前で生成される。通常はこれを消去してよいが、もし残しておきたい場合には、次の作業で上書きされるのを防ぐために、名前を変えておかねばならない。たとえば `a.exe` という名前で保存しておくならば、「`mv a.out a.exe`」と端末 window からコマンド入力する。

演 習 問 題

演習問題 1

画面に、以下の 2 行を出力するプログラムを作成しなさい。

```
Hello
World
```

解答 テスト・プログラムの第 4 行の後に、「　　`write(*,*) "World"`」という行を挿入する。

[*14] fedora の標準シェルである bash の場合

第 2 章　Fortran プログラミングの基本

§ 2.1 ソース・プログラムの記述規則

プログラム言語には、「言語」という名が表すように、規則にしたがって記述される。その規則を言語の場合と同様に「文法」という。文法は、プログラム言語ごとに異なる。

これから Fortran の文法を学んでいくことになるが、まずはじめにソース・プログラムの記述規則から始めることにする。これは言語で言えば使用される文字や、句読点、改行規則などの基本的な部分にあたるものである。そのために、前述のテスト・プログラムを多少改変した次の例題をもとに説明する。

例題 2-1

次のプログラムを実行し、画面に `Hello!` と出力されることを確認しなさい。

```
1   program example2_01 ! プログラム文 ここから end program 文までがプログラム本体
2   ! first test program（注釈行）
3       implicit none ! とりあえず、おまじないと考えること
4
5       write(*,*) "Hello!" ! 中心となる write 文で、Hello! と出力する。
6
7       stop ! プログラムの停止
8   end program example2_01
```

今回から説明の便宜をはかるために、行番号をプログラムの左端に表示している。プログラムを入力する際には、行番号部分は省略する。たとえば第 1 行の「`program`」は前と同様に 1 カラム目から始める。

各行の「`!`」より後は、後述する注釈なので入力する必要はないが、もし入力するならば、漢字は Plato3 では表示されないので、その部分に関しては、Terapad や NotePad などで入力しなければならない。

1. 文 (*statement*)

Fortran プログラムは、「**文**」の集まりで構成されている。文にはさまざまな種類があり、それぞれ名前がつけられ、固有の記述形式が定められている。コンパイラはそれらの文の記述内容を解釈して、指定された動作を行う。たとえば第 5 行は **write 文**という名前の文であり、`Hello!`という文字列を標準出力に書き出すことを意味する。

構文素

一般に 1 つの文は、空白で区切られたいくつかの語 (構文素) で構成されている。Plato3 あるいは Gnome editor によりプログラムを入力すると、入力した語が色分けされて表示される。この色分けには意味があり、上のプログラムでは、次に述べる注釈部分をのぞき、「`example2_1`」(2 カ所) と「`"Hello!"`」がユーザが自由に書いた部分である。それ以外は、Fortran の文法で意味が定められた語であり、文キーワード (予約語) ともいう。

2. 注釈 (*comment*)

ソース・プログラムには、プログラムをわかりやすくするために「覚え書き・メモ」を書いておくことができる。これを **注釈** という。注釈は Fortran の文ではないので、コンパイラはその部分は処理せずに読み飛ばす。したがって、本来コンパイラが処理できない日本語も、この部分ならば使うことができる。

注釈は形式上、他人に対して書くものであるが、参考書 [9] にあるように、未来の自分は他人と同じなので、自分のためにも詳しく注釈を書いておくことは非常に重要である。実際、この程度の短いプログラムでは、さして注釈の必要性を感じられないだろうが、プログラムが複雑になるにつれて、注釈の重要性は大きくなる。

注釈を書くには、行中に「!」(1 バイト文字、全角文字ではエラーとなる) を挿入すればよい。以後、行末まですべて注釈となる。ただし、「"」と「"」、または「'」と「'」で囲まれた文字列の中にある場合は例外であり、第 5 行の「"Hello!"」の「!」は注釈を表わす働きはしない。

3. 行 (*line*)

「文」を Fortran プログラムの論理的構成単位とするならば、物理的構成単位に相当するものが「**行**」である。他のプログラム言語には、改行は単に見やすくするだけの役割しか持たないものもあるが、Fortran では改行は明白な意味を持つ。とくにデータ・ファイルの場合、1 行に複数のデータを書く場合と、2 行以上に分割して書く場合とでは、結果が異なることがある。*1

1 行の文字数

1 行に書ける文字数は、**132 バイト**である。すなわちすべて半角文字 (1 バイト文字) で書くならば、132 文字である。全角文字 (漢字・記号) を注釈などで入れる場合の文字数は、使われる漢字コード系による。*2 処理系によっては、それ以上の文字数を許容するものもあるが、そのような長い行は書くべきではない。すべての処理系がそのような拡張を行っている訳ではないので、処理系間の可搬性が損なわれる。第一そのような長い行はディスプレイの画面に表示しきれない (折り返せば表示できないことはないが、それは改行が意味を持つ Fortran では望ましくない)。

なお、この制限はソース・ファイルについてのものであり、データ・ファイルの長さに制限はない。

Fortran のプログラムは、1 つの文を 1 行に書くこと、すなわち論理的構成単位と物理的構成単位を一致させることが原則である。ただし、以下のような例外がある。

注釈行・空白行

第 2 行は行全体が注釈からなっていて、文がない。この行は **注釈行** と呼ばれる。また、第 4,6 行も文がない。この行は **空白行** または **空行** と呼ばれる。これらの行は単にプログラムを読みやすくするためだけのものであり、コンパイラは処理を行わずに読み飛ばす。すなわち実行動作には関係しない。

一般的に、注釈行に書く内容は以下のようなものである。

○ プログラムの目的、使われている公式など
○ プログラムの作成日付
○ プログラム作成者の名前
○ プログラムの内容説明

継続行 (*continuation line*)

長い式や定義を書くとき、1 行 132 文字という制限には収まらない場合がある。そのようなときは、**最初の行の終わりに「&」を挿入し、** *3 **次の行の始めにも「&」を挿入する。** これによりコンパイラは最初の行の「&」の直前の文字と、次行の「&」の直後の文字が連続しているものと解釈する。たとえば、

```
write(*,*) "AAAAAAA&
&BBBBBB"
```

*1 これは Fortran が紙カードという、行が明確な意味を持つ時代からある言語であるためである。
*2 Windows で使われる SJIS コード系ならば全角文字は 2 バイト、Fedora で使われる utf8 コード系ならば 3 バイトである。
*3 その後に注釈を入れてはいけない。

というように継続した行は、

```
write(*,*) "AAAAAAABBBBBB"
```

という行と等価である。

ここで最初の行を **開始行** 、次行を**継続行** という。継続行の終わりに「&」を挿入すれば、また次の行に継続することができる。このようにして、継続行は連続して 39 行まで続けられる。

複文

逆に 1 つの行に、「;」(セミコロン) で区切って 2 個以上の文を書くことができる。これによりプログラムの行数を減らすことができるが、場合によっては大変プログラムがわかりにくくなるので、乱用すべきではない。[*4] 使うとしても、次例のように実行順序を変えても結果に影響しない場合に限るべきである。

```
a = 0 ; b = 0
```

4. Fortran で使われる文字

Fortran で使用できる文字は、以下の三種類に大別できる。

(i) 英数字

英語アルファベット 26 文字の大文字と小文字 (英字)、数字の 0 から 9、それにアンダーバー「_」を加えた計 63 文字である。文キーワードおよび各種の「名前」にはこの文字しか使われない。

Fortran では、**英字の大文字、小文字の区別はつけない**。「write」も「Write」も全く同じ意味を表す。[*5]

(ii) 特殊文字

「+」、「-」、「*」、「/」、「=」、「(」、「)」、「.」、「,」、「'」、「:」、「$」、「!」、「"」、「%」、「&」、「;」、「>」、「<」、「?」に空白「␣」を加えた 21 文字である。これらは演算子や書式指定など、特定の用途に使われる。

(iii) その他の文字

「@」、「{」、「|」や漢字などはこのカテゴリーに入る。このカテゴリーの文字は、注釈文や文字データとしてのみ扱うことができる。

名前の命名規則

例題 2-1 の「example2_1」は、ユーザがプログラムに付けた「名前」である。Fortran ではこれ以外にも、変数名、関数名、モジュール名など、いろいろなところでユーザが名前を付ける必要がでてくる。これらの名前に共通する命名規則は、「**英字ではじまる、31 文字以内の英数字**」である。処理系によってはこれより長い変数名も許容されるが、可搬性を損なうため、必ず 31 文字以内に納めるべきである。

なお、ファイル名の命名規則は OS による (Fortran では、上記 (3) カテゴリーの文字型データとして扱われる)。しかし、整理のために、**プログラムファイルの名前は、プログラム名と一致させる**ことを強く推奨する。

演 習 問 題

演習問題 1

以下の文字列で、Fortran の名前として有効なものをすべてあげなさい。

(1) a-b　(2) CO2　(3) _stdout　(4) a.out　(5) 2d　(6) o_1　(7) Ｂ (全角文字)

解答 (2) と (6)

[*4] プログラムの媒体が紙カードの時代には、経費と資源の節約のためによく使われたが、磁気ディスクではその意味がない。

[*5] Windows システムも同じく大文字、小文字を区別しないが、Linux では別の字として扱われるので、ファイル名を指定する際などには、注意が必要である。

§2.2 数式の計算

1. 演算子と式

例題 2-2

12＋3 を計算して、その結果を画面に出力しなさい。

この解答例は以下のようになる。前と同様に実行すれば、15 が出力される。

例題 2-2 のプログラム例

```
1   program example2_02
2       implicit none
3
4       write(*,*) 12 + 3
5
6       stop
7   end program example2_02
```

項

第 4 行の「12」や「3」のことを**項** という。 項は一続きの集まりとして扱われるので、間に空白などを入れずに続けて書かなければならない。たとえば「1234567」と言う数を「1 234 567」のように書いてはいけない。*6

項にはその示す内容によりいくつかの種類がある。上の「12」や「3」は数を表す項である。一方前問の 「"Hello"」 は文字を表す項である。*7

演算子

第 4 行の「+」のことを **演算子** (*operator*) という。演算子には、前後の項の種類によって、適用できる種類が決まっている。「+」のように数の項から数値を導く演算子を、**数値演算子** という。

演算される項のことを**オペランド** (*operand*) という (「演算数」という訳語もあるが、まず使われることはない)。

演算子には 1 項だけに作用する**単項演算子** と、項と項を結びつける 二項演算子 がある。たとえば、「+5」は単項演算子であり、「12 + 5」は二項演算子である。二項演算子の前後の二項は、同じ種類でなければならない。

演算子と項の間は、「12+5」のように詰めて記述しても、「12 + 5」のように適当な空白を置いてもよいので、見やすいように書けばよい。とはいえ、単項演算子と項の間には空白を置くことは、誤りやすいのですすめられない。

式

項と項を演算子で結んだものを **式** (*expression*) という。演算子のない単一の項もやはり式である。

式もその示す内容により、項と同様にいくつかの種類がある。「12 + 3」は数を表す式であり、**数式** (*numeric experssion*) という。一方「"Hello"」は文字を表す単項の式であり、**文字式** (*character expression*) という。演算子によっては、式と項 (オペランド) の内容が異なるものもある。

式と式、式と項を演算子で結んだものも、やはり式である。

*6 FORTRAN77 までは、これが許されていたので、このような書き方をしたプログラムがまれにある。

*7 これ以外に、配列を表す項、論理の真偽を表す項などがある。

2. 数の演算

例題 2-3

数 4 と 2 に対して、(和) $4+2$、(差) $4-2$、(積) 4×2、(商) $4/2$、(べき乗) 4^2 をそれぞれ計算して、結果を画面に表示しなさい。

Fortran で使える数値演算子は、以下の 5 種類である。

(i) 「+」··· 二項演算子としての加算。および単項演算子としてのプラス。[*8]

(ii) 「-」··· 二項演算子としての減算。および単項演算子としてのマイナス (符号逆転)。

(iii) 「*」··· 二項演算子としての乗算。「×」がキーボードにないためにこれを使う。

(iv) 「/」··· 二項演算子としての除算。「÷」がキーボードにないためにこれを使う。[*9]

(v) 「**」··· 二項演算子としてのベキ乗。

これらの演算子を使えば、以下のようなプログラムになる。

例題 2-3 のプログラム例

```
1   program example2_03
2       implicit none
3
4       write(*,*) 4 + 2
5       write(*,*) 4 - 2
6       write(*,*) 4*2
7       write(*,*) 4/2
8       write(*,*) 4**2
9
10      stop
11  end program example2_03
```

3. 演算子の優先順位

例題 2-4

次の式の値を計算して、画面に出力しなさい。

(1) $4+2\times3$ (2) $\dfrac{24}{2\times4}$ (3) $-1-2$ (4) -3×-5 (5) 4×2^3 (6) 2^{3^2}

(1) 加減算と乗除算では、通常の数式と同様に Fortran でも乗除算が先に計算される。したがってこれは通常の数式と同様に、「4 + 2*3 」と書けばよい。

なお、このように加減算の演算子の前後に空白を置き、乗除算の前後を詰めるのは、計算順序を (コンパイラでなく) プログラムを読む者に明示するためである。これを、「4 + 2 * 3 」または「4+2*3 」と書いたのでは、読む者が一目でわかりにくい。まして「4+2 * 3 」と書いたのでは、誤解を与えるおそれがある。このような表現法は、長い式を書くときには重要になってくる。

(2) 「24/2*4 」は当然誤りである。「24/2/4 」とするか、または**カッコの中の式は他の項よりも先に計算される**ことを利用して「24/(2*4) 」とするのが正しい。

なおこの二形式のうちでは、計算機は一般に加減乗除算の中では**除算に一番時間がかかる**という理由

[*8] 単項演算子のマイナスと対比させる以外の意味はない。

[*9] もともと欧米ではあまり「÷」を使わないようだが。

で、除算の回数の少ない後者の形式が好まれる。

(3) 減法演算子の「-」と、単項演算子の「-」では、単項演算子の方が優先順位が高いので、通常の数式通り、「-1 - 2」と書けば、「-1 から 2 を引く」という意味になり、「1 から 2 を引いたものをマイナスにする」という意味にはならない。

(4) 乗法演算子の「*」と、単項演算子の「-」では、乗法演算子の方が優先順位は高い。そこで式の記述通り「-3*-5」とすると、「*-」という部分を定義にない演算子とみなしてエラーになってしまう。*10

　これを避けるにはカッコを使って「*(-5)」とすればよい。さらに、前の -3 にもカッコをつければ明解である。

(5) ベキ乗は乗算よりも優先されるので、表式通り「4*2**3」と書いてもよい。しかしここは、マギレを避けるためにカッコを入れて、「4*(2**3)」とすべきだろう。

(6) ベキ乗は右から計算されるので、「2**3**2」と書いた場合、$(2^3)^2 = 8^2 = 64$ ではなく、$2^{(3^2)} = 2^9 = 512$ となる。しかしこれもカッコを入れて「2**(3**2)」または「(2**3)**2」として、何を計算したいのかを明白にした方がよいだろう。

演算子の優先順位の一覧表は、Appendix の表 1 にある。

例題 2-4　のプログラム例

```
1   program example2_04
2       implicit none
3
4       write(*,*) 4 + 2*3
5       write(*,*) 24/(2*4)
6       write(*,*) -1 - 2
7       write(*,*) (-3)*(-5)
8       write(*,*) 4*(2**3)
9       write(*,*) 2**(3**2)
10
11      stop
12  end program example2_04
```

演 習 問 題

演習問題 2

　以下の数式を、Fortran プログラムに書き下して、計算しなさい。

(1) $(1+3) \times (2-4)$　　(2) $\dfrac{6}{1+2}$　　(3) $2^3 \times 3^2$

解答 (1)　(1+3)*(2-4)

(2)　6/(1+2)

(3)　(2**3)*(3**2)

*10 これをエラーにせず、たぶん × (-5) の誤りだろうと解釈して計算を続行するコンパイラがあったが、まさに有り難迷惑というものであり、素直にエラーにしてくれた方がはるかにましである。役人とコンパイラは融通を利かせるべきではない。

§ 2.3 型

1. 整数型と実数型

例題 2-5

自然数の「割り算」には二つの答がある。一つは商と余りを求めるものであり (整除法)、もう一つは非整数の実数と同様に、分数または小数を求めるものである。今、7 を 3 で割るとき、以下の答をプログラムで求めて、出力しなさい、

(1) 商を求めなさい。

(2) 余りを求めなさい。

(3) 小数で求めなさい。

Fortran の数値型には、二つの表現型がある。一つは $1, 2, 3, \cdots$ あるいは $0, -1, -2, \cdots$ のような整数のみを表す表現型であり、もうひとつはすべての実数を (2 進または 16 進の) 有効数字で表す表現型である。前者を**整数型** (*integer data type*) といい、後者を**実数型** (*real data type*) という。[*11]

<u>整数型定数</u>

整数型は計算誤差が生じないので、個数や順序を表すために使われる。その最大値、最小値は、32 ビット・マシンの場合、デフォルトで 10 億程度である。[*12] この範囲を越すとオーバーフローを引き起こす。整数を整数型として扱うためには、以下のように小数点を入れずに書く。

```
12    0    -45    +5
```

<u>実数型定数</u>

実数型はそれ以外の一般的な計算に使われる。32 ビット・マシンのデフォルトでは、その有効ケタ数は 10 進換算で 6～7 ケタであり、表せる数値の範囲は -10^{30} から 10^{30} の程度である。この範囲を越すとオーバーフローまたはアンダーフローを引き起こす。ゼロで割ったときは、まだ別のエラーとなる。

実数型の定数を表記する方法は次のいずれかである。

○ 固定小数点表現

「符号」「小数点付きの数」の形式で表す。符号の「+」は略してもよい。 また、整数部あるいは小数部がゼロの場合略してもよい（「0.0」の場合に両方略すことはできない)。

```
11.3    -5.2    4.    -.25    0.
```

○ 指数表現

「符号」「整定数または (固定小数点表現) 実定数」「e」「符号付き整数」の形式で表す。符号の「+」は略してもよい。

```
1.2e3    -.2e-2    1e0    -2e-3
```

e 以下の整数は 10 のその数のベキ乗を意味し、e の前の数にそのベキ乗数をかけたものを表す。 たとえば「1e-4」は $1 \times 10^{-4} = 0.0001$ を表し、「10e3」は $10 \times 10^3 = 10000$ を表す。[*13]

<u>整除算と除算</u>

除算の場合、除数と被除数が両方とも整数型ならば整除算が行われ、演算結果はその商となる。どちらかが実数型ならば、通常の除算が行われる。余りを求めるには、商に除数をかけて、それを被除数から引けばよい。

[*11] 整数はどちらの表現型でも表せるが、整数型で表すときと実数型で表すときでは、内部のビット配置が異なる。

[*12] 正確には、$-2^{31} \leq$ (整数型の数) $\leq 2^{31} - 1$。

[*13] e の前の数が 1 の場合、これを略して e3 などと書くと、後述の変数名とみなされてしまうので注意が必要。

例題 2-5 のプログラム例

```
1  program example2_05
2      implicit none
3
4      write(*,*) 7/3
5      write(*,*) 7 - 7/3*3
6      write(*,*) 7.0/3.0
7
8      stop
9  end program example2_05
```

なお、整除算は通常除数も被除数も正でなければならないが、Fortran の整数型の除算は負数でも行われる。この場合、符号は通常の除算と同様に決まり、商は双方の絶対値の整除算となる。すなわち、

`(-7)/(-3)` → 2、`7/(-3)` → −2、`(-7)/3` → −2

となる。商 × 除数はいずれも −6 であり、余りは −1 となる。[*14]

2. 混合演算

例題 2-6

1.5 と $\frac{4}{3}$ の積を計算するために、次のような Fortran の数式を作成した。

(1) `1.5*4/3`

(2) `4/3*1.5`

これらの計算を実行した時の値はそれぞれいくつか。

また、このようなマギレを避けるためには、どのように数式を書けばよいか。

四則計算 (ベキ乗をのぞく加減乗除算) の演算規則に、以下の規則が加わる。

○ 二項演算子のオペランドの型が異なる場合、結果はより範囲の広い数を表す方の型となる。たとえば整数型と実数型の演算の場合ならば、結果はより範囲の広い数を表せる、実数型になる。

○ 優先順位の等しい演算子が複数ある場合、**原則的に**左から計算される。[*15]

「原則的に」という意味は、ほとんどの処理系で、デフォルトでの処理がそうなっている、という意味であり、Fortran の規程でそのように定められているわけではない。特に最適化オプションを指定したときに、この原則からはずれる可能性がわずかながらある。これはきわめて発見しにくいバグとなりうるので、できるだけ避けるようにプログラムを組まなければならない。

したがって例題の解答は、「ほとんどすべての場合」という注釈がつくが、

(1) まず左から、「`1.5*4`」が評価される。これは混合演算なので、結果は実数型の `6.0` となる。次に「`6.0/3`」が評価されるが、これも混合演算なので、結果は実数型の `2.0` となる。

(2) まず左から、「`4/3`」が評価される。これは整除算なので、結果は整数型の `1` となる。次に「`1*1.5`」が評価されるが、これは混合演算なので、結果は実数型の `1.5` となる。

これらのマギレを避ける方法は、たとえ整数値でも実数型で表して計算することである (例題の式ならば「`1.5*4.0/3.0`」または「`4.0/3.0*1.5`」)。 ただし、ベキ乗の乗数は例外であり、整数型の方がよい。

[*14] しかし、できればこのような紛らわしい計算は行わないに越したことはない。

[*15] すでに示したように、ベキ乗にはこれはあてはまらない。

3. 文字型

例題 2-7

整数型の数値 5 と、文字の 5 を、それぞれ画面に出力しなさい。

ここまでに使われた型のうちで、整数型数でも実数型数でもないものは、最初に現れた"Hello"である。これは数値でなく文字の集合であり、このような型を **文字型** (*character data type*) という。

数値型の「5」と、文字型の「5」は、表記は同じでも内部表現は異なる。write 文での扱われ方も異なり、出力形式も変わる。

例題 2-7 のプログラム例

```
1  program example2_07
2      implicit none
3
4       write(*,*) 5
5       write(*,*) "5"
6
7      stop
8  end program example2_07
```

文字型の要素を、**文字列** (*character string*) という。1 文字のみからなる文字列を、とくに「文字」ということもある。

文字列と文字列を演算子で結合したものを、**文字式** (*character expression*) という。数値の場合と同様に、単一の文字列も文字式である。

文字型定数

文字型の定数は「"」と「"」、または「'」と「'」でくくられた部分に記述する。[*16] ここには、注釈と同様にその他の文字、たとえば全角文字 (漢字) や全角記号の記述が可能である。[*17]

文字型に対する演算子

文字型の演算子は、連結演算子「//」のみである。これは 2 個の文字式を連結して、一続きの文字列とする。たとえば、 "ab"//'cde' の演算結果は、"abcde"という 1 個の文字列となる。

演 習 問 題

演習問題 3

次の式を Fortran で書いて、計算しなさい。

(1) $\dfrac{4\times5}{3\times2}$　　(2) $27^{\frac{1}{3}}$

解答 (1) (4.0*5.0)/(3.0*2.0) → 3.3333 ⋯

(2) 27.0**(1.0/3.0) → 3.0

*16 今のところ、Ruby のように「"」と「'」の区別はされておらず、同等である。ただし将来、区別がつけられる可能性はある。

*17 ただし、やはりその扱いは処理系依存である。しかも、こちらは注釈と異なり、プログラムの実行に影響を与えるので、それらの文字の扱いには、それ相応の処理が必要である。初心者は、それらの文字をここで安易に扱うべきではない。

§2.4 変数

1. 変数と宣言文

例題 2-8

変数 x に数値を与え、x、x^2、x^3、x^4 を計算して出力するプログラムを作成しなさい

たとえば $x = 2$ とすれば、求める式はそれぞれ「2.0」、「2.0**2」、「2.0**3」、「2.0**4」 となるが、x の値を変えたときに、これらの式の数値をいちいち書き直すのは大変メンドーであり、長いプログラムでは事実上不可能ですらある。

このような場合、プログラム言語では **変数** (*variable*) を使う。変数とは、記憶領域の特定の場所を確保して、名前 (変数名) を対応させ、その名前で値を格納したり引き出したりするための仕組みである。

Fortran での典型的な変数の使い方は、以下の手順による。

(i) 変数名と型を指定し、型宣言文で記憶領域を確保する。

(ii) 宣言された変数名を使い、代入文その他で記憶領域に値を格納する。

(iii) 以後はその変数名を指定すれば、格納された値が引き出される。

この手順に従った、例題のプログラムは次のようになる。

例題 2-8 のプログラム例

```
1   program example2_08
2
3       implicit none
4       real :: x         ! 型宣言文
5
6       x = 2.0           ! 代入文
7       write(*,*) x
8       write(*,*) x**2
9       write(*,*) x**3
10      write(*,*) x**4
11
12      stop
13  end program example2_08
```

x = 2.0 の右辺の値を変更すれば、出力値もそれに対応して変わる。

型宣言文

第 3 行が、**型宣言文** (*type declaration statement*) である。型宣言文は**宣言文** (*declaration statement*) の一種であり、この文の意味は、「x という変数名で、実数型の変数 1 個を記憶領域に確保する」である。*18 型宣言文の形式は、中央の二重コロン「::」の左側に **型指定子** (*type specifier*) 、右側に変数名である。

型指定子 :: 変数名

型指定子は型ごとに決まっている。これまでに出てきた型については、

(整数型)→ integer

(実数型)→ real

(文字型)→ character(len=*nn*)

*18 具体的にメモリ上のどの番地に確保されたかは、処理系が管理してくれるので、通常ユーザは気にする必要はない。

である。文字型は他と異なり、格納する文字列の最大長 *nn* (つまり格納領域の大きさ) を、あらかじめ宣言しておかなければならない。なお、「`len=`」は省略でき、`character(12)` などと書いてもよい。[*19]

変数名は Fortran の命名規則 (英字で始まる 31 文字以内の英数字または「_」) という規則に従う。一つのプログラム内で[*20]、同じ名前の変数を重複して使うことはできない。

同じ型の変数を複数宣言するときには、二重コロンの右側にコンマで区切って並べてよい。すなわち,

```
real :: x
real :: y
```

という 2 行の代わりに、以下のように書いてもよい。

```
real :: x, y
```

式の評価

宣言文では変数の記憶領域を確保しただけで値は格納されていないので、もし変数 x の値が与えられないまま第 7 行以下の write 文を実行すれば、実行時のエラーとなる。[*21] この種のエラーを避けるためには、式が評価される前に、以下に述べる代入文または初期化式によって、式の評価が行われる前に、変数に値を与えておかなければならない。

2. 代入文

第 5 行が、`x` と名付けられた記憶領域に、`2.0` という値を格納する文であり、**代入文** (*assignment statement*) という。一般的な代入文は、次の形式である。

変数名 = 式

代入文の働きは**「右辺の式を計算してその結果の値を左辺の変数に代入する」**ことである。等号「`=`」を使っていても、等式、すなわち左辺と右辺が等しいことを意味するものではない。たとえば「`x+1=y`」のような表現は、コンパイル・エラーとなる。その意味では、

変数名 <- 式

とでも表記した方が、むしろ実態に即している。[*22]

3. 初期化式

変数は宣言文をコンパイルと同時に、値を与えておくことができる。たとえば、例題 2-8 の第 4 行と第 6 行と合わせて、次の文で記述できる。

```
real :: x = 2.0
```

このとき宣言文の変数の右辺に現れる式を **初期化式** (*initialization expression*) といい、初期化式の値を初期値という。初期化式は、一般の式と異なり記述できる内容に制限があり、とりあえず定数、およびその四則計算のみが許されると考えてよい。一般的な形式は次のようである。

型指定子 :: 変数名 = 初期化式, 変数名 = 初期化式...

名前付き定数

変数に代入された初期値は、プログラムの実行中に代入文等で、その値を変更することができる。しかし、重力加速度 g や光速 c などの物理定数、および π などの数学定数で、誤って変更されてほしくないものに対しては、以下の例のように宣言を行えば、変更されることを避けることができる。

[*19] というより、ほとんどの場合省略される。
[*20] 正確には一つの手続き内で (後述)
[*21] FORTRAN77 では、多くの場合ゼロが格納されていて、エラーにならない。これもまた、原因究明が困難なバグとなりうる。
[*22] 実際、ALGOL や PASCAL では、代入文には「`=`」の代わりに、「`:=`」という表記を使う。この方が厳密で誤りもないだろうが、いったん覚えさえすればそれですむことなので、他の言語では「`=`」を使い続けている。

```
real, parameter :: pai = 3.14159265
```

ここで「parameter」は「pai」に対して、parameter という**属性** (*attibute*) を指定している。parameter 属性を持つものを **名前付き定数** (*named constant*) といい、必ず初期化式を伴っていなければならない。その一般形は以下の形式である。

```
型指定子, parameter :: 名前付き定数名 = 初期化式, 名前付き定数名 = = 初期化式...
```

名前付き定数は、変数ではなく定数である。すなわち、変数のように記憶領域が確保されるのではなく、上の例で言えばコンパイル時に「pai」という名前付き定数が、すべて「3.14159265」という定数に置き換わってしまうと考えてよい。したがって、実行時にその値を変更することはできなくなる。

名前付き定数の初期化式での使用

名前付き定数は「定数」であるから、他の初期化式で使うことができる。ただしその場合は、使われる名前付き定数が必ずその前で宣言されていなければならない。

たとえば、型宣言文の順序が、次の順に宣言されていれば有効である。

```
integer, parameter :: n = 10
integer, parameter :: n2 = n * 2
```

順序がこの逆であれば、エラーとなる。

4. 変数を含む数式

例題 2-9

(1) 上辺 a、底辺 b、高さ h の台形の面積を求める Fortran の数式を書きなさい。

(2) $a=1$、$b=2$、$h=1.5$ として、面積を計算して出力しなさい。

変数を含む数式の書き方も、定数だけの時と同様であり、以下のことに注意する。

○ 通常の計算は、原則的にすべて実数型で行う。

○ 乗算記号「*」を省略せずに記述する。

後者について補足すると、$x \times y$ を xy と書くと、コンパイラは単一の変数 xy と区別できなくなる。

例題 2-9 のプログラム例

```
1  program example2_09
2      implicit none
3      real :: a = 1.0, b = 2.0, h = 1.5 ! 初期化式付き型宣言文
4
5      write(*,*) (a+b)*h/2.0
6
7      stop
8  end program example2_09
```

5. 代入文による型変換

例題 2-10

次のような代入文で、変数にはどのような値が格納されるか。プログラムを作成して確かめなさい。

(1) 実数型の変数 x に、整数型定数 1234567890 を代入する。

(2) 整数型の変数 k に、実数型定数 2.8 を代入する。

代入文の左辺の変数と右辺の式は、同じ性質 (数値、文字 ...) でなければならないが、型は異なってもよい。つまり数値型であれば、整数型でも実数型でもよい。

(1) 実数型変数に整数式を代入する場合、有効数字表現に変換してから代入される。したがって、実数型の精度のケタ (6 ～ 7 ケタ) 以内の整数であれば、誤差なく変換される。この場合そのケタ数を超えているので、最後のケタを四捨五入した 1.234568×10^9 が変数に代入され、それ以上のケタは捨てられる。

(2) 整数型変数に実数式を代入する場合、整数型が表現できる範囲の数値であれば、小数部分を切り捨てた値が変数に代入される。これは負の数でも同様である。

この範囲を超えた数値を代入するとオーバーフローが起きるが、これはデフォルトではエラーにならず、予期しない数が代入されることになる。これも発見困難なバグになりうるので、注意が必要である。

例題 2-10 のプログラム例

```
1   program example2_10
2       implicit none
3       real :: x
4       integer :: k
5
6       x = 1234567890 ! 整数値を実数型変数に代入
7       write(*,*) x
8       k = 2.8
9       write(*,*) k   ! 実数値を整数型変数に代入
10
11      stop
12  end program example2_10
```

演 習 問 題

演習問題 4

次の式を Fortran で書きなさい。

(1) $x(y+z)$　(2) $a+\frac{b}{2}$　(3) $\frac{1}{t+1}$

解答 (1) `x*(y+z)`　(2) `a + b/2.0`　(3) `1.0/(t+1.0)`

§2.5 標準入出力

1. read 文と write 文

例題 2-11

キーボードから 4 個の実数型変数 x_1、x_2、x_3、x_4 を読み込み、それらの値を 1 行にまとめて出力し、次の行にその合計を出力するプログラムを作成しなさい。

変数の値をキーボードから入力するには、read 文を使う。

```
read(*,*) 変数 1, 変数 2, 変数 3...
```

画面に値を出力するのは、これまで行ってきたように write 文である。

```
write(*,*) 式 1, 式 2, 式 3 ...
```

例題 2-11　のプログラム例

```
1   program example2_11
2       implicit none
3       real :: x1, x2, x3, x4
4
5       read(*,*)  x1, x2, x3, x4 ! キーボードから入力
6       write(*,*) x1, x2, x3, x4 ! 画面に出力
7       write(*,*) "sum=", x1 + x2 + x3 + x4
8
9       stop
10  end program example2_11
```

read 文の変数、write 文の式には型の混在が可能である。たとえば、

```
write(*,*) "y=", y
```

のように、文字型と数値型を同時に出力することもできる。

標準入力と標準出力

例題の read 文と write 文の「`(*,*)`」のうち、「`,`」の前の第 1 項は、それぞれ入力装置および出力装置を指定する項である。この項が「`*`」である入出力を**標準入出力** という。

(i) `read(*,..` の「`*`」は、入力を標準入力装置から行うことを意味する。PC の場合、標準入力装置はキーボードに割り当てられている。

(ii) `write(*,..` の「`*`」は、出力を標準出力装置に行うことを意味する。PC の場合、標準出力装置はディスプレイ画面に割り当てられている。[*23]

並び入力と並び出力

一方、例題の read 文と write 文の「`,`」の後の第 2 項は、それぞれ入力および出力の様式を指定する項である。この項が「`*`」である入出力を**並び入出力** (*list-directed input/output*) という。並び入出力は、ユーザが入出力の様式を指定せず、Fortran の標準様式、および処理系によって定められた方法で入出力を行うことを意味する。

このように、「`(*,*)`」の前後の「`*`」はそれぞれ意味が違う。したがって必ずしもペアで使われる必要はなく、どちらかだけが「`*`」であるということも、当然ある。

[*23] 大型汎用機の場合は、プリンタに割り当てられているときもある。

2. 並び入力文

例題 2-11 の第 5 行の、標準入力からの並び入力文を例として、並び入力文の入力方法を説明する。

基本的な入力方法

(i) プログラムの実行中に標準入力文に遭遇すると、プログラムはキーボードからの入力待ちになる。
(ii) 入力並びにある変数の個数と、同数のデータ値を入力する。
(iii) 入力したデータ値は、並びの順番に格納される。
(iv) 対応する変数の型と、入力するデータ値の型は一致していなければならない。ただし例外として、実数型の変数に対し、整数型のデータ値を入力してもよい。これは実数に変換されて格納される。
(v) 入力方法は、キーボードから、値を区切り文字 (*separator*) で区切って入力し、最後に改行「Enter」を打つ。区切り文字としては、1 個以上の空白「␣」あるいは 1 個のコンマ「`,`」である。コンマの前後にいくつかの空白があってもよい。たとえば以下の 2 行は、いずれも有効な入力である。

```
1  2   3   4
1.0,2.0, 3.0 , 4.0
```

データ値の数の方が多い場合

(i) 入力並びにある変数の個数よりも多いデータ値を入力した場合。不必要なデータはその行の最後まで捨てられる。次にまた read 文が実行されても、捨てられた部分は読みこまれず、次の行から読み込みが始まる。これは Fortran と他のプログラム言語との違いの一つである。[*24]
(ii) 逆にこれを利用して、データにコメントを入れることもできる。たとえば、

```
5  6   8   9.2   ! ここには 4 個の実数を入れる。
```

などと文字列を追加しても、必要な `9.2` 以後は読み飛ばされるので、計算結果に影響は与えない。[*25] キーボードからの入力の際にはそれほど必要性はないが、ファイルからの入力の際には有用である。

データ値の数の方が少ない場合

(i) 1 行に入力したデータ数が、入力並びにある変数の個数よりも少ない場合 (空行の入力も含む)、プログラムは次の行で入力待ちになる。これは、データ数が変数の個数に達するまで繰り返される。最後の行で、要求される数より多いデータ値を入力した場合は、やはり不必要な部分は読み捨てられる。
(ii) この状態でプログラムを強制終了させるためには、前述したように cntl-C を押す。

データ入力を中断する場合

プログラムを中断せず、データの入力を途中で止めたい場合は、スラッシュ「`/`」を入力する。スラッシュの前の入力データ値は入力並びの対応する変数に格納されるが、それ以後の入力並びの変数には何も格納されない。

文字型の入力

(i) 入力並びの文字型変数に対する入力データは、データ中に「`,`」、「␣」、「`/`」がなく、かつ先頭の文字が「`"`」または「`'`」でない場合は、数値型と同様に入力してよい。
(ii) データ中に上記の記号を含む場合には、データ値を「`"`」と「`"`」か、「`'`」と「`'`」で囲む。囲み記号は、データ中に含まれない記号の方を選択する。
(iii) 両方の記号を含む文字列の場合、囲み記号に使われた文字を連続して入力すれば、1 つの記号として格納される。たとえば、`"ab'cd""ef"` と入力すると、「`ab'cd"ef`」という値として格納される。

[*24] おそらくこれは、Fortran がパンチ・カード時代からの言語であることに由来する。
[*25] これは注釈ではないので、「`!`」は必要ないが、コメントであることを明示するために入れている。

3. 並び出力文

数値式の出力

(i) 並び出力文の出力並びが数値式の場合、まず先頭に 1 文字分の空白が出力され、その後に数値が処理系のデフォルトの形式で出力される。
(ii) 出力並びが複数の場合、第 1 の数値式の値の出力後に、数個分の空白が出力され、その後に次の数値式の値が出力される。
(iii) 出力の長さが 1 行には長すぎると処理系が判断した場合、適当な長さで改行される。

文字式の出力

(i) 並び出力文の出力並びが文字式の場合、まず先頭に 1 文字分の空白が出力されるところまでは同じだが、その後は文字列がそのまま出力される。
(ii) 出力並びが複数の場合、第 1 の文字式の値の出力後に、次の文字式の値が 続けて 出力される。
(iii) 出力の長さが 1 行には長すぎると処理系が判断した場合、やはり適当な長さで改行される。

4. リダイレクションによる標準入出力の切り替え [*26]

標準入力のリダイレクション

データの数が多くなると、入力のタイミングにあわせていちいちキーボードから打つのは面倒である。また、万一打ち間違いをしたら、ctrl-C でプログラムを中断し、最初から打ち直さなければならない。

そのような場合、あらかじめデータをエディタ等でデータ・ファイルとして保存しておき、それを実行時に呼び出した方が便利である。そのために、プログラム実行時に、リダイレクションによりキーボードからの標準入力を、該当ファイルからの入力に切り替える。[*27]

(i) まず、データファイルをエディタで作成し、たとえば `xxx.txt` などのファイル名で保存しておく。

(ii) プログラムはこれまで通り、標準入力を使って作成する。これをたとえば `abc.f90` として保存する。

```
read(*,*) var1, var2,...
```

(iii) プログラムをコンパイル・リンクして実行ファイルを作成する。その方法については、第 1 章の「実行までの操作手順」を参照すること。
(iv) 端末モードで、ディレクトリを変更して実行準備をする。これについても上記の記述を参照すること。
(v) ここで端末モードから、

(Linux の場合)　`./a.out < xxx.txt`
(Windows の場合)　`abc.exe < xxx.txt`

と入力すれば、プログラムはキーボードの代わりに `xxx.txt` からデータを読み取って実行する。

標準出力のリダイレクション

出力の場合も、長い出力の場合はいったんファイルに出力しておいて、あとでエディタ等で見た方が便利である。

(i) この場合も、プログラムはこれまで通り、標準出力を使って作成する。これをたとえば `def.f90` として保存する。

```
write(*,*) expression1, expression2,...
```

(ii) プログラムをコンパイル・リンクして実行ファイルを作成する。

[*26] リダイレクションは Fortran の機能ではなく、OS の機能である。
[*27] ただし、リダイレクションでは、1 つのプログラムでそれぞれ 1 つずつのファイルにしか入出力が行えない、等の欠点がある。それらの点の解決には、open 文を使ってファイル入出力を行わなければならないが、それについては基礎編で扱う。

(iii) 端末モードで、ディレクトリを変更して実行準備をする。

(iv) ここで端末モードから、

(Linux の場合)　`./a.out > zzz.txt`

(Windows の場合)　`def.exe > zzz.txt`

とすれば、結果は画面ではなく、ファイル zzz.txt に書き込まれる。

ただし、プログラム中に、

```
write(*,*) "x を入力してください"
```

などという記述がある場合、「x を入力してください」の部分も、zzz.txt に書き込まれてしまうので注意を要する。

演習問題

演習問題 5

以下のプログラム、

```
program main
    implicit none
    integer :: i1, i2, i3, i4

    read(*,*) i1
    read(*,*) i2, i3
    read(*,*) i4
    write(*,*) i1, i2, i3, i4

    stop
end program main
```

により、次のデータ・ファイル

```
1, 2
3
4, 5
6, 7
```

を読み込んだとき、i1,i2,i3,i4 にはどのような数値が入るか。

解答 i1 ··· 1　i2 ··· 3　i3 ··· 4　i4 ··· 6

§2.6 組込み関数

1. 関数

一般にプログラミング言語では、数学の関数 $f(x)$ 等に相当する**関数** (*function*) という手法がある。関数は、大きく次の二種類に分類できる。

(i) 組込み関数 (*intrinsic function*)

あらかじめコンパイラで定義されて提供され、ユーザがそのまま使える関数。Fortran95 では、100 を超える組込み関数が、規程により標準で提供されている。*28

また、それ以外にベンダーがコンパイラごとに独自に提供する関数があるが、これらを利用するとプログラムの移植が非常に困難になるので、注意を要する。

(ii) ユーザ定義関数

ユーザが独自にプログラムすることにより、関数を定義することができる。定義の方法については、基礎編で扱う。また、他のユーザの作成した関数を組み込むことも可能である。

関数の基本

(i) Fortran では $f(x)$ の f を**関数名**、x にあたるものを**引数**(ひきすう)(*argument*) といい、関数値にあたるものを **戻り値** と言う。

(ii) 関数の基本形式は、次の形である。

```
関数名 (引数 1[, 引数 2][, 引数 3], ...)
```

引数の数および型は、関数によって定まっている。ただし、必ずしも一通りではない。

(iii) 戻り値の型は関数により定まっている。ただし、引数によって変わることがある。

(iv) 関数は、式の中に項として埋め込んで使う。

2. 数値関数

例題 2-12

実数型変数 x に対して、

(1) x の小数部分を切り捨てた整数値。

(2) x を四捨五入した整数値。

(3) x より小さいか等しい整数の中で、最大の整数の値。

(4) x より大きいか等しい整数の中で、最小の整数の値。

を求める Fortran 式を書きなさい。 ただし $|x| < 2^{31} - 1$ とします。

また、その式を使って、$x = -2.5$ のときの値をそれぞれ出力しなさい。

この例題のように、代入による変換ではなく明示的に型変換を行うためには、以下のような関数を使う。

○ `int(x)` ··· 戻り値は `x` の小数部分を切り捨てた値で、型は整数型。`x` は数値型ならどれでもよい。

なお、同じ働きをする関数で、戻り値が実数型のものとして、`aint(x)` がある。

○ `nint(x)` ··· 戻り値は `x` を四捨五入した値で、型は整数型。`x` は実数型。

なお、同じ働きをする関数で、戻り値が実数型のものとして、`anint(x)` がある。

○ `floor(x)` ··· 戻り値は `x` 以下の整数の中で、最大の整数値であり、型は整数型。`x` は実数型。

○ `ceiling(x)` ··· 戻り値は `x` 以上の整数の中で、最小の整数値であり、型は整数型。`x` は実数型。

---

*28 すべての組込み関数を説明することは、ここではできないので、必要に応じて参考書の [3][4][5][8] あたりを参照していただきたい。

例題 2-12 のプログラム例

```
 1  program example2_11
 2      implicit none
 3      real :: x
 4
 5      x = -2.5
 6      write(*,*) int(x)
 7      write(*,*) nint(x)
 8      write(*,*) floor(x)
 9      write(*,*) ceiling(x)
10
11      stop
12  end program example2_11
```

実数型への変換

整数を実数型に変換する関数は、次の一つだけである。

○ `real(x)` ··· 戻り値は `x` の値で、型は実数型。`x` は数値型ならどれでもよい。

数値関数

このように、数値型に関する操作を行う関数の集合のことを**数値関数** (*numeric function*) と言う。上述した 5 個の関数以外の主なものとしては、次のようなものがある。

○ `abs(x)` ··· 戻り値は `x` の絶対値で、型は引数と同じ。`x` は数値型ならどれでもよい。

○ `mod(n,m)` ··· 戻り値は `n` を `m` で割った余りで、型は引数と同じ。`n` は整数型または実数型、`m` は `n` と同じ型。

○ `max(x1,x2[,x3]...)` ··· 戻り値は `x1,x2,...` の中での最大値で、型は引数と同じ。`x1,x2,...` は整数型または実数型で 2 個以上の任意の個数。ただしすべて同じ型でなければならない。

○ `min(x1,x2[,x3]...)` ··· 戻り値は `x1,x2,...` の中での最小値で、型は引数と同じ。`x1,x2,...` は整数型または実数型で 2 個以上の任意の個数。ただしすべて同じ型でなければならない。

これ以外に、`aimag`(複素数の虚数部分の取り出し)、`cmplx`(複素数型への変換)、`conjg`(複素共役)、`dble`(倍精度化)、`dim`(超過分)、`dprod`(倍精度化乗算)、`modulo`(剰余))、`sign`(符号変更) といった数値関数がある。

3. 数学関数

例題 2-13

実数型変数 x、y に対して、

(1) $\sqrt{x^2+1}$

(2) e^{x-y}

(3) $\log(x+y)$

(4) $\sin x$

(5) $\arctan x$

を求める Fortran 式を書きなさい。

また、その式を使って、$x=1.0$、$y=0.5$ のときの値をそれぞれ出力しなさい。

数学関数

Fortan では、いわゆる数学でいう関数のいくつかが標準で用意されている。これらを **数学関数** (*mathematical function*) という。数学関数の引数はすべて実数型で、整数型は許されない。*29 戻り値の型は、引数と同じである。

数学関数は重要なので、すべてを以下にリストアップする。

○ `sqrt(x)` ··· 戻り値は x の平方根 $\sqrt{x}$。$x < 0$ の場合は、実行時エラーとなる。

○ `exp(x)` ··· 戻り値は指数関数 e^x の値。

○ `log(x)` ··· 戻り値は e を底とする自然対数 $\log_e x$ の値。$x < 0$ の場合は、実行時エラーとなる。

○ `log10(x)` ··· 戻り値は 10 を底とする常用対数 $\log_{10} x$ の値。$x < 0$ の場合は、実行時エラーとなる。

○ `sin(x)` ··· 戻り値は正弦関数 $\sin x$ の値。x の単位はラジアン。

○ `cos(x)` ··· 戻り値は余弦関数 $\cos x$ の値。x の単位はラジアン。

○ `tan(x)` ··· 戻り値は正接関数 $\tan x$ の値。x の単位はラジアン。

○ `asin(x)` ··· 戻り値は逆正弦関数 $\arcsin x$ の値。$|x| \leqq 1$ でないとエラーとなる。戻り値の単位はラジアンであり、主値は $-\pi/2 \leqq$ 戻り値 $\leqq \pi/2$。

○ `acos(x)` ··· 戻り値は逆余弦関数 $\arccos x$ の値。$|x| \leqq 1$ でないとエラーとなる。戻り値の単位はラジアンであり、主値は $0 \leqq$ 戻り値 $\leqq \pi$。

○ `atan(x)` ··· 戻り値は逆正接関数 $\arctan x$ の値。戻り値の単位はラジアンであり、主値は $-\pi/2 <$ 戻り値 $< \pi/2$。

○ `atan2(y,x)` ··· 戻り値は原点から座標 (x,y) に引いた線分と x 軸のなす角を、x 軸から左回りに測った角度。戻り値の単位はラジアンであり、主値は $-\pi <$ 戻り値 $\leqq \pi$。`x` と `y` とは同じ型でなければならない。また、両方とも 0 であってはならない。

○ `sinh(x)` ··· 戻り値は双曲線正弦関数 $\sinh x = \dfrac{e^x - e^{-x}}{2}$ の値。

○ `cosh(x)` ··· 戻り値は双曲線余弦関数 $\cosh x = \dfrac{e^x + e^{-x}}{2}$ の値。

○ `tanh(x)` ··· 戻り値は双曲線正接関数 $\tanh x = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$ の値。

例題 2-13 のプログラム例

```
1   program example2_12
2       implicit none
3       real :: x, y
4
5       x = 1.0
6       y = 0.5
7       write(*,*) sqrt(x*x + 1.0)
8       write(*,*) exp(x-y)
9       write(*,*) log(x+y)
10      write(*,*) sin(x)
11      write(*,*) atan(x)
12
```

*29 `sqrt`、`exp`、`log`、`sin`、`cos` は、複素数型の引数が可能である。

```
13        stop
14    end program example2_12
```

4. 組込み関数についての補足

組込み関数の機能別分類

組込み関数の機能別分類は、前述した数値関数、数学関数の他に以下のような種類がある

∘ 配列関数 ··· 配列を引数、あるいは戻り値とする関数。

∘ 文字列操作関数 ··· 文字または文字列を引数、あるいは戻り値とする関数。

∘ ビット処理関数 ··· 整数型のビッドのオン・オフを操作する関数。

これ以外にも他種類の関数があるが、それらの分類方法は解説書により異なる。

組込み関数を利用する場合の注意事項

(i) 組込み関数の引数は、一般に項ではなく式 (実数式) であり、式中に他の組込み関数があってもよい。

(例) → `sqrt(abs(-y))`

(ii) 組込み関数の関数名と、ユーザの定義した名前がたまたま一致した場合、ユーザの定義名の方が優先される。したがって、すべての組込み関数名を記憶しておく必要はない。

(例) →　　`real :: sin`

とはいえ、紛らわしい上に、その組込み関数は当該のプログラムで使えなくなるので、可能な限り避けるべきである。

(iii) 三角関数 `sin`、`cos`、`tan` の引数がラジアン単位であることに留意すること。計算を正確にするためには、π に十分な精度を与えておく必要がある。

演 習 問 題

演習問題 6

次の式の値を Fortran で求めなさい。

(1) $a=-2$ のとき、$\log_e\left|\dfrac{a+1}{a-1}\right|$

(2) $t=\dfrac{\pi}{4}$ のとき、$\sin t+\cos 2t$

(3) 二次元座標点 $(x,y)=(-\dfrac{1}{2},\dfrac{\sqrt{3}}{2})$ の原点からの距離および偏角

解答 (1) `log(abs((a+1.0)/(a-1.0)))=-1.09861`

(2) `sin(t) + cos(2.0*t)=0.707107`

(3) `sqrt(x*x+y*y)=1.00000`、`atan2(y,x)=2.09440`

2 章 の 章 末 問 題

【問題 2-1】
次の式を Fortran の数式に書き直し、$x = 1$ のときの値を求めなさい。

(1) $\sqrt{x^3+2x^2+2x+1}$　　(2) $\dfrac{1+\cos 2x}{2}$　　(3) $\dfrac{1}{\sqrt{2\pi}}e^{-x^2/2}$

【問題 2-2】
　球の半径 r を変数として読み込み、その体積 $\frac{3}{4}\pi r^3$、および表面積 $4\pi r^2$ を出力するプログラムを作成しなさい。

【問題 2-3】
　実数値 x を入力し、それを小数第 2 位で四捨五入した値を出力するプログラムを作成しなさい。

【問題 2-4】
(1) 時刻の、時 (h)、分 (m)、秒 (s) をキーボードから読み込んで、零時からの経過秒数 (t) を出力するプログラムを作成しなさい。ただし、これらの値は $0 \leqq h < 24$、$0 \leqq m < 60$、$0 \leqq s < 60$ である整数とします。
(2) 逆に零時からの経過秒数 (t) を読み込んで、時、分、秒を出力するプログラムを作成しなさい。ただし経過時刻は、$0 \leqq t < 86400$ である整数とします。

【問題 2-5】
　1000 円未満の金額を入力し、それに消費税分 5% を加えた合計の金額を出力するプログラムを作成しなさい。ただし、合計金額の 1 円未満は切り捨てるものとします。

【問題 2-6】
　まず、平面上の三点 $A(1.0,1.0)$、$B(5.0,3.0)$、$C(4.0,5.0)$ の座標値を、1 行に一点ずつ (x,y) を組にして、

```
1.0,1.0
5.0,3.0
4.0,5.0
```

という形式でファイルに保存しなさい。
(1) このファイルを読み込んで、三角形の対辺 a,b,c の長さを計算して出力するプログラムを作成しなさい。ただし、それぞれの辺の長さは、「a=」、「b=」、「c=」の後に出力するものとします。

(2) これらの辺の長さに加えて、ヘロンの公式、

$$S = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{ただし } s = \frac{a+b+c}{2}$$

により、三角形 ABC の面積 S を計算するプログラムを作成しなさい。

(ヒント) まず計算しやすい座標値、たとえば (0.0,0.0)、(3.0,0.0)、(0.0,4.0) などでプログラムが正しく作成されたかどうかをテストした後、上記の座標値での計算を行う。

■　第 II 部　基礎編　■

第 3 章　制御構文

§ 3.1 制御構造

1. 実行の流れ

ここまでのプログラムの計算方式は、基本実行順序あるいは逐次実行順序といい、電卓を叩くようにすべて 1 行ずつ順に実行し、stop 文に達して終了する、という方式であった。このような計算方式は、基本的ではあるが、プログラミングの利点をフルに利用しているとはいえない。たとえば、計算の途中の値により処理方式を変える、あるいは同じ計算式をデータを替えて何度も計算する、といった、電卓ならば人間が判断することを、プログラムにより計算機に自動的に行わせることにより、高速な計算が可能になるのである。

このようなプログラムの計算順序のことを、**制御構造** (*control structure*) または**制御フロー** (*control flow*) といい、プログラム言語で制御構造を指定する文のことを **制御文** (*control statement*) という。また、単独の文ではなく、複数の文を合わせて制御構造を指定するときの文の集合を **制御構文** (*control constructs*) という。[*1]

とりあえず簡単な制御文の例を一例示しておく。このプログラムは読み込んだ値の絶対値を出力する。

制御文の例

```
1   program example_control
2       implicit none
3       real    :: x
4
5       read(*,*) x
6       if ( x < 0 ) x = -x ! 負ならば符号を反転
7       write(*,*) x
8
9       stop
10  end program example_control
```

第 6 行が制御文の一つである if 文である。この文は、もし読み込んだ x の値が負ならば、`x = -x` の部分を実行することを指示している。x の値が正またはゼロならば、その部分は実行されない。これにより、読み込んだ x の値の符号にかかわらず、x にその絶対値が格納され、次の write 文で出力される。

制御構文は、コンパイラ言語ごとにその種類と構文表現が大きく異なる。また、同じ Fortran でも、FORTRAN66、FORTRAN77、Fortran90/95 はかなり異なる。多くの場合上位互換であり、FORTRAN66、FORTRAN77 の制御構文は Fortran90/95 でも使えるが、中には廃止あるいは廃止予定のものもあり、[*2]そのような制御構文は、たとえ使用しているコンパイラが実装していたとしても、使うべきではない。

[*1] 本冊子では、制御文と制御構文を合わせて、制御構文と呼ぶこともある。
[*2] 割当て型 goto 文、pause 文、算術 if 文、entry 文、return 文の選択戻りなど、特によく使われているのは算術 if 文。

2. 構造化

制御構文を積極的に取り入れてプログラミングを行うことを、**構造化**という。わざわざこのような言葉を使う理由は、構造化を行わなくてもプログラムを書くことは可能だからである。それは、プログラムはコンパイラにより最終的には機械語に変換されるが、機械語はごく単純な制御構造しか持っていない、という理由による。

すなわち構造化は計算機のために行うというよりは、人間、すなわちユーザが容易にプログラムを理解できるように行うものである。初期の計算機は低速の上に高価であり、可能な限り速く計算を行う必要があったが、その後の高速化、低価格化により、現在ではプログラム開発に要する時間の方が重要となっている。プログラムが容易に理解できれば、それだけ全体としての生産性の向上につながり、開発コストも低下することになる。

しかしながら、構造化を行うためには、プログラミング言語自体が構造化に対応していなければならない。ダイクストラ*3はこの観点から早くから構造化の必要性を唱えた。FORTRAN66 から FORTRAN77 への改定は、これに対応して大幅に構造化を取り入れたものとなっている。また FORTRAN77 から Fortran90 への改定も、いくらかの追加があった。

これらの改訂により、Fortran の制御構文は、歴史の長い言語でありながら他の新しいプログラミング言語に比べても、それほど劣らないものになっている。ユーザはこれを活用して、できるだけ理解しやすいプログラムを書くことにつとめるべきである。

3. 制御構造の種類

一般的に、プログラミング言語の制御構造は、以下のように分類できる。*4

(i) 順接 (sequence) ··· 何も制御を行わず、次の文を実行する。
(ii) 条件判断 (あるいは分岐、選択) ··· 何らかの条件の成立、不成立によって実行する部分を変える。
(iii) 反復 (ループ) ··· 特定の文の並びを、繰り返し実行する。
(iv) ジャンプ (ブランチ) ··· 指定した文に制御を移す。
(v) サブプログラム ··· (制御文とは独立した場所にある) 指定した文の並びを実行した後、次の文に制御を戻す。
(vi) 停止 ··· プログラムを停止させる。

Fortran90/95 の制御文

上記の分類に対応する Fortran90/95 の制御文は次のようなものである。*5

(i) 順接 ··· continue 文
(ii) 条件判断 ··· if 文、if 構文、case 構文
(iii) 反復 (ループ) ··· do 構文
(iv) ジャンプ ··· goto 文、exit 文、cycle 文
(v) サブプログラム ··· call 文、return 文
(vi) 停止 ··· stop 文

(i) の continue 文は、Fortran90/95 では、主に goto 文と組み合わせで使われる。

また、(v) のサブプログラムについては、Fortran の解説書では制御文の範疇に含めないことが多い。本冊子もそれにしたがい、別に章を立てて解説することにする。

---

*3 エドガー・ダイクストラ (Edsger Wybe Dijkstra)「Structured Programming」(1967)
*4 この分類にはいろいろな議論があり、一般的なものはない。以下はこの冊子を書くにあたって分類した私見。
*5 これ以外にも廃止あるいは廃止予定、あるいは使う必要が文がいくつかあるが、「はじめに」で述べた方針により記述しない。

4. 流れ図

フローチャートと PAD 図

一般に仕事を行うとき、作業の手順を図にすれば考えやすいし、他人にも説明しやすい。このような図化には多くの方式があるが、**アルゴリズム** (*algorism*)、すなわちプログラムの計算手順の図示に使われるのは、**フローチャート** (*flowchart*) と **PAD 図** の二種類である。これらの特徴を比較すると以下のようになる。

○ フローチャート

· プログラミングに限らず、一般的な作業全般に使える。

· その反面、アルゴリズムの記述には必ずしも適していない。同じ制御構造を表すのに、書く人によって図がかなり変わる可能性がある。

· 記号の書き方が JIS により定められていて、書く人により違いがない。

· 一般によく知られている。

○ PAD 図

· アルゴリズムの記述を目的とした図である。

· 特に、反復構文の記述に関して、フローチャートに対して明らかな優位性を持つ。

· 記号の書き方に、人によりいくらかの方言がある。

· プログラマ以外はまず知らない。

この冊子では、Fortran のユーザの多くが情報処理の専門家でないこと、およびデータ処理のアルゴリズムはそれほど複雑でないことを考慮して、応用範囲の広いフローチャートの方を使うことにする。

フローチャート記号

JIS で定められたフローチャート記号のうち、主要なものを以下に示す。

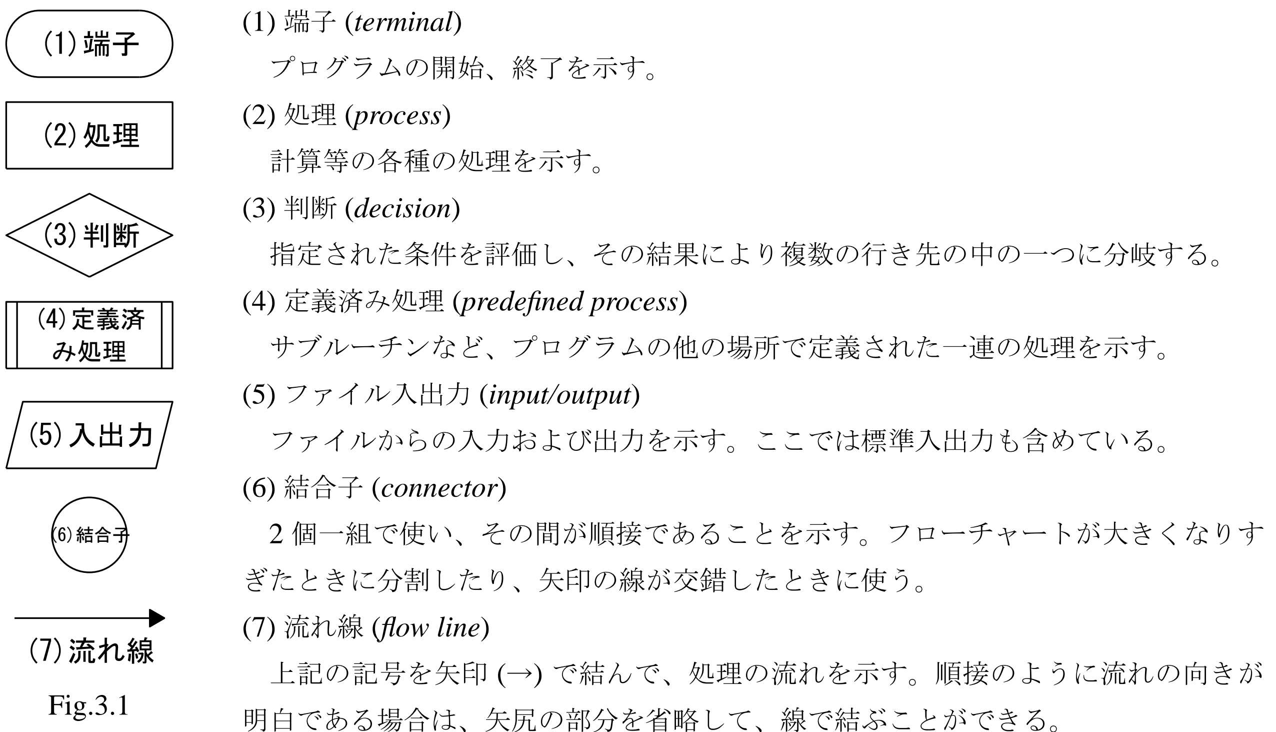


Fig.3.1

(1) 端子 (*terminal*)
プログラムの開始、終了を示す。

(2) 処理 (*process*)
計算等の各種の処理を示す。

(3) 判断 (*decision*)
指定された条件を評価し、その結果により複数の行き先の中の一つに分岐する。

(4) 定義済み処理 (*predefined process*)
サブルーチンなど、プログラムの他の場所で定義された一連の処理を示す。

(5) ファイル入出力 (*input/output*)
ファイルからの入力および出力を示す。ここでは標準入出力も含めている。

(6) 結合子 (*connector*)
2 個一組で使い、その間が順接であることを示す。フローチャートが大きくなりすぎたときに分割したり、矢印の線が交錯したときに使う。

(7) 流れ線 (*flow line*)
上記の記号を矢印 (→) で結んで、処理の流れを示す。順接のように流れの向きが明白である場合は、矢尻の部分を省略して、線で結ぶことができる。

· フローチャートは、順接の場合に上から下へと処理が行われるように書く。
· 各記号への入り口点は、原則的に上方からの 1 カ所のみとする。
· 流れ線を交差させてはいけない。結合子を使うか、交差でないことを明示する必要がある。

5. ブロック

複数の文が順接で連続している場合、制御構造としてはそれらを一続きのものとして扱った方が便利である。このような文の並びを**ブロック** (*block*) という。この場合、定義済み処理は通常の処理と同様にみなすことができる。

ブロックの特徴は、入り口点が 1 カ所でありそれ以外からブロック内の文に制御が移されることがないこと、および出口点が 1 カ所でありそれ以外の文からブロック外に制御が移ることがないことである。*6

拡張ブロック

文の並びが内部に if 文や do 文などの制御文を含む場合でも、並びが全体として入り口点 1 つ、出口点 1 つという条件を満たせば、並び全体を 1 つのブロックとみなすことができる。このような拡張ブロックもまた、ブロックという。プログラム全体もまた、1 つのブロックである。

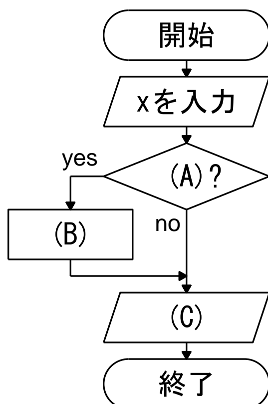
拡張ブロックでは、ブロックの中に他のブロックが存在することになる。このような状況を**入れ子** (*nesting*) という。

字下げ

1 つのブロックを構成する文は、それぞれ同じ数だけの空白を文の最初の文字の前に置き、同一のブロックにあることを明示する。これを**字下げ** (*indent*) という。字下げは何文字という決まりはないが、読みやすさのために必ず行うべきである。特に拡張ブロックの場合、どのネスティング・レベルにあるかを明示するために重要である。この冊子のプログラム例では、プログラム文に対し 4 文字の字下げ、拡張ブロックに対して 2 文字の字下げを行っている。

演習問題

演習問題 1



左は、この章の冒頭の `example_control` のプログラムをフローチャートで表したものである。(A)、(B)、(C) に当てはまる語を埋めなさい。

解答 (A) `x < 0`　(B) `x = -x`　(C) `x` を出力

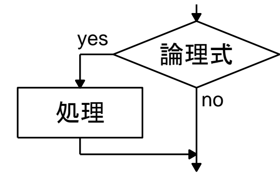
*6 ブロックを小ブロックに細分することは可能だが、実際上の意味はない。条件が満たされる中で最大の範囲の文から構成する。

§3.2 条件判断制御

1. 条件判断構文の種類

条件判断に関する制御構文は、分岐の形態により分類される。

単純分岐

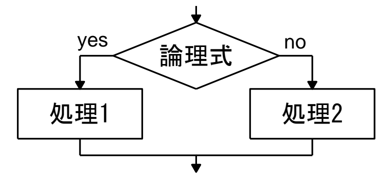


条件文は論理式で表され、その式の値が真であるとき、分岐して処理が行われる。偽であるときには何も処理を行わず次の文を実行する。

この制御構造は、次の選択分岐によって実現できる。それにもかかわらず単純分岐が残っている理由は、一つには以前のプログラムとの互換性を維持するためである。

もう一つの理由は、他の判断構文よりは CPU 時間がかからないので、少しでも速くするためである。

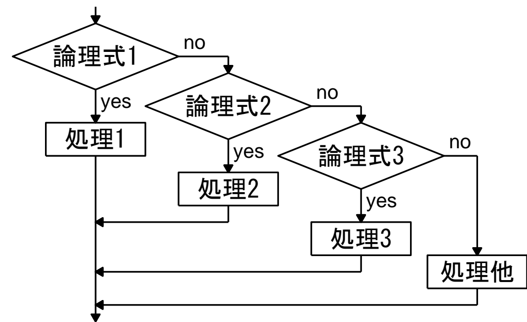
選択分岐



条件文は論理式で表され、その式の値が真であるとき、処理 1 が行われる。偽であるときには処理 2 が行われる。

もっとも基本的な条件判断構文であり、たぶんすべてのプログラム言語に実装されている。

多段分岐



一連の論理式で表される条件文からなる構文である。

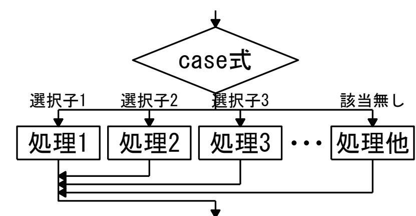
最初に論理式 1 が評価され、真であれば処理 1 が行われ、その後次の文が実行される。

論理式 1 が偽であるときは、論理式 2 が評価され、真であれば処理 2 が行われ、その後次の文が実行される。

このようにして論理式を次々に評価し、真であれば処理を行う。最後にどの論理式も偽であった場合は、「処理他」を実行したのち、次の文へと制御が移される。

このフローチャートは 3 段の場合を示したものであるが、実際には段数に制限はない。

多方向分岐



1 つの条件式 (case 式) と複数の条件の並びからなる構文である。

case 式が評価された後、選択子 1、選択子 2 ··· の中から適合するものが選択され、対応する処理が行われる。適合する選択子がない場合は、「処理他」が実行される。多段分岐との違いは、

(i) 多段分岐は、順番に条件の適合判定が行われる。したがって、論理式 1 も論理式 2 も真であるということも許される。この場合は先に評価した処理 1 が実現される。

一方多方向分岐では、選択判定は並列に行われる。したがって、選択子 1、選択子 2 ··· が同時に適合してしまうと、処理の選択に困る。したがってこれらの条件は、互いに排他的でなければならない。

(ii) case 式としては、論理式だけでなく、整数式、文字式も使用できる。実際、論理式では真、偽の二値しかとれないので、多方向分岐をする意味がない。

Fortan での実装

· Fortran では単純分岐は一つの文に対してのみ処理が行われ、ブロックを処理することはできない。

· 多段分岐は FORTRAN77 から導入された、選択分岐は多段分岐を利用して行える。

· 多方向分岐も、同じく FORTRAN77 で導入された。

2. if 文

まず、この章の冒頭のプログラムを、再び例題として取り上げ、解答例を示す。

例題 3-1

実数 x を入力し、その絶対値を出力するプログラムを、if 文を使って作成しなさい。

例題 3-1　のプログラム例

```
1   program example3_01
2   ! 読み込んだ値の、絶対値を出力する
3       implicit none
4       real    :: x
5
6       read(*,*) x
7       if ( x < 0 ) x = -x ! if 文
8       write(*,*) x
9
10      stop
11  end program example3_01
```

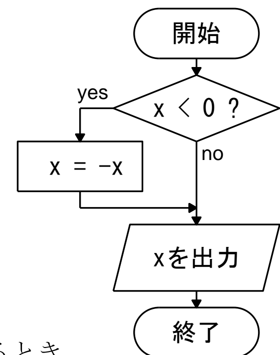
第 7 行を **if 文** または論理 if 文という。その形式は、

```
if ( 論理式 ) 実行文
```

である。ここで論理式は、真か偽かどちらかの値をとる式で、実行時に評価される。

実行文は、( ) の中が真である場合に実行される文で、単一の文である。ここにブロックを置くことはできない。複文を置くことはできるが、紛らわしいので置いてはいけない。[*7]

例題のアルゴリズムを、フローチャートで表すと右の図になる。



関係演算子

第 7 行の ( ) の中、 `x < 0` を**関係式** といい、数値の比較を行いその真偽を値とする論理式である。ここで「<」を**関係演算子** といい、演算子の左側の数値が右側の数値より大きいときに真の値をとる、二項演算子である。

関係演算子は全部で 6 種類ある。今、a、b を実数型あるいは整数型の式とするとき、

(1) `a < b` ··· a が b より小さいときに「真」、そうでなければ「偽」

(2) `a <= b` ··· a が b より小さいか等しいときに「真」、そうでなければ「偽」

(3) `a > b` ··· a が b より大きいときに「真」、そうでなければ「偽」

(4) `a >= b` ··· a が b より大きいか等しいときに「真」、そうでなければ「偽」

(5) `a == b` ··· a と b が等しいときに「真」、そうでなければ「偽」

(6) `a /= b` ··· a と b が等くないときに「真」、そうでなければ「偽」

[*7] 複文を置いてはいけない理由は、たとえば、

```
if ( 論理式 ) 実行文１ ; 実行文２
```

と書いた場合、これは、以下の 2 行の文と同義になり、論理式が偽の場合でも実行文 2 は実行されるからである。

```
if ( 論理式 ) 実行文１
実行文２
```

3. if 構文

ブロックへの単純分岐

例題 3-2

二次方程式、 $ax^2+bx+c=0$ の係数 a、b、c を入力し、方程式が二実根をもつときにその実根 x_1、x_2 を出力するプログラムを作成しなさい。

if 文により分岐される実行部分が、複数の文のブロックからなるときは、**if 構文** を使う。この例題のように、if 構文の一番簡単な形式は、

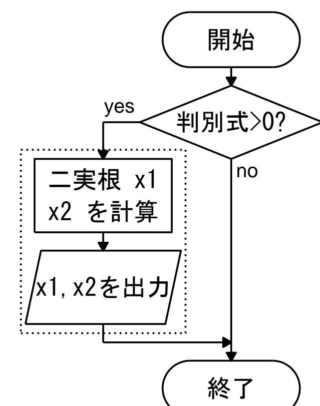
```
if ( 論理式 ) then
   ブロック
endif
```

である。if 構文は、FORTRAN77 ではブロック if 文と呼ばれた。

例題 3-2 のプログラム例

```
1   program example3_02
2   ! 二次方程式の二実根
3       implicit none
4       real    :: a, b, c, x1, x2
5
6       read(*,*) a, b, c
7       if ( b*b-4.0*a*c > 0 ) then
8         x1 = ( -b - sqrt(b*b-4.0*a*c) )/a/2.0
9         x2 = ( -b + sqrt(b*b-4.0*a*c) )/a/2.0
10        write(*,*) x1, x2
11      endif
12
13      stop
14  end program example3_02
```

例題のアルゴリズムをフローチャートで表すと右の図のようになる。ここで、点線で囲まれた部分がブロックである。



第 7 行と第 11 行が、if 構文の要素であり、( ) 内の論理式の値が真のとき、この 2 つの構文要素で囲まれた第 8 行から第 10 行までのブロックを実行する。

第 8 行から第 10 行までは、ブロックであることを明示するために 2 字分、字下げしている。このように、構造を明示するための字下げは重要なので、必ず行うべきである。

ブロックは単一の文でもよいので、if 文を使わずに if 構文だけでプログラムを書くことができる。ファイルが長くなり、多少実行速度が低下するが、こちらの方がわかりやすい。

選択分岐

例題 3-3

あるタクシー会社の料金は、最初の 2km までが 700 円、その後 300m まで 100 円の追加、以後 300m ごとに 100 円ずつが追加される。入力した距離 *k*[m] に対する料金を出力するプログラムを作成しなさい。ただし、*k* は整数とします。

真、偽、二分岐の場合の if 構文は、次の形式をとる。

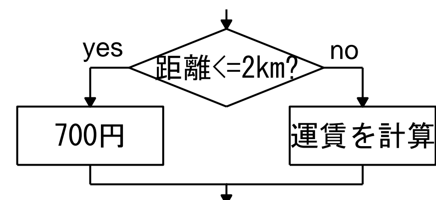
```
if ( 論理式 ) then
    ブロック 1
else
    ブロック 2
endif
```

( ) の中の論理式の値が真のとき、ブロック 1 が実行され、偽のときブロック 2 が実行される。

例題 3-3 のプログラム例

```
1   program example3_03
2   ! タクシーの料金
3       implicit none
4       integer :: k, fee
5
6       read(*,*) k
7       if ( k <= 2000 ) then
8         fee = 700
9       else
10        fee = 700 + ceiling((real(k)-2000.0)/300.0)*100.0
11      endif
12      write(*,*) "料金は",fee," 円"
13
14      stop
15  end program example3_03
```

例題のアルゴリズムの、if 構文の部分だけをフローチャートで表すと右の図のようになる。



論理式が真のときは `then` 以下、`else` の前までのブロックが実行され、実行の後は `endif` の次の文が実行される。

論理式が偽のときは `else` 以下、`endif` の前までのブロックが実行され、実行の後は `endif` の次の文が実行される。

このような if 文を書く場合、「`if`」、「`else`」、「`endif`」は同じ字数だけ字下げしなければならない。特に if 文が入れ子になった場合、そうでないとたいへん見にくくなる。

なお「`endif`」は、「`end if`」と表記することもでき、そのように書く人もある。

多段分岐

例題 3-4

入力された西暦年 y が、閏年か平年かを判別し、その結果を出力するプログラムを作成しなさい。

グレゴリオ暦の閏年は、以下のように定められている。

(i) 4 で割り切れない年は平年。

(ii) 4 で割り切れ、かつ 100 で割り切れない年は閏年。

(iii) 100 で割り切れ、かつ 400 で割り切れない年は平年。

(iv) 400 で割り切れる年は閏年。

上記のように分岐が複数ある時には、多段分岐を使う。その場合の if 構文は、次の形式をとる。

```
if ( 論理式 1 ) then
    ブロック 1
elseif ( 論理式 2 ) then
    ブロック 2
else
    ブロック 3
endif
```

論理式 1 の値が真のとき、ブロック 1 が実行される。論理式 1 が偽で、論理式 2 が真のときブロック 2 が実行される。すべて偽の場合はブロック 3 が実行される。`elseif` のブロックは、何段続けてもよい。

`else` 文およびその後のブロックは、必要がなければ省略することができる。

論理演算子

論理式 `p`、`q` について、「`p` かつ `q` である」という合成命題を表すには、Fortran では `p` と `q` を「`.and.`」という構文素で連結し、「 `p .and. q` 」と表す。このような構文素を**論理演算子** といい、論理式ををオペランドとし、論理値をとる演算子である。

論理演算子は次の 5 種類がある。今、`p`、`q` を論理式とするとき、

(1) 否定「 `.not. p` 」 ⋯ `p` の否定。`p` が真のとき偽、`p` が偽のとき真。

(2) 論理積「 `p .and. q` 」 ⋯ `p` かつ `q`。`p` と `q` が両方とも真のとき真。それ以外は偽。

(3) 論理和「 `p .or. q` 」 ⋯ `p` または `q`。`p` と `q` のどちらかが真のとき真。それ以外は偽。

(4) 相等「 `p .eqv. q` 」 ⋯ `p` と `q` が同値。`p` と `q` の真偽が一致するとき真。それ以外は偽。

(5) 排他的論理和「 `p .neqv. q` 」 `p` と `q` の真偽が一致しないとき真。それ以外は偽。

論理演算子の計算優先順位と真理表については、Appendix を参照していただきたい。

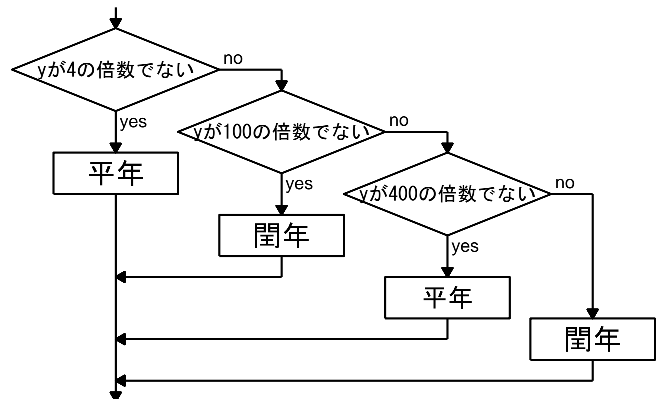
これらを使って、上記の閏年の条件をプログラム化すると、以下のようになる。

例題 3-4 のプログラム例

```
1  program example3_04
2      implicit none
```

```
 3      integer :: y
 4
 5      read(*,*) y
 6      if ( y/4*4 /= y ) then ! y が 4 で割り切れるときのみ等号成立
 7        write(*,*) "平年"
 8      elseif( ( y/4*4 == y) .and. (y/100*100 /= y) ) then
 9        write(*,*) "閏年"
10      elseif( (y/100*100 == y) .and. (y/400*400 /= y )) then
11        write(*,*) "平年"
12      else
13        write(*,*) "閏年"
14      endif
15
16      stop
17  end program example3_04
```

例題のアルゴリズムの、if 構文の部分だけをフローチャートで表すと右の図になる。



ここで、プログラム第 8 行は、閏年の定め方の条件 (ii) に相当する文であるが、注意してみれば、ここでは「4 で割り切れて」の部分の条件が実は必要がないことに気づくだろう。4 で割り切れない場合は、第 6 行の if 文の値が真になり、第 7 行が実行されたあと、第 15(16) 行に制御が移って第 8 行は実行されないからである。

したがって、この場合上記のプログラムは、論理演算子を含まない次の形に簡略化できる。

例題 3-4　のプログラム例　その２

```
 1  program example3_04_2
 2      implicit none
 3      integer :: y
 4
 5      read(*,*) y
 6      if ( y/4*4 /= y ) then
 7        write(*,*) "平年"
 8      elseif( y/100*100 /= y ) then
 9        write(*,*) "閏年"
10      elseif( y/400*400 /= y ) then
11        write(*,*) "平年"
12      else
13        write(*,*) "閏年"
14      endif
15
16      stop
17  end program example3_04_2
```

4. case 構文

例題 3-5

0 点から 100 点までの成績を入力し、90 点以上ならば「秀」、80 点以上ならば「優」、70 点以上ならば「良」、60 点以上ならば「可」、59 点以下ならば「不可」を出力するプログラムを作成しなさい。

多段分岐の条件式が、整数式の大小に関するものならば、多方向分岐に関する **case 構文** を使った方が、わかりやすくプログラムが書ける。[*8] case 構文の条件式は **case 式** といい、次の形式をとる。

```
select case ( case 式 )
  case (case 選択子 1)
     ブロック 1
  case (case 選択子 2)
     ブロック 2
  case default
     ブロック 3
end select
```

ここで case 式は、整数式である。(case 選択子) は、case 式の値に対する大小の条件であり、次のいずれかの形をとる。

(i) (n) ··· case 式の値が n に一致する。

(ii) (n:) ··· case 式の値が n 以上である。

(iii) (:n) ··· case 式の値が n 以下である。

(iv) (n:m) ··· case 式の値が n 以上 m 以下である。

n および m は、整数型定数あるいは名前付き整数型定数 (およびその初期化式) であり、整数型変数は使えない。すなわちコンパイル後に値が定まっていなければならない。

case 式の値が case 選択子 1 の範囲にあるとき、ブロック 1 が実行される。同様に、case 選択子 2 の範囲にあるとき、ブロック 2 が実行され、どの範囲にもなければ、case default の後のブロックが実行される。なお、case default 文とその後のブロックは、必要がなければ省略できる。

重要な注意点として、「これらの case 選択子の示す範囲が重なってはならない」という条件がある。

case 構文を使って、例題をプログラム化すると、以下のようになる。

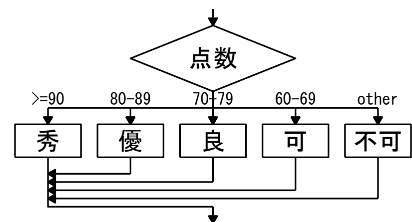
例題 3-5 のプログラム例

```
1  program example3_05
2      implicit none
3      integer :: point
4
5      read(*,*) point
6      select case (point)
```

[*8] 文字式に対しても使える。その場合文字コードの大小で判断する。

```
 7         case(90:)
 8           write(*,*) '秀'
 9         case(80:89)
10           write(*,*) '優'
11         case(70:79)
12           write(*,*) '良'
13         case(60:69)
14           write(*,*) '可'
15         case default
16           write(*,*) '不可'
17       end select
18
19       stop
20   end program example3_05
```

例題のアルゴリズムの、case 構文の部分だけをフローチャートで表すと右の図になる。



case 構文による多方向分岐と、if 構文による多段分岐との違いは、次のようなものである。

(i) case 構文では、case 式に整数型 (および文字型) の大小しか扱えないが、if 構文の条件式では実数型も扱える。

(ii) if 構文では、条件は順に評価されるので、条件が重なってもよいが、case 選択子では範囲が重なることはできない。

(ii) case 構文では、case 選択の条件は実行前に確定していなければならないが、if 構文の条件は、構文実行時に確定していればよい。

すなわち、if 構文の方が適用範囲が広く、case 構文は使った方が明らかにプログラムがわかりやすくなる場合にだけ使用される。

演 習 問 題

演習問題 2

例題 3-5 を、if 構文の多段分岐を使って書き直しなさい。

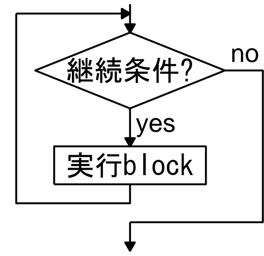
解答 プログラムの if 構文の部分は、以下のようになります。

```
if(point >= 90) then
  write(*,*) '秀'
elseif(point >= 80) then
  write(*,*) '優'
elseif(point >= 70) then
  write(*,*) '良'
elseif(point >= 60) then
  write(*,*) '可'
else
  write(*,*) '不可'
endif
```

§3.3 反復制御

1. 反復構文の種類

反復 (ループ) に関する制御構文は、ループからの脱出形態により分類される。



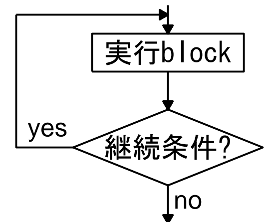
前条件判定反復

まず、継続条件が評価され、偽であればブロックを実行せず、ループから脱出する。真であればブロックが実行される。

ブロックの実行後、再び継続条件が評価され、真であれば再びブロックが実行される。このようにして継続条件が真である間、ブロックの実行が繰り返される。継続条件が偽になる条件は、ユーザが設定しなければならない。

この yes/no が逆、すなわち継続条件が偽である限りループを反復する構文のある言語もある。

後条件判定反復

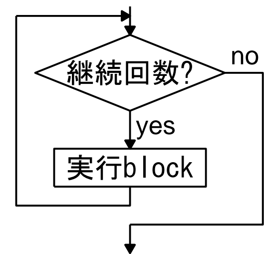


まずブロックが実行され、その後継続条件が評価される。偽であればループから脱出し、真であれば再びブロックが実行される。前条件判定反復と同じく、継続条件が真である間、ブロックの実行が繰り返される。継続条件が偽になる条件は、ユーザが設定しなければならない。

この構文にも、この yes/no が逆、すなわち継続条件が偽である限りループを反復する構文のある言語もある。

前条件判定反復との違いは、継続条件にかかわらず、必ず 1 回はブロックが実行されることである。

定数回反復



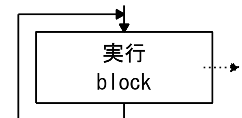
ループを反復実行する回数を、ループ実行前にあらかじめユーザが指定しておく。プログラムはブロックを実行するごとに、回数をチェックし決められた回数に達したらループから脱出する。

右図はブロックの実行前に反復回数のチェックを行っているが、実行後に行っても本質的に変わらない。

配列要素・集合要素反復

配列型、および集合型のある言語において、そのすべての要素について、それぞれ処理を反復する。

不定回反復 (無限ループ)



この構文では、ループ脱出の条件は指定されない。単純にループを反復する。

ループからの脱出は、反復構文以外の制御文で行う。

Fortan での実装

· Fortran90 から前条件判定反復が導入された。逆の条件、すなわち継続条件が偽である限りループを反復する構造は、継続条件に`.not.`を付すことにより実現される。

· Fortran には後条件判定反復構文はない。これは、データ解析には幾分不便ではある。

· Fortran には、厳密に言えば定数回反復構文はない。ただし、制御変数と呼ばれる整数型の変数と、前条件判定反復の組み合わせで、定数回の反復を実現している (FORTRAN66 時代から)。

· Fortran95 から、配列に対する配列要素反復構文が導入された。ただし、ここでは解説せず、配列のところで説明する。

· Fortran には集合型 (列挙型) の概念がないので、集合要素反復構文はない。

· Fortran90 から、不定回反復構文が導入された。

2. do while 文

例題 3-6

1 から n までの自然数の和を $S(n) = \sum_{i=1}^{n} i$ とします。$n = 1,2,3,\cdots$ について、1 行に n と $S(n)$ を出力して、表を作成しなさい。ただし、$S(n)$ の値が 100 を超えた時点で終了するものとします。

「… という条件をみたしている間 … を繰り返す」という前条件判定反復構造を表すには、Fortran では次の **do while 文** を使う。ここで enddo は do 実行ブロックの終了を示す文であり、**do の端末文** という。

```
do while ( 論理式 )
  ブロック
enddo
```

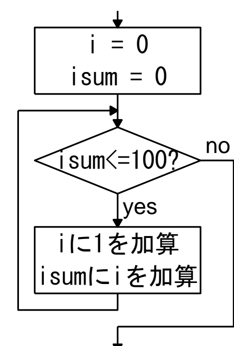
論理式が真であれば、ブロックを実行し、偽であれば enddo 文の次の文に制御を移す。

逆に「… という条件を初めてみたすまで … する」という構造を表すためには、論理式の先頭に否定の論理演算子.not. を挿入すればよい。

例題 3-6 のプログラム例

```
1   program example3_06
2       implicit none
3       integer :: isum, i
4
5       i = 0
6       isum = 0
7       write(*,*) "           n        S(n)"
8       write(*,*)
9       do while ( isum <= 100 )
10        i = i + 1
11        isum = isum + i
12        write(*,*) i, isum
13      enddo
14
15      stop
16  end program example3_06
```

例題のアルゴリズムの、主要な部分だけのフローチャートを右の図に示す。



まず第 3 行で、自然数を表す整数型変数 i と、その和を格納する整数型変数 isum を宣言し、第 5 行と第 6 行で、それらを 0 に初期化しておく。

第 7 行は、表の項目を出力する行である。つづいて第 8 行で空行を出力し、項目と表の数字の間を 1 行あける。

第 10 行で i を 1 つずつ増していき、第 11 行で isum に i を加えることにより、$S(n)$ の値を得る。それらを第 12 行で出力する。

この処理を第 9 行の条件が満たされる間、繰り返す。

3. do 構文

基本形な **do** 構文

例題 3-7

A 君は、期末試験に 5 科目を受験した。試験の成績は、0 点から 100 点までの整数値がつけられる。その点数を 1 行に 1 科目ずつ、たとえば、

67
73
81
66
75

のように入力するとき、その平均点を算出するプログラムを作成しなさい。

一般にプログラミング言語では、繰り返しの範囲や回数があらかじめわかっている場合、do while 文のような条件判定反復構文ではなく、定数回反復構文を使う。その方がわかりやすい上に計算も速いからである。

ただし、Fortran には厳密な意味での定数回反復構文はない。その代わりに、**do 制御変数** という整数型の変数を使って反復回数を指定する、**do 構文** を使う。do 構文の基本的な形式を以下に示す。

```
do 制御変数 = 初期値, 終了値
    ブロック
enddo
```

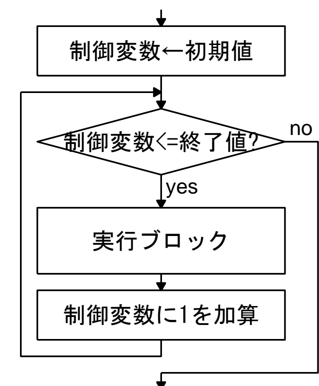
上記の do 構文は、右図の前条件判定反復構造と等価である。

(1) まず loop に入る直前に、制御変数に初期値が代入される。
(2) 判定部分で制御変数が終了値以下であるかどうかを判定する。
(3) 結果が真であればブロックを実行し、偽であればループから脱出する。
(4) ブロック実行後に、制御変数に 1 が加えられる。
(5) 再び判定部分で、制御変数が終了値以下であるかどうかを判定する。

この構文による繰り返しの回数は、次の式で表される。

```
反復回数 = ( 終了値 - 初期値 ) + 1
```

終了値が初期値より小さい場合には、loop は一度も実行されずに、端末文の次の文に制御が移される。



例題 3-7 のプログラム例

```
1  program example3_07
2      implicit none
3      integer :: i, ip, isum
4
5      isum = 0
6      do i = 1, 5
7        read(*,*) ip
```

```
8          isum = isum + ip
9        enddo
10       write(*,*) real(isum)/5.0
11
12       stop
13   end program example3_07
```

この do loop では i が制御変数である。一般に do 制御変数は、 do 構文で制御変数として指定されたときに、do の実行ブロック (do loop) の中でのみ、制御変数としての役割を行う。do loop の外では通常の整数型変数であり、通常の整数型で型宣言を行う。

一方、do loop の中では、制御変数の値は do 構文によって制御されるので、**ユーザは代入文等で制御変数の値を変更してはならない**。これは多くの初心者が陥る、代表的な誤りの一つである。

do 構文の拡張

例題 3-7 の系

B 君は、今学期だけでなく、これまでに受けた期末試験すべてについて、それぞれ平均点を算出することにした。ただ、学期ごとに受けた試験の数が異なるので、これらを一つのプログラムで処理するために、最初に受けた試験の数を挿入することにした。たとえば、

4
65
78
73
82

のように入力する。このようなデータから、平均点を算出するプログラムを作成しなさい。

do 構文の初期値、終了値は、整定数だけでなく、整数変数または一般の整数式で指定することもできる。do 構文が実行されるときに、それらの式に含まれる変数がすべて確定し、初期値、終了値が計算できればよい。

例題 3-7　のプログラム例 その 2

```
1    program example3_07_2
2        implicit none
3        integer :: n, i, ip, isum
4
5        read(*,*) n
6        isum = 0
7        do i = 1, n
8          read(*,*) ip
9          isum = isum + ip
10       enddo
11       write(*,*) real(isum)/real(n)
12
13       stop
14   end program example3_07_2
```

増分を含む do 構文

例題 3-8

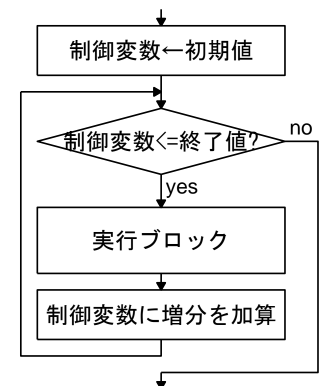
入力した自然数 n $(\geqq 2)$ について、1 から n までのすべての偶数の和を出力するプログラムを作成しなさい。ただし、n が奇数の場合は、$n-1$ までの和とします。

do 構文の基本形では、ループの実行後に制御変数に 1 を加えてたが、その代わりに 0 以外の任意の整数値を加えることができる、この整数を do の**増分** といい、次のような形式で指定する。

```
do 制御変数 = 初期値, 終了値, 増分
    ブロック
enddo
```

増分も初期値や終了値と同様に、整数変数や整数式で与えることができる。

上記の do 構文は、右図の前条件判定反復構造と等価である。最下段の加算の部分が、基本形の 1 から増分に置き換わっている。



上の例題では、和は 2 から開始され、1 つおきに加えられるので、初期値は 2、増分も 2 である。

終了値は、n が偶数であれば n でよいことはすぐわかる。n が奇数の場合、$n-1$ が和に加えられた後、増分 2 が加算されて、制御変数は $n+1$ になる。これは n を超えるので、やはり終了値に n を指定しておけば、そこでループの外に出ることがわかる。

すなわち、この構文による繰り返しの回数は、次の式で表される。

```
反復回数 = ( 終了値 - 初期値 + 増分 )/増分
```

この式は整数式であり、除算は整除算、すなわち余りを切り捨てた商の値である。

この式の形を見てもわかるとおり、**増分は 0 であってはならない**。もし増分が 0 になってしまった場合の処理は処理系依存である。エラーメッセージを出力して停止するコンパイラもあれば、暴走して無限ループに陥るコンパイラもある。

例題 3-8 のプログラム例

```
1   program example3_08
2       implicit none
3       integer :: n, i, isum
4
5       read(*,*) n
6       isum = 0
7       do i = 2, n, 2
8         isum = isum + i
9       enddo
10      write(*,*) isum
11
12      stop
13  end program example3_08
```

負の増分

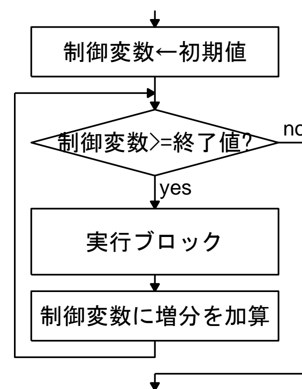
例題 3-9

入力した自然数 n (≥ 2) について、n が偶数の場合は 1 から n までのすべての偶数の和、n が奇数の場合は 1 から n までのすべての奇数の和を出力するプログラムを作成しなさい。

これまでのように、増分を 2 とした do 構文によりプログラムを作成しようとすると、n の偶奇性により if 構文で場合分けを行ったうえで、それぞれの場合について do 構文を loop させなければならない。しかし、増分を-2 として制御変数を n から 2 ずつ減らしていくようにすれば、場合分けをする必要はなくなる。

増分が負の場合の do 構文の形式は、増分が正の場合と全く同じである。ただし、反復を行うためには、初期値は終了値よりも小さな値でなければならない。

増分が負である do 構文と等価な、前条件判定反復構造のフローチャートは右図のようになる。増分が正の場合とは、判定部分の不等号の向きが逆になっている。



上の例題では、和は n から開始され、1 つおきに加えられるので、初期値は n、増分は-2 である。

ループを最後に回るときの制御変数の値は、n が偶数であれば 2、n が奇数であれば 1 であるべきだが、終了値として 1 を指定しておけば、2 または 1 を加算した後ループの外へ出て行くことがわかる。

すなわち、この構文による繰り返しの回数も、増分が正の場合と同様に、次の式で表される。

反復回数 = (終了値 - 初期値 + 増分)/増分

さらに一般的に、商が負になる場合も考慮すれば、その場合は 1 回もブロックは実行されないので、反復回数は 0 だから、

反復回数 = max((終了値 - 初期値 + 増分)/増分, 0)

と表される。

例題 3-9 のプログラム例

```
1   program example3_09
2       implicit none
3       integer :: n, i, isum
4
5       read(*,*) n
6       isum = 0
7       do i = n, 1, -2
8         isum = isum + i
9       enddo
10      write(*,*) isum
11
12      stop
13  end program example3_09
```

多重ループ

例題 3-10

p、q、r を自然数とするとき、$p+q+r=12$ となる組み合わせは何通りあるか計算しなさい。ただし、前後の順序が異なるだけのものは、1 通りと数えます。

題意より、$p \leqq q \leqq r \leqq 10$ として、条件をみたす組み合わせを求めればよいことがわかる。

例題 3-10　のプログラム例

```
1   program example3_10
2       implicit none
3       integer :: p, q, r, number
4
5       number = 0
6       do p = 1, 10
7         do q = p, 10
8           do r = q, 10
9             if( p+q+r == 12 ) number = number + 1
10          enddo
11        enddo
12      enddo
13      write(*,*) number
14
15      stop
16  end program example3_10
```

do loop の範囲に注意。答は 12 通りである。

4. do 文

「··· を単純に繰り返す」という不定回反復構造を表すために、Fortran では以下の形式の **do 文** を使う。

```
do
    ブロック
enddo
```

この構造は、それ自身ではループを脱出する手段を持たず、必ずジャンプ制御文、または停止制御文と組み合わせて使わなければならない。そのため、実例はそれらの制御文の説明のところで示す。

演習問題

演習問題 3

例題 3-9 で n が次の値のとき、ループから出た直後の制御変数 i の値はいくつになりますか。

(1) 4　　(2) 5

解答後に write 文を挿入して確認しなさい。

解答 (1) 0　　(2) -1

§3.4 ジャンプ制御

Fortran90/95 のジャンプ制御文には、次の 3 種類がある。

○ cycle 文

○ exit 文

○ goto 文

このうち、cycle 文と exit 文は、do (構) 文のみに使われるジャンプ制御文である。goto 文は汎用のジャンプ制御文であり、制御構造の外でも使うことができる。

1. cycle 文

例題 3-11

入力した自然数 n について、その約数をすべて出力しなさい。また、その和を計算して出力しなさい。

cycle 文 は do loop の中で使われる文であり、多くの場合論理 if 文と組み合わせて使われる。

```
do ........
   ブロック 1
 ..... cycle   ! -> enddo 文へ
   ブロック 2
enddo
```

cycle 文が実行されると、以後の実行文を飛ばして do 端末文に制御を移す。上の例ではブロック 1 を実行後に cycle 文が実行されたならば、ブロック 2 は実行せずに、enddo 文にジャンプする。

例題 3-11 のプログラム例

```
1   program example3_11
2       implicit none
3       integer :: i, n, isum
4
5       read(*,*) n
6       isum = 0
7       do i = 1, n
8         if( n/i*i /= n ) cycle
9
10        write(*,*) i
11        isum = isum + i
12      enddo
13      write(*,*) "sum=",isum
14
15      stop
16  end program example3_11
```

解答　i が n の約数であるということは、n を i で割った余りが 0 ということであり、これは、n/i*i が n に等しいことと同値である。do loop で i が n の約数であるときに、第 10 行、第 11 行が実行され、約数でないときには 12 行の enddo 文にジャンプし、次の i についてのループに移る。

多重 do loop 中の cycle 文

例題 3-12

10000 より小さい素数の中で、最大のものを求めなさい。

以下のような、cycle 文を含む多重 do loop 構造があるとする。

```
1      do
2        do
3          .....
4          .... cycle
5          do
6            .....
7          enddo
8        enddo
9      enddo
```

(i) 一般に cycle 文が実行された場合、そのジャンプ先は **cycle 文が含まれる中でもっとも内側のループ**の端末文となる。したがって、この場合第 4 行の cycle 文のジャンプ先は、第 8 行の enddo 文である。

(ii) 第 9 行の enddo 文を cycle 文のジャンプ先にしたい場合は、対応する do 構文に**構文名** をつけて、その構文名を cycle 文で指定する。

構文名は、以下のように do 文の文頭にコロンとともに「**構文名:**」を挿入して指定する。また、対応する端末文の後にもその構文名を書く (コロンはつけない)。構文名の付け方は Fortran の命名規則に従う。

その上で、cycle 文の後にその構文名を指定すれば、実行時にその端末文へとジャンプする。

```
構文名: do ........
          cycle 構文名
      enddo 構文名
```

(iii) 第 4 行から、第 7 行の enddo 文へジャンプはできない。その do loop に含まれないからである。

例題 3-12 のプログラム例

```
1   program example3_12
2       integer :: i, j
3
4   lp: do j = 10000-1, 2, -2
5         do i= 3, j/2, 2
6           if( j/i*i == j ) cycle lp
7         enddo
8         write(*,*) j
9         stop
10      enddo lp
11
12      stop
13  end program example3_12
```

上の例では、j を 10000 未満の奇数について上からループさせている。その j について、奇数 i をその約数であるかどうか調べる。一つでも約数であれば、i についてのループを打ち切り、次の j を調べる。

2. exit 文

例題 3-13

入力した自然数 n について、n 自身をのぞく、その最大の約数を求めなさい。

exit 文 も do loop の中で使われる文であり、多くの場合論理 if 文と組み合わせて使われる。

```
do ........
   ブロック 1
 ..... exit   ! -> enddo 文の次の文へ
   ブロック 2
enddo
```

exit 文が実行されると、do 構文を終了して、do 端末文の次の文に制御を移す。上の例ではブロック 1 を実行後に cycle 文が実行されたならば、ブロック 2 は実行せずに do loop を脱出し、enddo 文の次の文にジャンプする。

例題 3-13 のプログラム例

```
1   program example3_13
2       integer :: n, i
3
4       read(*,*) n
5       do i = n/2, 1, -1
6         if( n/i*i == n ) exit
7       enddo
8       write(*,*) "最大の約数=", i
9
10      stop
11  end program example3_13
```

n が i で割り切れたところで、do loop から脱出して write 文を実行する。

不定回 do 文中の exit 文

例題 3-14

C 君は、これまでに受けたすべての試験の平均点を算出することにした。ただ、数が多くなると数えるのが面倒なので、試験の点数を 1 行に 1 つずつ入力したあと、最後に「999」を入力してデータの終了を表すことにした。たとえば、次のように入力する。

71

89

75

69

999

このようなデータから、受けた試験の回数と、その平均点を算出するプログラムを作成しなさい。

このプログラムを、たとえば do while 文を使って作成すると、あまりすっきりしない形になる。

例題 3-14 のプログラム例 その 1

```
1  program example3_14_1
2      implicit none
3      integer :: i, isum, ip = 0 ! 999 以外の適当な値
4
5      i = 0
6      isum = 0
7      do while ( ip /= 999 )
8        read(*,*) ip
9        if( ip /= 999 ) then
10         i = i + 1
11         isum = isum + ip
12       endif
13     enddo
14     write(*,*) "回数",i ,"  平均" ,isum/real(i)
15
16     stop
17 end program example3_14_1
```

このプログラムの問題点は、ループに入る前に ip に仮の値を入れて確定させておかなければならないこと、do loop の中に if 構文が入って、構造が複雑になっていること、第 7 行と第 9 行で、同じ比較を 2 回行わなければならないこと、である。

これは、不定回反復構文 (無限ループ) と exit 文を組み合わせれば、もっとすっきりした形で書ける。

例題 3-14 のプログラム例 その 2

```
1  program example3_14_2
2      implicit none
3      integer :: i, isum, ip
4
5      i = 0
6      isum = 0
7      do
8        read(*,*) ip
9        if( ip == 999 ) exit
10       i = i + 1
11       isum = isum + ip
12     enddo
13     write(*,*) "回数",i ,"  平均" ,isum/real(i)
14
15     stop
16 end program example3_14_2
```

多重 do loop 中の exit 文

これについては、扱いは cycle 文と同様である。すなわち exit 文が実行されたときのジャンプ先は、exit 文が含まれる中でもっとも内側のループの端末文の次の文となる。また、do 構文に構文名を付加し、その構文名を指定すれば、その do 構文の端末文の次の文にジャンプする。

3. goto 文

例題 3-15

入力した自然数 n が素数かどうかを判定し、素数ならば「n **は素数**」、素数でないならば「n **は合成数**」と出力するプログラムを作成しなさい (出力の「n」の部分には n の値が入る)。

goto 文は汎用のジャンプ文であり、do 構文や if 構文でなくても、実行可能な位置ならば置くことができる。

goto 文のジャンプ先は実行文でなければならず、ジャンプ先を指定するために、実行文の先頭に 5 ケタ以内の数字からなる**文番号**と 1 個以上の空白を置く。goto のあとにその文番号を書くことによって、ジャンプ先を指定する。

```
        ..... goto 文番号 ! ->その文番号の実行文

   文番号 実行文
```

do 構文や if 構文があるプログラムでは、それらの構文中から構文外に goto 文でジャンプして脱出させることは可能である。ただし、逆に**構文外から構文内に goto 文でジャンプさせることはできない**。

例題 3-15 のプログラム例

```
1   program example3_15
2       implicit none
3       integer :: n, i
4
5       read(*,*) n
6       do i = 2, n/2
7         if( n/i*i == n ) goto 10
8       enddo
9       write(*,*) n," は素数"
10      stop
11
12   10 write(*,*) n," は合成数"
13      stop
14  end program example3_15
```

この場合、ループからの脱出先が enddo 文の直後ではないので、exit 文でなく goto 文を使っている。なお「goto」は、「go to」と空白を入れて書いてもよい。

goto 文の多用に関する注意

goto 文は機械語命令に近く、たいへん汎用性の高い文である。しかし便利だからといってこれを使いすぎると、あちこちに制御が飛ぶ、たいへんわかりにくいプログラムになってしまう。*9

したがって、他の制御文で書けるならば goto 文の使用は控えるべきであり、とくにプログラムの前方へジャンプする goto 文は使うべきではない。とはいえ、データ解析では goto 文を使った方がわかりやすい場合がよくあるので、その場合は遠慮なく使ってよい。*10

*9 俗に「スパゲッティ・プログラム」という。

*10 「goto 文は絶対に使ってはならない」と主張する一派がいて、著者はこれを「ダイクストラ厨」と勝手に呼んでいる。

§3.5 順接と停止

Fortran90/95 には、順接のための制御文として continue 文、停止制御文として stop 文がある。

1. continue 文

continue 文は、何も動作は行わず次の実行文に制御を渡すだけである。continue 文が使われる目的は、主に goto 文の適当なジャンプ先がない場合に、文番号を付けてジャンプ先として使うことである。

```
文番号 continue
```

何も動作は行わないが、continue 文は実行文である。

2. stop 文

stop 文は、プログラムを終了させるための制御文である。ただし、stop 文が省略されても、end 文に遭遇すればプログラムは停止する。

stop 文は 1 つのプログラムに 1 個とは限らず、複数の場所に置くことができる (ex. 例題 3-15)。どの stop 文で停止されたかを判別するために、stop 文のあとに文字定数、あるいは 5 ケタ以内の正の整定数を指定することができる。これらはプログラムの停止時に標準出力に出力される。

```
stop {文字定数|整定数}
```

これらは名前付き定数は不可であり、定数でなければならない。

演習問題

演習問題 4

例題 3-15 を、goto 文を使わずに書き直しなさい。

解答 答は一通りではありませんが、たとえば exit 文を使えば、主要部分は以下のようになります。

```
read(*,*) n
do i = 2, n/2
  if( n/i*i == n ) exit
enddo
if( i == n/2+1 ) then
  write(*,*) n," は素数"
else
  write(*,*) n," は合成数"
endif
```

3 章 の 章 末 問 題

【問題 3-1】

自然数 m、n を入力したとき、ユークリッドの互除法により、その最大公約数 (GCM) をもとめるプログラムを作成しなさい。

(注) ユークリッドの互除法とは、以下の手順で最大公約数を求める方法である。

(i) m / n の余りを k とする。

(ii) $k \neq 0$ ならば、n を m に代入し、k を n に代入し、(i) へ戻る。

(iii) $k = 0$ ならば、n が最大公約数である。

【問題 3-2】

フィボナッチ数列、

$a_0 = 1$、$a_1 = 1$、$a_2 = 2$、$\cdots a_n = a_{n-2} + a_{n-1}$、$\cdots$

を a_{10} まで計算して出力しなさい。

【問題 3-3】

自然数 n を入力したとき、それを素因数分解して出力するプログラムを作成しなさい。

【問題 3-4】

レジで客が n 円 ($n < 1000$) の買い物をして、千円札を出しておつりを要求します。おつりを 500 円、100 円、50 円、10 円、5 円、1 円硬貨で支払うとき、硬貨の枚数が一番少なくなるためには、それぞれ何枚出したらよいか計算するプログラムを作成しなさい。

【問題 3-5】

4 けたの数 (1000-9999) で、次の条件をみたすものは、それぞれ何個あるか。 プログラムで求めなさい。

(1) 4 つの数字の和が 12 である。

(2) 「1」を一つ以上含む。

(3) 4 つの数字が全て異なる。

【問題 3-6】

あるバス会社の料金は、最初の 2km までが 160 円、その後 1km につき 50 円の割合で追加され、10 円未満は切り上げられる。また、子供料金は、大人料金の半額で、10 円未満は切り上げられる。入力した距離 k[m] に対する御店料金と子供料金を出力するプログラムを作成しなさい。ただし k は整数とします。

第 4 章 配列

§4.1 配列の基礎概念

1. データ並びとしての配列

添え字付き変数

同種類の多数データを扱う場合、データの個数 n が大きくなるとその 1 つ 1 つに別の変数名をつけて扱うのはたいへん手間がかかる。たとえば $n = 5$ のとき、`x1,x2,x3,,x4,x5` の 5 個の変数の平均を求める場合、

```
average = ( x1 + x2 + x3 + x4 + x5 )/5.0
```

と書くことはできるが、データの個数 n が変わった場合にはプログラムを書き直さなければならないし、n が大きくなれば書き下すこと自体がたいへんな手間になる。

このような場合、数学では変数 x に**添え字** (*subscript*) をつけて、x_1、$x_2 \cdots$ のように表しておき、

$$average = \frac{1}{n}\sum_{i=1}^{n} x_i$$

というように簡明に表すことができる。

Fortran でも同様のことが可能である。ただし添え字として下付き文字は使えないので、代わりに「`( )`」の中に添え字をいれて、`x(1)`、`x(2)`、$\cdots$ `x(n)` のように表す。これにより上の数式に対応する Fortran の式は、次のように書くことができる (説明は後述)。

```
average = sum(x(1:n))/real(n)
```

このような `x` を**配列** といい、個々の `x(1)`、`x(2)` $\cdots$ のことを**配列要素** という。配列に対して、通常の量を**スカラー** という。

基本的な配列 (p 61)

ある配列に対して型宣言を行えば、その配列要素は、すべてその型を持つスカラー変数と同様に扱うことができる。 たとえば次のように整数の配列 `x` の 2 番めの配列要素と実数の配列 `y` の 5 番目の配列要素を加えて、実数の配列 `z` の 3 番目の配列要素に代入するいうようなことも可能である。

```
z(3) = real(x(2)) + y(5)
```

一般に、配列中の配列要素の数は有限である。すなわち配列は同じ型の有限個の配列要素の並びとみなすことができ、数学のような無限個の並びは定義できない。配列の配列要素の数を**寸法** (*extent*) という。たとえば、3 個の配列要素 `x(1)`、`x(2)`、`x(3)` からなる配列 `x` の寸法は 3 である、という。

二次元の配列 (p 68)

表計算ソフトでは、データが行、列の二次元の平面上に整列している。Fortran では、二次元の配列がこれに対応している。二次元の配列を `a` としたとき、その第 `i` 行、第 `j` 列の配列要素は、`a(i,j)` のように二つの添え字で表される。

このように、配列の要素を指定するのに必要な添え字の数を、配列の**次元** (*dimension*) という。Fortran では、最大 7 次元までの配列が使用できる。

一般に寸法は各次元ごとに異なる。配列のすべての次元について、その寸法を掛け合わせたものは、その配列のすべての配列要素の数であり、**配列の大きさ** (*size*) という。

2. 変数としての配列

配列の演算 (p 74)

数学や物理学では、ベクトルという概念がある。 たとえば力の合成の問題で力 f と力 g、およびその合力 h との関係は、それぞれの力の x、y、z 成分をそれぞれ f_x のように書けば、

$$\begin{cases} h_x = f_x + g_x \\ h_y = f_y + g_y \\ h_z = f_z + g_z \end{cases}$$

となるが、f_x、f_y、f_y を成分とするベクトル $\vec{f}$ 等を使えば、上式は、

$$\vec{h} = \vec{f} + \vec{g}$$

と簡潔に表記することができる。

Fortran の配列に関しても、個々の配列要素に対してそれぞれ演算する代わりに、配列を 1 つの変数のようにみなして演算を行うことができる。このようなことから、配列を**配列変数** ともいう。たとえば、寸法 3 の一次元配列 a、b、c に対して、

```
c(1) = a(1) + b(1)
c(2) = a(2) + b(2)
c(3) = a(3) + b(3)
```

と書く代わりに、

```
c = a + b
```

と、配列同士の演算の形で書くことができる。

要素別処理関数 (p 76)

関数の中には、四則計算と同様に、配列をその引数としたとき、そのすべての配列要素について、それを引数とした関数値を配列要素とする配列を出力する機能を持つものがある。そのような関数を**要素別処理関数** という。たとえばそれぞれ 3 個の要素を持つ配列 a、b に対して、

```
b = abs(a)
```

という Fortran 表現は、

```
b(1)=abs(a(1))
b(2)=abs(a(2))
b(3)=abs(a(3))
```

と同値である。

配列関数

また、関数の中には、引数がスカラーではなく配列専用のものがある。そのような関数を**配列関数** という。前ページの sum も配列関数の一つである。

配列関数の中には、戻り値 (関数値) がスカラーでなく、配列のものもある。

多次元の配列変数

ベクトルを一次元の配列変数に対応させたように、行列を二次元の配列変数に対応させ、二次元配列で行列計算を行うことができる。ただし、加法、減法、スカラー倍以外は、特別な配列関数を使用しなければならない。

同様に三次元以上の配列変数も扱うことができる。

形状適合と例外

数学では、行列とベクトルを等号で結ぶことはできない。また行列の積の計算では、左側の行列の列の数が、右側の行列の行の数に等しくなければならないという制約がある。Fortran の配列にも同様の制約があり、たとえば代入文ならば左辺の配列変数と、右辺の配列の式は次元数が等しく、またすべての次元の寸法が等しくなければならない。このような配列演算が可能であるような条件を、**形状適合** という。

例外は配列の式中のスカラー変数であり、すべての配列と形状適合する。たとえば配列 `c` に対して、

```
c + 1
```

という式は、`c` のすべての配列要素に対して、`1` を加えることを意味し、形状適合でありエラーにはならない。

3. 配列とテンソルとの対応

数学・物理学の用語との対応

FORTRAN の配列用語は、一般の数学・物理用語とは一致していない。混乱を防ぐために、次に対応表を示しておく。

FORTRAN	数学・物理学
配列 (array)	テンソル (tensor)
次元 (dimension)	階 (order)
0 次元配列 = スカラー	スカラー (scalar)
1 次元配列	1 階テンソル = ベクトル (vector)
2 次元配列	2 階テンソル = 行列 (matrix)
多次元配列	高階テンソル
寸法 (extent)	次元・次 (dimension)
大きさ (size)	—

特殊な配列

数学・物理学の概念には現れない、FORTRAN 独特の配列も存在する。

(i) 数学では、1 次のベクトル $\{x\}$ は、スカラー x と同義である。しかし FORTRAN では、寸法 1 の 1 次元配列は、スカラー変数とは全く別モノである。`y` を前者、`z` を後者とすると、代入文、

```
z = y
```

はエラーになる。

同様に、数学では n 行 1 列の行列は、n 次ベクトルと同義であるが、FORTRAN ではこれらは別モノである。以下、高次元の配列でも同様である。

(ii) 大きさがゼロの配列、がある。(p 71)

Fortran(90/95) では、寸法が 0 の配列を定義することができる。この場合エラーは起こらない。

そして、配列の大きさは、各次元の寸法の積であるから、次元の寸法がどれか一つでも 0 であれば、配列の大きさもまた 0 になる。

§4.2 配列の基本

まず、一次元配列により、配列の基本的設定と操作について述べる。

1. 配列の宣言

例題 4-1

n 個の実数 x を入力し、その標本平均と標本分散を出力するプログラムを、配列を使って作成しなさい。ただし、n は 10 以下の整数で、最初に読み込むものとします。

配列を使うために、まず配列を宣言する。Fortran90/95 のもっとも基本的な配列宣言は、

```
型名 :: 配列名（寸法宣言子）
```

という形式である。ここで、「型名」は `integer`、`real` などの型の名前、「配列名」は一般的な Fortran の命名規則にしたがう。「寸法宣言子」は想定される配列に格納する最大のデータ数を指定する。

添え字の大きさがこの寸法を超えた場合の動作は保証されない。もしエラーが起きても、その原因個所の特定は難しい。また、エラーが起きないままプログラムが予期しない動作をする場合もあり、その場合の原因の特定はさらに難しくなる。したがって、配列の寸法の宣言には十分注意しなければならない。

例題 4-1　のプログラム例

```
1   program example4_01
2       implicit none
3       real    :: x(10)
4       real    :: average, variance
5       integer :: n, i
6
7       average = 0.0
8       read(*,*) n
9       do i = 1, n
10        read(*,*) x(i)
11        average = average + x(i)
12      enddo
13      average = average/real(n)
14
15      variance = 0.0
16      do i = 1, n
17        variance = variance + (x(i)-average)**2
18      enddo
19      variance = variance/real(n)
20
21      write(*,*) "average=", average, " variance=", variance
22
23      stop
24  end program example4_01
```

第 3 行が配列宣言文であり、ここでは寸法 10 の配列変数を宣言している。すなわち、添え字は 1 から 10 までの整数値でなければならない。

配列宣言の一般形

例題 4-2

あるクラス *n* 名にアンケートを採り、「非常によい」「よい」「ふつう」「悪い」「非常に悪い」という評価を、それぞれ 2,1,0,-1,-2 点として記入させた。その結果をもとに、まず先頭の行に *n*、その後 1 行に結果を 1 つずつ、*n* 行入力したファイルを作成した。
このファイルを読み込んで、それぞれの評点の度数分布と、評点の平均を求めるプログラムを作成しなさい。

select case 文で分岐させてもよいが、この場合評点が連続した整数値なので、評点を添え字と一致させてしまった方が簡明である。添え字の範囲は-2 から 2 までとなるが、そのような場合の寸法宣言子の一般形は以下のように書く。

（寸法宣言子）→（添え字の下限:添え字の上限）

寸法は「添え字の上限 – 添え字の下限＋1」となる。添え字の下限が 1 の場合は、「添え字の下限:」の部分を省略できて、前述した形となる。なお、上限、下限値には定数だけでなく、**名前付き定数およびその式も使用できる**。ただし宣言時に値が確定していなければならない。

例題 4-2 のプログラム例

```
1   program example4_02
2       implicit none
3       integer :: k(-2:2)
4       integer :: i, j, n
5       real    :: average
6
7       do i = -2, 2
8         k(i) = 0
9       enddo
10      average = 0.0
11
12      read(*,*) n
13      do i = 1, n
14        read(*,*) j
15        k(j) = k(j) + 1
16        average= average + real(j)
17      enddo
18
19      do i = -2, 2
20        write(*,*) i, " ->",k(i)
21      enddo
22      write(*,*) "average=", average/real(n)
23
24      stop
25  end program example4_02
```

第 3 行が寸法宣言子の一般形であり、添え字-2 から 2 までの配列を宣言している。

2. 配列の入出力

配列全体の入出力

例題 4-3

10 人の試験の点数を、区切り子 (「␣」か「,」) で区切って入力し、1 行に収まらない場合は適宜改行を入れることにします。そのデータから、最高点、最低点、平均点を算出するプログラムを作成しなさい。ただし、試験の点数は 0 以上 100 以下の整数値とします。

配列の要素を、その下限から上限までをすべて標準入力から読み込むとき、およびその下限から上限までをすべて標準出力へ書き出すときは、以下のように記述する。

```
read(*,*) 配列名

write(*,*) 配列名
```

たとえば、x が下限 1、上限 5 の配列ならば、

```
read(*,*) x
```

という文は、次の文と同値である。

```
read(*,*) x(1), x(2), x(3), x(4), x(5)
```

例題 4-3 のプログラム例

```
1   program example4_03
2       implicit none
3       integer :: ip(10), max_point, min_point
4       real    :: average
5       integer :: i
6
7       read(*,*) ip
8
9       max_point = 0
10      min_point = 100
11      average = 0.0
12      do i = 1, 10
13        if(ip(i) > max_point) max_point = ip(i)
14        if(ip(i) < min_point) min_point = ip(i)
15        average = average + real(ip(i))
16      enddo
17
18      write(*,*) "max point=",max_point
19      write(*,*) "min point=",min_point
20      write(*,*) "average=  ", average/10.0
21
22      stop
23  end program example4_03
```

最小値を値の範囲の最大に、最大値を値の範囲の最小に初期化し、第 13 行、第 14 行で、それまでの最大値より大きい数、それまでの最小値より小さい数があれば、最大値、最小値を入れ替えている。

入出力 do 並び

例題 4-4

> **例題 4-1** で 1 行に 1 つずつデータを入れると、n が大きくなると入力ファイルの行数が増えて見にくくなる。これを改善するため、区切り子でデータを区切って入力し、1 行に収まらない場合は適宜改行を入れることにする。そのようにプログラムを変更しなさい。

配列要素の 1 部だけを入出力するために、**入出力 do 並び** が使われる。標準入力からの入力の場合ならば、以下の形式である。

```
read(*,*)（配列名（制御変数）, 制御変数=添え字の初期値, 添え字の終了値, 添え字の増分）
```

増分が 1 ならば、「, 添え字の増分」の部分は省略できる。この並びにより、do 構文と同様に、
(i) 初期値
(ii) 初期値＋増分
(iii) 初期値＋増分 × 2
⋮
が終了値を超えるまで次々に計算され、その値を添え字とする配列要素に格納される。

出力の場合も同様に、do 並びで指定された配列要素から、続いて出力される。

例題 4-4 のプログラム例

```
1  program example4_04
2      implicit none
3      real    :: x(10)
4      real    :: average, variance
5      integer :: n, i
6
7      average = 0.0
8      read(*,*) n, (x(i), i = 1, n)
9
10     do i = 1, n
11       average = average + x(i)
12     enddo
13     average = average/real(n)
14
15     variance = 0.0
16     do i = 1, n
17       variance = variance + (x(i)-average)**2
18     enddo
19     variance = variance/real(n)
20     write(*,*) "average=", average, " variance=", variance
21
22     stop
23 end program example4_04
```

第 8 行、n がデータと同じ行にあっても、入出力 do 並びに至るまでにその値が確定していればよい。

3. 配列構成子

配列構成子の基本形

例題 4-5

海外のメーカーのロガーでは、記録日のデータとして月と日を記録する代わりに、1 月 1 日から積算した暦年通算日を記録することがある。月と日を入力して、暦年通算日を出力するプログラムを作成しなさい。ただし、閏年ではない平年とします。

変数に対する定数と同様に、配列変数に対する配列定数にあたるものが、**配列構成子** である。寸法 n の配列構成子は、以下のように配列要素の値を「,」で区切り、前後を「(/」と「/)」でくくって定義する。

```
(/ 1 番目の配列要素の値, 2 番目の配列要素の値, ··· , n 番目の配列要素の値 /)
```

最初の「(/」と最後の「/)」は、2 文字で一続きの構文素であり、間に空白を入れてはならない。

例題 4-5 のプログラム例

```
1   program example4_05
2       implicit none
3       integer :: month_day(12)=(/31,28,31,30,31,30,31,31,30,31,30,31/)
4       integer :: month, day, day_year, i
5
6       read(*,*) month, day
7
8       day_year = 0
9       do i = 1, month-1
10        day_year = day_year + month_day(i)
11      enddo
12      day_year = day_year + day
13
14      write(*,*) "day of year=", day_year
15
16      stop
17  end program example4_05
```

第 3 行の初期化式で、配列変数に配列構成子を代入している。このとき、**左辺と右辺の寸法は一致していなければならない**。

配列構成子の一般形

配列構成子の一般形は、以下のようになる。

```
(/ 1 番目の配列構成値, 2 番目の配列構成値, ···  /)
```

配列構成値は、配列構成子の要素であり、全て同じ型でなければならない。文字型の場合は、文字長が全て等しくなければならない。その配列構成値の型が、配列構成子の型となる。

配列構成子は、代入文等の実行文でも使われるが、初期化式で使われる場合が多い。この両者で、配列構成値に使える要素が多少異なる。

実行文での配列構成値

それぞれの配列構成値は、次のうちのいずれかである。

(i) スカラー値 (定数、名前付き定数、変数) の並び。混在も可、ただしすべて同じ型でなければならない。

`(/ 2.2, x, 1.2 /)` ··· ただし、x は実数型

(ii) 「( )」でくくった do 型並び。

`(/ (i, i=1,9,2) /)` ··· 寸法は 5 になる。整数型変数 i が宣言されていなければならない。

(iii) 配列あるいはその一部。

`(/ a, b /)` ··· ただし、a、b は一次元の配列。

初期化式での配列構成値

初期化式で使う場合は、その文がコンパイルされる時点で、使用されるすべての変数が宣言され、do 制御変数以外は値が確定していなければならない。そのため、配列構成値に使える要素に、以下のような制約が生じる。

(i′) スカラー値 (定数、名前付き定数) の並び。混在も可、ただしすべて同じ型でなければならない。また、名前付き定数はこれより前に定義されていなければならない。

```
real, parameter :: x = 1.7
real            :: a(3) = (/ 2.2, x, 1.2 /)
```

(ii′) 「( )」でくくった do 型並び。ただし、制御変数はこれより前に整数型変数として宣言されていなければならない。

```
integer :: i
integer :: k(5) = (/ (i, i=1,9,2) /)
```

(iii′) 定数配列および名前付き定数配列、あるいはその一部。

```
integer, parameter :: ic(2) = (/ 1, 2 /)
integer            :: id(4) = (/ ic, (/ 3, 4 /) /)
```

演 習 問 題

演習問題 1

例題 4-5 で、逆に暦年通算日を入力して、月、日を出力するプログラムを作成しなさい。

解答例

```
program example4_05a
    implicit none
    integer :: month_day(12)=(/31,28,31,30,31,30,31,31,30,31,30,31/)
    integer :: month, day

    read(*,*) day
    do month = 1, 12
      if( day <= month_day(month) ) exit
      day = day - month_day(month)
    enddo

    write(*,*) "month=", month, " day=", day

    stop
end program example4_05a
```

4. data 文

例題 4-6

C、H、O より構成された化合物の一群がある。これらの化合物の炭素数、水素数、酸素数をこの順に入力して、その分子量を求めるプログラムを作成しなさい。ただし、原子量は C= 12、H= 1、O= 16 とします。

変数に値を格納する方法としてこれまでに、
(i) 代入文や read 文などの実行文で、実行時に格納する方法
(ii) 宣言文の初期化式で、コンパイル時に格納する方法
の二つを示した。これらに加え、第 3 の方法として、
(iii) data 文で、コンパイル時に格納する方法
があり、主に配列の初期値の代入に使われる。[*1] data 文の形式は以下の形である。

```
data 変数名の並び１ / 定数の並び１ /, 変数名の並び２ / 定数の並び２ /
```

変数名の並びは、スカラー変数、配列名、及び配列の一部分を「,」で区切った並びである。定数の並びは、同じく定数を「,」で区切ったものである。変数名の並びに含まれる、スカラー変数の数と配列要素の数の和は、定数の並びの数と等しくなければならない。変数の並びと定数の並びは並び順に一対一に対応し、定数の値が対応する変数に格納される。

変数名の並びには、配列の一部として、(a(i),i=5,10) のような形式が可能で、配列の一部だけを初期化することができる。これが初期化式による配列の初期化との大きな違いである。これにより、大きな配列に初期値を与える場合、配列をいくつかに分割し、それぞれ data 文で初期化することができる。

定数の並びに同じ数字が続く場合、たとえば「··· 1,2,0,0,0,0,3 ···」のようなときは、「繰り返し回数*」を使って、「··· 1,2,4*0,3 ···」と書くことができる。

data 文を置く位置は、宣言文のあとで end 文の前ならどこでもよいが、通常は宣言文の直後に置く。

例題 4-6 のプログラム例

```
1   program example4_06
2       implicit none
3       real    :: w(3), w_mole
4       integer :: k(3), i
5       data w/12.0, 1.0, 16.0/
6
7       read(*,*) k
8       w_mole = 0.0
9       do i = 1, 3
10        w_mole = w_mole + w(i)*real(k(i))
11      enddo
12      write(*,*) "molecular weight=", w_mole
13
14      stop
15  end program example4_06
```

[*1] スカラー変数の初期化にも使えるが、その場合は初期化式の方がわかりやすい。

§4.3 種々の配列

1. 多次元配列

多次元配列の宣言

例題 4-7

A 君の英語、数学、世界史の点数は、順に 82 点、75 点、78 点、B 君は 65 点、90 点、70 点、C 君は 70 点、73 点、85 点、D 君は 66 点、72 点、71 点であった。
これらの点数を、1 行に 1 人分 3 個の点数を区切り子で区切って入力する。それを 4 行 4 人分繰り返す。これから教科別の平均点を出力するプログラムを作成しなさい。

Excel などの表計算ソフトでよく遭遇する問題である。個人別の平均点ならば 1 行ずつ処理すればよいが、教科別となるといったん全部の点数を二次元配列に格納しておかなければならない。その宣言は、

型名 :: 配列名（第 1 次元の寸法宣言子, 第 2 次元の寸法宣言子）

という形式である。ここで問題のデータは、下左のような 4 行 3 列の形である。行列や表計算ソフトでは、行が第 1 次元、列が第 2 次元にあたるので、第 1 次元が 4、第 2 次元が 3 である配列を宣言して、点数を格納すれば、下右のようにその配列要素が対応する。

```
82,75,78              ip(1,1) ip(1,2) ip(1,3)
65,90,70      ->      ip(2,1) ip(2,2) ip(2,3)
70,73,85              ip(3,1) ip(3,2) ip(3,3)
66,72,71              ip(4,1) ip(4,2) ip(4,3)
```

例題 4-7 のプログラム例

```
1  program example4_07
2      implicit none
3      integer :: ip(4,3)
4      real    :: ave_subject(3)
5      integer :: i, j
6
7      do i = 1, 4
8        read(*,*) (ip(i,j), j= 1, 3)
9      enddo
10
11     do j = 1, 3
12       ave_subject(j) = 0.0
13       do i = 1, 4
14         ave_subject(j) = ave_subject(j) + ip(i,j)
15       enddo
16       ave_subject(j) = ave_subject(j)/4.0
17     enddo
18
19     write(*,*) (ave_subject(j), j = 1, 3)
20
21     stop
22 end program example4_07
```

配列要素順序

二次元配列はイメージ的には行列、あるいは表のように扱われるが、メモリ空間は一次元なので、そのままの形では格納できない。何らかの方法により一次元に変形して格納しなければならない。

Fortran では、両次元とも下限の配列要素を第 1 番目とするとき、まず第 1 列について、行の順に格納していく。上の行列では、左端の列を縦に格納していくことになる。行の上限 (行列の下端) に達したら第 2 列にうつり、同じく下限の行から上限の行まで格納する。このように最後の列まで格納していく。[*2]

たとえば前の例題の、3 行 4 列の 2 次元配列「ip」は、次の順で格納される。[*3]

```
ip(1,1), ip(2,1), ip(3,1), ip(4,1), ip(1,2),…… , ip(3,3), ip(4,3)
```

このような格納順を、**配列要素順序** という。

入出力文で、配列要素ではなく、単に配列名で、

「read(*,*) ip」　あるいは　「write(*,*) ip」

と指定すれば、配列要素順序にしたがって入出力される。

二次元配列の初期化

Fortran では、二次元以上の配列に直接初期値を入れる手段がないので、いったん配列要素を順序にしたがって並べた配列構成子を作成してから、二次元配列あるいは多次元配列に成形する。その方法は二通りある。

(1) data 文による方法

宣言文で配列宣言をしておき、data 文で初期値を与える、配列全体に初期値を与えるならば、右辺に配列要素順序通りに配列要素の値を並べればよい。例題のデータを data 文で初期化すると、

```
integer :: ip(4,3)
        |
data ip/ 82, 65, 70, 66, 75, 90, 73, 72, 78, 70, 85, 71 /
```

あるいは、do 型並びにより並びの順序を入れ替えて、

```
data ((ip(i,j),j=1,3),i=1,4) / 82, 75, 78, 65, 90, 70, 70, 73, 85, 66, 72, 71 /
```

(2) reshape 関数による方法

reshape 関数の基本形は、以下の形式である。

```
reshape( source, shape )
```

ここで source は変形すべき元の配列である。ここでは配列要素順序にしたがって、配列要素を並べる。

shape は一次元の整数型配列で、(/ 行数、列数 /) の順に数値を指定して、目的の配列の形状を指定する。戻り値は、(行数) 行、(列数) 列の配列となる。

この形式の初期化は、「ip(4,3) =」という形で宣言文の初期化式としてコンパイル時に使われるだけでなく、実行文として実行時にも有効である。

たとえば、例題の場合は以下のように指定する。

```
ip = reshape((/ 82, 65, 70, 66, 75, 90, 73, 72, 78, 70, 85, 71 /), (/ 4, 3 /))
```

[*2] C では逆に、列から変えて格納する。

[*3] 実は規格上では、実メモリ空間にこの順で配置されるとは規定していないのであるが、ユーザはこのように配置されているとしてプログラムできる … はず。

2. 部分配列

例題 4-8

ある実験担当の講師は、受講生の人数から判断して、学籍番号順に 3 班に分けて実験を行うことにした。そこで受講生ファイルから学籍番号を抜き出してテキスト・ファイルを作成したところ、

　1,2,4,7,8,10, ⋯

と、不連続な番号であり、全体の人数は *n* 人であった。
このファイルを標準入力から読み込んで、3 班の班員の学籍番号を出力するプログラムを作成しなさい。ただし班分けは学籍番号の小さい順から 1 班は 1,4,7,⋯、2 班は 2,5,8,⋯、3 班は 3,6,9,⋯ 番目に分けていくとします。

このプログラムは do 構造を使えば実現できるが、部分配列を使えばより簡明に記述することができる。
部分配列は配列の一部を取り出して新たな配列としたものであり、その一般形は元の配列を a として、

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

と表せる。この表現を**添え字三つ組** という。ここで n1 は、部分配列の先頭となる要素の、配列 a での添え字、n2 は、部分配列の最後となる要素の、配列 a での添え字、n3 は部分配列を抜き出す間隔であり、n3=1 のときに連続して抜き出す。n3≠1 のときは、do 構文と同じく、部分配列の最後の a での添え字が、n2 と一致しないこともある。

また、次のような場合は、n1、n2、n3 を省略することができる。
(i) n1=1、すなわち先頭から抜き出す場合は、「1」を省略できる。
(ii) n2 が配列 a の寸法、すなわち最後まで抜き出す場合は、「n2」を省略できる。
(iii) 間隔 n3 が 1、すなわち連続して抜き出す場合は、「:1」を省略できる。このとき必ずセミコロン「:」も省略する。「1」だけを省略することはできない。

部分配列に対する元の配列を**全体配列** という。この規則に従えば、部分配列「a(:)」は。全体配列に等しくなる。

例題 4-8 のプログラム例

```
1   program example4_08
2       implicit none
3       integer, parameter :: m = 14
4       integer :: n(m)
5
6       read(*,*) n
7
8       write(*,*) n(1:m:3) ! n(::3) と書いてもよいが、わかりにくい。
9       write(*,*) n(2:m:3)
10      write(*,*) n(3:m:3)
11
12      stop
13  end program example4_08
```

配列の一部を出力する方法としては、前述の do 型並びによる出力があるが、添え字三つ組みによる部分配列の出力の方が、制御変数を必要としない分、簡明である。

多次元の部分配列

多次元配列の部分配列は、各次元について「(n1,n2[,n3])」の形式で範囲を指定でき、その規則は (i)、(ii)、(iii) をみたす。たとえば全体配列 a(4,3)

```
a(1,1) a(1,2) a(1,3)
a(2,1) a(2,2) a(2,3)
a(3,1) a(3,2) a(3,3)
a(4,1) a(4,2) a(4,3)
```

に対して、部分配列 a(2:4:2,:) は、

```
a(2,1) a(2,2) a(2,3)
a(4,1) a(4,2) a(4,3)
```

という形式で抽出される。

また、いずれかの次元で整数式により添え字が指定されているとき、たとえば部分配列 a(3,:) は、

```
a(3,1) a(3,2) a(3,3)
```

となり、**配列要素が指定された次元の数だけ次数が下がる**。

注意すべきことは、添え字 a(n1) と範囲 a(n1:n1) とは意味が異なることである。前者は添え字を指定しているため次元が 1 つ下がるが、後者は下がらない。たとえば、「a(n1,n2)」は、2 個の添え字が指定されているので、配列の次元はゼロ、すなわちスカラーである。一方「a(n1:n1,n2:n2)」は次元の寸法がそれぞれ 1 である二次元配列である。したがって「a(2,2) = a(2:2,2:2)」という代入文は、コンパイル・エラーとなる。

ゼロ配列

部分配列 a(n1,n2) で n1>n2 になった場合には、そのような配列は存在しないが、エラーにはならずに大きさ (size) ゼロの配列になる。このような配列をゼロ配列という、

多次元配列で、一つの次元の寸法がゼロの場合、大きさはそれぞれの次元の寸法のかけ算であるから、全体のサイズもゼロになる。

ベクトル添え字

配列の添え字が、またある配列の配列要素であることがある。

あるクラス n 人の試験の成績が、学籍番号順に配列 a に格納されていて、その中から推薦を希望する者の成績だけを抜き出して出力するという作業を考える。ただし、推薦を希望する者は m 人であり、その学籍番号は整数型配列 num に格納されているとする。

このとき、配列 a の配列要素を整数型配列 num に形式的に置き換えた、

```
a(num)
```

は、a(num(1)),a(num(2))...a(num(m)) を配列要素とする、寸法 m の a の部分配列を表す。このような配列を**ベクトル添え字** による部分配列という。これによりプログラムの出力部分は、

```
write(*,*) (a(num(i)), i=1, m)
```

と do 型出力並びで書く代わりに、以下のように部分配列の形式で書くことができる。

```
write(*,*) a(num(:))
```

num による添え字指定には重複が許される。したがって、num の寸法が a の寸法よりも大きいこともありうる。その場合、部分配列と言っても寸法が元の全体配列の寸法よりも大きくなる。ただし、read 文の入力並び、および配列代入文の左辺では、重複した添え字を指定することはできない。

3. 割付け配列

例題 4-9

n 個の実数 x を入力し、その標本平均を出力するプログラムを、配列を使って作成しなさい。ただし、n は最初に読み込むものとし、その数はあらかじめ知られていないものとします。

テストデータ

```
5
81, 72, 70, 64, 90
```

この例題は**例題 4-1** を改変したもので、その相違は、n の上限があらかじめ知られていないことにある。

このようなプログラムを書く方法は二つある、その一つは考えうる範囲で十分大きな配列を宣言することである (この場合なら `x(10000)` など)。しかし、この方法は次のような問題を含んでいる。
(i) データ数が非常に大きい場合、大量のメモリー空間をムダに消費する可能性が高い。
(ii) 万一、データ数が配列の大きさを超えた場合、正常動作しない。特に非デバッグモードで動かした場合、実行時エラーが出ないこともあり、その場合誤った計算結果を信用してしまうという危険性がある。
(iii) 全体配列の一部分しか計算に使わないため、それぞれの計算でいちいち計算範囲 n を指定しなければならない。これはプログラムが冗長になることに加え、コーディングミスの可能性を増加させる。

これに対し、第二の方法は実行文中で大きさ n の配列を確保する方法である。これを配列の**動的割付け**といい、その配列のことを**割付け配列** という。割付け配列を利用することにより、上記のような問題は回避されるが、その利用に際してはいくつかの手続きを踏まねばならない。

step1: 型名に `allocatable` 属性指定子をつけた上で、寸法宣言子を指定せずに、宣言文で配列宣言を行う。

```
型名, allocatable :: 配列名 (:)
```

上は一次元の配列の場合で、二次元配列ならば、**配列名 (:,:)** とする。それ以上の次元も同様。

step2: `allocate` 文により、寸法宣言子を指定して、配列を動的に割り当てる。下は一次元配列の場合。

```
allocate (配列名 (寸法宣言子))
```

通常の配列宣言と異なるのは、配列の上限値、下限値を指定する整数値が、整数定数ではなく整数変数や整数式で可能なことである。上の例題ならば、n を上限値として寸法宣言できる。

`allocate` 文と、次の `deallocate` 文の間では、割付け配列は通常の配列と同様に使用できる。

step3: `deallocate` 文により、配列の割り当てを解除し、メモリを解放する。

```
deallocate (配列名)
```

`deallocate` 文には寸法宣言子は不要である。

`deallocate` 文の実行後は、もはやその配列は使用できない。どうしても使いたければ、再び `allocate` 文で割当てなければならない。その場合、寸法が以前と同じである必要はない。

例題 4-9　のプログラム例

```
1   program example4_09
2       implicit none
3       real, allocatable :: x(:)
4       real    :: sum
5       integer :: n, i
6
7       read(*,*) n
8       allocate (x(n))
9       read(*,*) x(:)
10      sum = 0.0
11      do i = 1, n
12        sum = sum + x(i)
13      enddo
14      write(*,*) "average =", sum/real(n)
15
16      deallocate (x)
17      stop
18  end program example4_09
```

第 3 行が割付け配列の宣言、第 8 行が割付け、第 16 行が配列領域の解放である。第 9 行は配列要素の読み込みであり、全体配列の場合は do 並びは不要である。また、第 9 行は単に、

```
read(*,*) x
```

と書くこともできるが、それでは x が配列でなく単なる変数であるかのように誤解されるおそれがあるので、このように書いて配列であることを明示している。

演 習 問 題

演習問題 2

例題 4-7 で、人数が 4 人ではなく n 人であり、n は先頭行から読み込むように変更しなさい。

解答例

```
program example4_09a
    implicit none
    integer, allocatable :: ip(:,:)
    real    :: average
    integer :: i, j, n

    read(*,*) n;     allocate (ip(n,3))
    read(*,*) ((ip(i,j), j =1, 3), i =1, n)
    do j = 1, 3
      average = 0.0
      do i = 1, n
        average = average + ip(i,j)
      enddo
    write(*,*) average/real(n)
    enddo
    deallocate (ip)
    stop
end program example4_09a
```

§4.4 配列演算

配列を配列要素の並びとして見て、その配列要素ごとに一度に同じ計算を行うものが配列演算である。

1. 数値演算子による演算

例題 4-10

2 個のベクトル、$\vec{a} = \begin{pmatrix} 2.0 \\ 1.5 \\ 3.2 \end{pmatrix}$ および $\vec{b} = \begin{pmatrix} 1.2 \\ -1.1 \\ 0.6 \end{pmatrix}$ について、

(1) $\vec{a} + \vec{b}$

(2) $2\vec{a} - 3\vec{b}$

(3) $\vec{a} \cdot \vec{b}$ (内積)

を、それぞれ計算しなさい。

n 次のベクトルは、Fortran では寸法 n の 1 次元配列に置き換えられる。そこで寸法 3 の 1 次元配列 a、b を宣言して、data 文あるいは read 文で配列要素の値を書き込んでから、計算を行う。

四則計算

配列 a と配列 b の、対応する配列要素ごとの和、差、積、商を計算して、結果を配列 c に代入するには、

```
(和)  c = a + b    (差)  c = a - b    (積)  c = a * b    (商)  c = a / b
```

と、あたかもスカラー同士の四則計算であるように記述する。[*4] これにより、たとえば和の場合なら、それぞれ対応する配列要素について「c(i) = a(i) + b(i)」が計算されて代入される。多次元の場合も同様である。

· 配列の加算と減算については、線形代数の行列和、行列差と定義が同一である。

· 配列の乗算は、線形代数の行列積とは、定義が異なる。

· 配列の除算については、対応する線形代数の演算はない。

· 計算が行われるためには、a、b、c が**形状適合**でなければならない。すなわち次元の数が等しく、かつすべての次元での寸法が一致していなければならない。

· ただし、型の混在はスカラー演算と同様に許される。

スカラー倍

配列 x のすべての配列要素を a 倍して、結果を配列 y に代入するには、

```
y = a * x あるいは、y = x * a
```

と、やはりスカラー同士の四則計算であるように記述する。

· これは行列のスカラー倍と定義が同一である。

· 計算が行われるためには、x と y は形状適合でなければならない。

· ただし、スカラーはすべての配列と形状適合するので、配列に直す必要はない。

[*4] ただし、c(:) = a(:) + b(:)、あるいは c(:,:) = a(:,:) + b(:,:) (2 次元) などと書いた方が、スカラー変数と混同するおそれは減少する。

べき乗

配列 d のすべての配列要素の e 乗を、結果を配列 f に代入するには、

```
f = d ** e
```

と、やはりスカラー同士の四則計算であるように記述する。

· あとは、スカラー倍と同様である。

例題 4-10　のプログラム例

```
1   program example4_10
2       implicit none
3       real    :: a(3)=(/2.0,1.5,3.2/), b(3)=(/1.2,-1.1,0.6/)
4       real    :: c(3), sum
5       integer :: i
6
7       write(*,*) a(:) + b(:)
8       write(*,*) 2.0*a(:) - 3.0*b(:)
9       c(:) = a(:) * b(:)
10      sum = 0.0
11      do i = 1, 3
12        sum = sum + c(i)
13      enddo
14      write(*,*) sum
15
16      stop
17  end program example4_10
```

内積については、後述の配列関数により、より効率よく計算することができる。

演習問題

演習問題 3

行列 $\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$ と 行列 $\begin{pmatrix} 2.5 & 1.0 \\ 0.5 & -1.5 \end{pmatrix}$ の和を計算しなさい。

解答例

```
program example4_10a
    implicit none
    real :: a(2,2), b(2,2), c(2,2)
    data a/1.0,3.0,2.0,4.0/   ! 順序に注意
    data b/2.5,0.5,1.0,-1.5/  ! 同上

    c(:,:) = a(:,:) + b(:,:)
    write(*,*) c(1,1), c(1,2)
    write(*,*) c(2,1), c(2,2)

    stop
end program example4_10a
```

2. 要素別処理関数

例題 4-11

ベクトル、$\vec{a} = \begin{pmatrix} 1.2 \\ 0.5 \\ -0.9 \end{pmatrix}$ の各要素をそれぞれ四捨五入した整数を要素とする、ベクトル $\vec{b}$ を作成して出力しなさい。

これは do 構文を使って、

```
real    :: a(3)=(/1.5,0.5,-0.9/)
integer :: b(3), i
do i=1,3
  b(i) = nint((a(i))
enddo
```

と書くことができるが、ある配列 1 のすべての配列要素が、同一の組み込み関数 (この場合は `nint`) の引数になっているならば、

```
配列２ = 関数名（配列１）
```

のように簡明に記述できることがある。

· 配列 1 と配列 2 は形状適合でなければならない。

· 配列のうちの一方あるいは両方が、部分配列でも可である (ただし形状適合すれば)。

このように配列の各要素を引数として、それぞれの関数値を要素とする配列を戻り値とする組み込み関数のことを、**要素別処理関数** という。特定の組み込み関数が要素別処理関数であるかどうかは、文法書に記述されているが、一般に次のような規則がある。

· すべての数値関数は要素別処理関数である。

· すべての数学関数は要素別処理関数である。

· 文字列操作関数の大部分は要素別処理関数である。

· 配列関数は要素別処理関数でない。

· その他、配列を引数にとらない関数は、そもそも要素別処理関数になり得ない。

例題 4-11 のプログラム例

```
1   program example4_11
2       implicit none
3       real    :: a(3)=(/1.5,0.5,-0.9/)
4       integer :: b(3)
5
6       b(:) = nint(a(:))
7       write(*,*) b(:)
8
9       stop
10  end program example4_11
```

第 6 行で配列要素ごとに処理を行っている。

§4.5 配列制御構文

1. where 文と where 構文

where 文および where 構文は、Fortran90 より導入された、スカラー変数の if 文に対応した配列の分岐処理のための文である。配列の各配列要素に対して、条件分岐により、それぞれ配列計算を行う。

例題 4-12

大きさ n の一次元実数型配列 a を入力し、各配列要素について負の値であれば 0.0 にそれぞれ置き換えた配列を作成して出力しなさい。ただし、n はあらかじめ与えられているものとします。

テストデータ

```
a = (/2.5,1.2,-1.2,1.1,-0.5/)
```

where 文

where 文は、論理 if 文に対応した配列の分岐処理文であり、以下の形式である。

```
where(選別式|mask 配列) 配列代入文
```

where の次の () の中には、配列代入文と形状適合である、論理型の配列が入る。この配列の配列要素の値が真である場合のみ、対応する配列要素に対して、配列代入文が実行される。

論理型の配列としては、以下のどちらかを使う。

· 論理値「.true.(真)」「.false.(偽)」をとる配列要素からなる論理型配列 (mask 配列)
· 関係演算子のスカラーを配列に置き換えた形式の**選別式** (例：a(:) < 1.0)。

where 構文 (ブロック where 文)

where 構文は、ブロック if 文に対応した配列の分岐処理文であり、以下の形式である。[*5]

```
where(選別式)
  配列代入文 1
  配列代入文 2
      :
end where
```

処理は、選別式、配列代入文 1、配列代入文 2、… の順に行われる。すなわち配列代入文 1 によって配列要素の値が変化しても、それは選別式の評価に影響せず、配列代入文 2 以下の実行は、最初の選別式の真偽によって行われる。

例題 4-12　のプログラム例

```
1  program example4_12a
2      implicit none
3      real    :: a(5)=(/2.5,1.2,-1.2,1.1,-0.5/)
4
5      where( a(:)< 0.0 ) a(:) = 0.0
```

[*5] 多くのコンパイラでは、elesewhere(選別式)、elsewhere の構文が追加されていて、それぞれブロック if 文での elseif、else に対応した処理を行うことができる。

```
6       write(*,*) a(:)
7
8       stop
9   end program example4_12a
```

2. forall 文と forall 構文 (95)

forall 文は、Fortran95 より導入された、where 文と do 文とをあわせた機能を持つ。すなわち、where 文の配列式ではなく配列要素ごとに処理を指定できる文である。 やはり forall 文と forall 構文があり、それぞれ以下の形式である。

forall 文

```
forall(do 制御変数 1 の三つ組 [、do 制御変数 2 の三つ組]、..[、選別式]) 配列要素の代入文
```

ここで、do 変数の三つ組みは、以下の形式である。

```
制御変数=初期値:終了値:増分
```

《例》 `forall(i=1:n, a(i)>0) a(i)=1.0/a(i)`

forall 構文

```
forall(do 制御変数 1 の三つ組 [、do 制御変数 2 の三つ組]、..[、選別式])
  配列要素の代入文 1
  配列要素の代入文 2
      :
end forall
```

do 文による loop と異なるのは、do 文の loop は制御変数の順に実行されるが、forall 文では配列式のように、すべての配列要素について同時に処理されることである。このためマルチ CPU やベクトル・プロセッサによる並行処理には、forall 文の方が有利である。

例題 4-12 のプログラム例

```
1   program example4_12b
2       implicit none
3       real    :: a(5)=(/2.5,1.2,-1.2,1.1,-0.5/)
4       integer :: i
5
6       forall(i=1:5, a(i)<0.0) a(i) = 0.0
7       write(*,*) a(:)
8
9       stop
10  end program example4_12b
```

§4.6 組み込み配列関数 (90/95)

配列を実引数とする関数を、配列関数という。fortran90/95 で大いに機能強化された部分である。多くの種類があるが、その分類方法は書籍によってさまざまであり、以下の分類は普遍的なものではない。

1. 線形代数計算

1 次元および 2 次元の配列は、ベクトル・行列に対応して、数学、物理、統計方面で使用されることが多いため、Fortran(90/95) ではこれらの線形代数としての計算のために、特別に以下の 3 個の関数が用意されている。これらの関数は、特定の次元、形状に対してのみ使えるという特徴がある。

例題 4-13

(1) 2 個のベクトル、$\vec{a} = \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$ および $\vec{b} = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}$ について、内積 $\vec{a} \cdot \vec{b}$ を計算しなさい。

(2) 行列 $\mathbf{C} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$ について、$\mathbf{C}^2$ および $\mathbf{C}^t\mathbf{C}$ を計算しなさい ($\mathbf{C}^t$ は $\mathbf{C}$ の転置行列)。

ベクトルの内積 (dot_product)

○ ベクトル **u** と、ベクトル **v** の内積、**(u,v)** を計算する組み込み関数である。その形式は以下の通り、

```
dot_product(u,v)
```

ここで「u」、「v」は 1 次元の配列であり、その大きさは等しくなければならない。「u」、「v」の配列要素が、ベクトル **u**、**v** の要素にそれぞれ対応する。

·「u」、「v」は、整数型、実数型、あるいは複素数型の数値配列である。「u」、「v」の型や種別は異なってもよいが、それぞれの配列の配列要素同士は全て同じでなければならない。

· 戻り値はスカラーであり、その型と種別は「u」、「v」の要素同士の混合演算の規則による。

·「u」が複素数型のときは、**u** の複素共役 (conjugate) ベクトルとの積和 $\sum_i \mathrm{u(i)}^*, \mathrm{v(i)}$ が計算される。

○「u」、「v」が、ともに大きさが同じ論理型配列でもよい。この場合、それぞれの対応するする成分同士の論理積「u(i) .and. v(i)」がそれぞれ計算され、「真」が一つでもあれば戻り値は「真」になる。「真」が一つもなければ、戻り値は「偽」である。

○ 大きさがともにゼロのときは、数値型の配列の場合戻り値は「0」、論理型の場合「偽」になる。

例題 4-13 のプログラム例

```
1   program example4_13a
2   ! ベクトルの内積
3       implicit none
4       integer :: a(3) = (/2, 1, 3/)
5       integer :: b(3) = (/1,-1, 2/)
6       integer :: p
7
8       p = dot_product(a,b)
9       write(*,*) p
10
11      stop
12  end program example4_13a
```

行列の積 (matmul)

○ 行列 **A** と行列 **B** の行列積、**C=AB** を計算する組み込み関数である。その形式は以下の通り、

```
c = matmul(a,b)
```

ここで「a」が k 行 m 列の 2 次元の配列であるとすると、行列積が計算できるために、「b」は m 行 n 列の 2 次元の配列でなければならない。そしてその結果を受け取る配列「c」は k 行 n 列の 2 次元の配列でなければならない。それぞれの配列要素が、行列の要素に対応する。

· 「a」、「b」は、整数型、実数型、あるいは複素数型の数値配列である。「a」、「b」の型や種別は異なってもよいが、それぞれの配列の配列要素同士は全て同じでなければならない。

· 戻り値の型と種別は「a」、「b」の要素同士の混合演算の規則による。

· 戻り値の配列の配列要素を直接参照することはできない。同型の配列で受け取ってから参照しなければならない。

○ 2 次元配列「b」の代わりに大きさ m の 1 次元配列「u」を引数にとり、**v=Au** を計算できる。

```
v = matmul(a,u)
```

戻り値「v」は、大きさ k の 1 次元配列になる。

○ 2 次元配列「a」の代わりに大きさ k の 1 次元配列「v」を引数にとり、$\mathbf{u} = {}^t\mathbf{v}\mathbf{A}$ を計算できる。

```
v = matmul(v,a)
```

戻り値「u」は、大きさ m の 1 次元配列になる。

転置行列 (transpose)

○ 行列 **A** の転置行列[*6] $\mathbf{C} = {}^t\mathbf{A}$ を計算する組み込み関数である。その形式は以下の通り、

```
c = transpose(a)
```

· ここで「a」が k 行 m 列の 2 次元の配列であるとすると、戻り値は m 行 k 列の 2 次元の配列となる。

· やはり、戻り値の配列の配列要素を直接参照することはできないので、形状適合する配列を宣言した上で戻り値を代入して、参照しなければならない。

例題 4-13 のプログラム例

```
1   program example4_13b
2   !  行列の積
3       implicit none
4       integer :: c(2,2), d(2,2), e(2,2)
5       integer :: i, j
6       data c/1,0,2,1/
7
8       d = matmul(c,c)
9       write(*,*) d(1,:) ; write(*,*) d(2,:) ; write(*,*)
10
11      e = matmul(transpose(c),c)
12      write(*,*) e(1,:) ; write(*,*) e(2,:)
13
14      stop
15  end program example4_13b
```

[*6] 転置行列とは、行と列の配置を逆にした行列である。すなわち $\mathbf{C} = {}^t\mathbf{A}$ ならば、$A_{ij} = C_{ji}$ である。

2. 形状問い合わせ関数

例題 4-14

任意の配列 A を与えたとき、その次元数をスカラー整数型変数 n に、それぞれの次元の寸法を大きさ n の一次元整数型配列 m に格納した後出力しなさい。

テストデータ

```
real :: a(2,-1:3,0:2)
```

配列の形状その他は、組み込み関数を使って得ることができる。

配列の形状 (shape)

○ 配列 **A** の形状を戻り値として返す関数である。その形式は以下の通り、

```
shape(A)
```

· 戻り値は、整数型の 1 次元の配列である。その寸法は **A** の次元に一致し、その第 i 番目の配列要素の値は、**A** の第 i 次元の寸法となる。

配列の大きさと寸法 (size)

○ 配列 **A** の大きさまたは寸法を戻り値として返す関数である。その形式は以下の通り、

```
size(A [,dim])
```

· dim を指定した場合、配列 **A** の第 dim 番目の次元の寸法が、スカラーで返される。

· dim を省略した場合、配列 **A** の大きさが、スカラーで返される。

配列の上限 (ubound)

○ 配列 **A** の特定の、または全ての次元の添え字の上限を戻り値として返す関数である。その形式は以下の通り、

```
ubound(A [,dim])
```

· dim を指定した場合、配列 **A** の第 dim 番目の次元の添え字の上限が、スカラーで返される。

· dim を省略した場合、配列 **A** のすべての次元の添え字の上限が、寸法が **A** の次元の数である、一次元整数型の配列で返される。

配列の下限 (lbound)

○ 配列 **A** の特定の、または全ての次元の添え字の下限を戻り値として返す関数である。その形式は以下の通り。使い方は、ubound と同じである。

```
lbound(A [,dim])
```

例題 4-14 のプログラム例

```
1   program example4_14
2       implicit none
3       real :: a(2,-1:3,0:2)
4       integer :: n, m(7) ! 最大は 7 次元だから
5       n = size(shape(a))
6       m(1:n) = shape(a)
7       write(*,*) n
8       write(*,*) m(1:n)
9       stop
10  end program example4_14
```

3. 配列集計関数

例題 4-15

ある測定値の並びが、一次元の実数型配列 A に格納されている。ただし、欠測がありその場合は測定値に 999.0 以上の値が格納されている。このとき、
(1) 欠測をのぞいた有効な測定個数 n
(2) 有効な測定値のみの平均値 ave
をそれぞれ出力しなさい。

テストデータ

```
real :: a(7) = (/1.2, 2.1, 3.0, 999.0, 2.0, 1.1, 999.0/)
```

この章の冒頭の例で述べたように、配列全体の平均をとるためには、配列関数 sum を使えばよい。しかし、実際のデータ解析では「ある条件を満たす配列要素に対してのみ、··· という処理を行う。」という状況が往々にして発生する。このような場合、各配列要素ごとに if 文で分岐処理を行うことは可能であるし、実際 FORTRAN77 まではそのような方法しかなかったが、それでは配列関数のメリットが生きない。

そこで Fortran90/95 では、**mask 配列** という論理型の配列を指定して処理を制御する。mask 配列の配列要素が真の場合のみ、対応する配列要素に対して処理が行われる。たとえば、配列 A の配列要素で、正のものだけの和をとりたいときは、以下のように記述すればよい。

```
sum(A,mask=(A>0))
```

· mask 配列を指定するときは、「mask=」の後に、mask 配列の配列名、あるいは選別式を記述する。
· mask 配列は、処理対象となる配列 (この場合「A」) と形状適合でなければならない。
· 配列集計関数は、1 行に 1 回しか記述できない。

配列要素の和 (sum)

○ 配列要素の合計を計算する関数である。その形式は以下の通り。

```
sum(A[,dim][,mask])
```

· 配列 A だけを指定した場合、A の全配列要素の和 (スカラー) が返される。
· 次元 dim を指定した場合、dim 次元についての和がそれ以外の次元について、それぞれ計算されて返される。結果は A の次元-1 の配列となる。
· 選別式 mask を指定した場合、mask が真になる配列要素についての和が返される。

配列要素の積 (product)

○ 配列要素の積を計算する関数である。その形式は以下の通り。

```
product(A[,dim][,mask])
```

使い方は、sum の「和」を「積」に変えたものである。

配列要素の最大値 (maxval)

○ 配列要素の最大値を返す関数である。その形式は以下の通り。

```
maxval(A[,dim][,mask])
```

· 配列 A だけを指定した場合、A の全配列要素の最大値 (スカラー) が返される。
· 次元 dim を指定した場合、dim 次元のうちの最大値がそれ以外の次元について返される。結果は A の次元-1 の配列となる。
· 選別式 mask を指定した場合、mask が真になる配列要素のうちの最大値が返される。

配列要素の最小値 (minval)

○ 配列要素の最小値を返す関数である。その形式は以下の通り。

```
minval(A[,dim][,mask])
```

使い方は、maxval の「最大値」を「最小値」に変えたものである。

以下は、mask 配列自体に対する配列集計関数である。この場合「mask=」は不要である。

「真」の値を持つ配列要素の個数 (count)

○ 論理型配列の「真」の値を持つ配列要素の個数を返す関数である。その形式は以下の通り。

```
count(mask[,dim])
```

· 論理型配列 mask だけを指定した場合、「真」の値を持つ配列要素の個数 (スカラー) が返される。
· 次元 dim を指定した場合、dim 次元の「真」の値を持つ配列要素の個数が、dim 以外の次元についてそれぞれ返される。結果は mask の次元-1 の配列となる。

「真」の値を持つ配列要素の存否 (any)

○ 論理型配列の配列要素に「真」の値があるかどうかを返す論理型関数である。その形式は以下の通り。

```
any(mask[,dim])
```

· 配列 mask だけを指定した場合、配列要素のうちで 1 つでも「真」があれば「真」そうでなければ「偽」(スカラー) が返される。
· 次元 dim を指定した場合、dim 次元で「真」1 つでもあれば「真」、そうでなければ「偽」が、dim 以外の次元についてそれぞれ返される。結果は mask の次元-1 の配列となる。

「偽」の値を持つ配列要素の存否 (all)

○ 論理型配列の配列要素が、すべて「真」の値であるかどうかを返す論理型関数である。その形式は以下の通り。

```
all(mask[,dim])
```

· 配列 mask だけを指定した場合、配列要素がすべて「真」であれば「真」そうでなければ「偽」(スカラー) が返される。
· 次元 dim を指定した場合、dim 次元の配列要素がすべて「真」であれば「真」、そうでなければ「偽」が、dim 以外の次元についてそれぞれ返される。結果は mask の次元-1 の配列となる。

例題 4-15 のプログラム例

```
1   program example4_15
2       implicit none
3       real    :: a(7) = (/1.2, 2.1, 3.0, 999.0, 2.0, 1.1, 999.0/), ave
4       integer :: n
5
6       n = count(a<999.0)
7       write(*,*) n
8       ave = sum(a, mask=(a<999.0) )
9       write(*,*) ave/real(n)
10
11      stop
12  end program example4_15
```

4. 配列内位置関数

例題 4-16

ある測定値の並びが、一次元の実数型配列 A に格納されている。その測定値の最大値、およびその最大値をとる添え字の値を求めなさい。ただし、999.0 以上は欠測値とします。また、配列の下限値は 1 とは限りません。

テストデータ

```
real :: a(-3:3) = (/1.2, 2.1, 3.0, 999.0, 2.0, 1.1, 999.0/)
```

最大値、最小値の配列内の位置を求めるには、以下の関数を使う。ただし、これらの戻り値は最初から数えた順番を示すものであり、添え字と一致するとは限らない。添え字を求めるには、別に配列の下限値を求めて、それに順番を足して 1 を引く必要がある。

最大値をとる配列要素の位置 (maxloc)

○ 最大値をとる配列要素の配列内の位置を返す関数である。その形式は以下の通り。

```
maxloc(A[,mask])
```

· 戻り値は、一次元の配列である。A が一次元の場合も寸法 1 の一次元配列であり、スカラーではない。
· 選別式 mask を指定した場合、mask が真になる配列要素のうちの最大値となる位置が返される。

最小値をとる配列要素の位置 (minloc)

○ 最小値をとる配列要素の配列内の位置を返す関数である。その形式は以下の通り。

```
minloc(A[,mask])
```

使い方は、maxloc の「最大値」を「最小値」に変えたものである。

例題 4-16 のプログラム例

```
1   program example4_16
2       implicit none
3       real    :: a(-3:3) = (/1.2, 2.1, 3.0, 999.0, 2.0, 1.1, 999.0/)
4       integer :: m(1)
5
6       m = maxloc(a, mask=(a<999.0))
7       write(*,*) lbound(a,1) + m(1) - 1
8
9       stop
10  end program example4_1
```

4 章 の 章 末 問 題

【問題 4-1】(素数)

1000 以下の素数の個数はいくつあるか、エラトステネスのふるいを使って求めなさい。

(注) エラトステネスのふるいとは、ギリシャ時代から知られている素数を求める方法であり、

(1) 最初の素数は 2 である。

(2) 2 の倍数はすべて素数ではないので省く。

(3) 省かれなかったもののうちで、最小のものは 3。これが 2 番目の素数である。

(4) 3 の倍数はすべて素数ではないので省く。

このような操作を繰り返して素数をすべて求める方法である。

【問題 4-2】(完全数)

完全数とは、その数の約数から、その数自体を除いた和が、その数に一致する数のことである。 たとえば、6 の約数は 1,2,3,6 であり、6 を除いた 1+2+3 が 6 に一致するので、6 は完全数である。

4 ケタ以下 (10000 未満) の完全数の個数を求めなさい。

【問題 4-3】(友愛数)

自然数の組で、それぞれの約数の合計（その数自身はいれない）がお互いの数に一致するものを友愛数という。 両方とも 4 ケタ以下である、友愛数の組をすべて求めなさい。

【問題 4-4】(ソート)

すべて互いに異なる整数を並べたファイルがあり、その個数が先頭行に記載されている。このファイルから数値を取り出し、大きい順に並べ替えて出力するプログラムを作成しなさい。

テストデータ

```
5
25, 12, 50, 35, 29
```

(注) バブル・ソート ··· 隣り合った配列要素を、条件に合わなければ入れ替え、条件に合えばそのまま、という操作を最後の配列要素まで行い。 入れ替えの数が 0 になるまでその操作を続ける、という操作法。

【問題 4-5】

$\mathbf{A} = \begin{pmatrix} 1.0 & 4.0 & 7.0 \\ 2.0 & 5.0 & 8.0 \\ 3.0 & 6.0 & 9.0 \end{pmatrix}$ とするとき、$\mathbf{A}^2 (= \mathbf{AA})$、および ${}^t\mathbf{AA}$ をプログラムで求めなさい。

【問題 4-6】

ベクトル $\mathbf{a} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \end{pmatrix}$ と、$\mathbf{b} = \begin{pmatrix} -1.0 \\ 0.0 \\ 1.0 \end{pmatrix}$ の外積、$\mathbf{a} \times \mathbf{b}$ を計算するプログラムを作成しなさい。

【問題 4-7】

ベクトル $\mathbf{a} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \end{pmatrix}$ と、$\mathbf{b} = \begin{pmatrix} 0.0 \\ 1.0 \\ 2.0 \end{pmatrix}$ の内積、$(\mathbf{a}、\mathbf{b}) = {}^t\mathbf{a}\mathbf{b}$ および行列 $\mathbf{a}{}^t\mathbf{b}$ を計算するプログラムを作成しなさい。

（ヒント）ベクトルには `matmul` は使えないので、1 行 3 列、3 行 1 列の 2 次元配列を作る。

第 5 章　プログラムの構成

§ 5.1 Fortran プログラムの階層構造

1. 階層構造を持つ効用

これまでは、長くても数十行程度の比較的短い練習用のプログラムを作成してきたが、実際のデータ解析では、数百行、ときには数千行を必要とすることがある。そのような長いプログラムを書く場合には、全体をいくつかの「パーツ (部品)」に分割して作成した方が以下の点で有利である。

(i) プログラムでエラーが起きた場合の原因の発見は、一般にプログラムが短いほど容易である。そこで、長いプログラムを一挙に作成するのではなく、より小さなパーツに分割して一つずつ、テストを繰り返しながら順に作成させていく方が効率がよい。
(ii) 長いプログラムは複数のプログラマの共同作業によって作成することがあるが、その場合パーツに分割することは必須である。
(iii) パーツに分解しておけば、後に類似のプログラムを書く場合にも、共通に利用できる部分はそのまま再利用できる。

このように、長いプログラムを効率的に作成するためには、全体をどのようなパーツに分解するか、あるいはそれらのパーツをどのように組み合わせて全体を構成するか、ということは、Fortran に限らずすべてのプログラム言語で重要な要素である。

2. プログラム単位

Fortran のプログラムを構成する基本的な構成要素のことを、**プログラム単位** という。プログラム単位には次の 4 種類がある。

(1) 主プログラム

program 文で始まり、プログラムの開始と終了を行うプログラム単位であり、一つのプログラムに必ず 1 個だけ必要である。これまで作成したプログラムは、すべて主プログラムのみで構成されていた。

(2) モジュール (90)

Fortran90 から導入されたプログラム単位であり、以下の (3) から (5) の機能はモジュールで扱うことができる。言い換えれば、(3) から (5) は Fortran90/95 ではほとんど必要がない。

(3) 外部サブ・プログラム

このプログラム単位の機能は、ほとんどがモジュールで代替できる上に、暴走を防ぐためのチェックが面倒なので、Fortran90 で新規にプログラムを作成する際には、あまり使われることはない。ただし、FORTRAN77 はこれらを使って書かれているので、77 のライブラリを取り込む際には使う必要がある。

(4) データ定義ブロック

新規に Fortran90/95 を作成する際には、モジュールを使った方が便利なので、まず使われることはない。Fortran77 で作成されたライブラリを利用するときのみに使われる。[*1]

手続き

これらのユーザの作成するプログラム単位と、処理系の提供する、

(5) 組込みサブ・プログラム

とを合わせて **手続き** (procedure) といい、手続きを利用することを、手続きを呼び出す、という。

[*1] (3) および (4) については、この章ではごくわずかにふれるにとどめる。

3. 内部サブ・プログラム (90)

プログラム単位の内で、(1) 主プログラム、(2) モジュール、(3) 外部サブプログラム、はそのプログラム中にさらに **内部サブ・プログラム** という構成部分を持つことができる。[*2]

主プログラムおよび外部サブ・プログラム中の内部サブ・プログラムは、その属するプログラム単位に対して完全に従属しており、他のプログラム単位から呼び出すことはできない。

一方モジュール中のサブ・プログラムは、他のプログラム単位 (当該のモジュールを含む) から呼び出されることを目的としたものである。また、その呼び出しの可・不可を、必要に応じてユーザが指定することもできる。したがって、長大なプログラムを書く場合には、モジュールの使い方が重要である。

なお、サブ・プログラムの中に、さらにサブ・プログラムを入れ子にすることはできない。

4. サブ・プログラムの種類

Fortran のサブ・プログラムには、**関数** (function) と**サブルーチン** (subroutine) の二種類がある。

関数

関数とは、一般にそれを呼び出したプログラムからパラメータを受け取り、それらから戻り値 (関数値) を計算して返すサブ・プログラムのことをいう。使い方は、数学での関数と同様に、以下のように数式の中に組み込んで使う。

```
z = a * func1(x, y, 2) + b
```

このパラメータ、数学でいう独立変数 (上の場合なら x、y、2) のことを、Fortran では **引数**(ひきすう) とう。

サブルーチン

サブルーチンも関数と同様に、呼び出し元のプログラムから引数を受け取って処理を行うが、戻り値はなく、結果は (必要なら) 引数に代入して返す。サブルーチンは、次のように **call 文** で呼び出して使う。

```
call sub1(x, y)
```

5. プログラムのファイル分割の効用

ここまでは、プログラムを論理的に分割し、再構築することに関しての記述であったが、次にプログラムを物理的に分割することの効用を考える。

プログラムの実体は、テキスト・ファイルであり、そのサイズは 1 万行の巨大プログラムでも 1MB に達しない。現在のハードディスクおよび CPU ならば、全く問題なく処理 (コンパイル) できる量にすぎないが、それをなぜ複数のファイルに分割する必要があるかというと、次のような問題があるからである。

一般的なデータ解析では、ユーザは 1 つのプロジェクトに対して複数の (たぶん多数の) プログラムを作成する必要がある。そしてそれらのプログラムは、往々にして 90% 以上のコード (プログラムの文) が共通であり、ユーザはそのごく一部だけを変えて、種々の目的の計算を行うのである。

問題はその作業途中で、共通コード部分に不都合が見つかったときである。もしファイルが分割されていなければ、ユーザはこれまでに作成したすべてのプログラムに対して、該当部分のソース・コードの修正を行わなければならない。これは手間がかかるばかりでなく、修正ミスを引き起こす可能性もある。一方、分割されていれば、そのファイルの該当部分の修正 → 再コンパイル、だけの手間ですむ。[*3]これが比較的短いプログラムでも、ファイル分割の必要な理由である。

ファイル分割に当たっては、原則として「一つのプログラム単位は、その全てが一つのファイルに含まれていなければならない」。逆に一つのファイルが複数のプログラム単位を含むことは問題ない。

[*2] ただし、内部サブ・プログラムを含む外部サブ・プログラムを Fortran90 で新規に作成する機会は限られる。

[*3] 後述するように、モジュール・プログラム単位を使用するプログラムは、コンパイル順序に制限ができる。

§5.2 内部サブ・プログラム

まずは、主プログラムでの内部サブ・プログラムの使用法を述べることにする。なお、外部サブ・プログラムでの内部サブ・プログラムの使用法もほとんど変わらない。

1. 内部サブルーチン (引数のない場合)

例題 5-1

「整数型変数 i の値を 2 乗する」という処理を行う内部サブルーチン nijou を作成しなさい。ただし i は呼び出し側の主プログラムで宣言され、値を与えられているものとします。

内部サブルーチンの記述形式を以下に記す。内部関数もほとんど同じである。

内部サブルーチンの形式 (引数なし)

(i) 記述位置は、主プログラムの本文の直後。すなわち stop 文の後で、end program 文の前。

(ii) 先頭に「これ以後が内部プログラムである」という意味の、「contains」だけの 1 行を挿入する (**contains 文**)。内部サブ・プログラムが複数ある場合でも、contains 文は 1 行だけである。

(iii) 内部サブルーチンの先頭は、引数のない場合には以下の形式の subroutine 文である。

subroutine サブルーチン名

(iv) 内部サブルーチンの最後は、引数のあるなしにかかわらず、以下の end subroutine 文である。

end subroutine サブルーチン名

これらの形式は、主プログラムの program 文の「program」を「subroutine」に変えたものである。

(v) 「implicit none」は不要。これはプログラム単位の先頭に一つあればよい。

(vi) サブルーチンを使用するには、必要とする場所で call 文「call サブルーチン名」により呼び出す。

(vii) サブルーチンからの復帰は、stop 文の替わりに return 文を使用し、制御を呼び出し側に戻す。

これらをまとめると、以下の形式となる。

```
program 主プログラム名
    implicit none
    [主プログラム本体]
    call 内部サブルーチン名
    [主プログラム本体]
    stop
contains
  subroutine 内部サブルーチン名
    [内部サブルーチン本体]
    return
  end subroutine 内部サブルーチン名
end program 主プログラム名
```

例題 5-1 のプログラム例

```
1   program example5_01
2   ! 整数 i を 2 乗する内部サブ・プログラム (引数なし)
3       implicit none
4       integer :: i
5
6       i = 10
7       call nijou          ! サブルーチンの呼び出し
8       write(*,*) i
9       stop
10
11  contains                ! contains 文
12
13    subroutine nijou      ! subroutine 文
14
15      i = i*i
16      return              ! return 文
17
18    end subroutine nijou ! end subroutine 文
19
20  end program example5_01
```

親子結合

上の例題では、整数型変数「i」が主プログラム側で宣言された上で値が代入され、内部サブ・プログラム側はその値をそのまま使って計算を行っている。これはコンパイラが、内部サブ・プログラム側の変数「i」を、主プログラム側の変数「i」に結びつけているということである。このように内部サブ・プログラム側の変数を、同名の呼び出し側の変数と結びつけることを、**親子結合** (*host association*) という。

親子結合は便利ではあるが、逆に危険なこともある。たとえば、次の例のように整数型変数 i を呼び出し側とサブ・プログラム側の両方で do 制御変数として使ったとする。

```
program main
    implicit none
    integer :: i
        |
    do i=1,10
      call sub1
    enddo
        |
contains
  subroutine sub1
    do i=1,2
        |
```

これは「do loop の中で制御変数が変えられた」というエラーを引き起こす。それでもエラーが起きれば気がつくが、気づかないうちに誤った結果を出したら大変である。

この場合に限れば、subroutine 文の後に宣言文「　integer :: i」を挿入すれば、親子結合せずに二つの「i」は別の変数として認識されるが、一般に少し大きなプログラムでは、親子結合による変数の値の共有は避けるべきである。

2. 内部サブルーチン (引数のある場合)

例題 5-2

「整数型変数の値を 2 乗する」という処理を行う内部サブルーチン `nijou2` を作成しなさい。ただし 2 乗される変数は 呼び出し側の主プログラムで宣言され、値を与えられているが、その変数名は一定でないものとします。

たとえば平均値を求めるサブルーチンに対して、求めるべき変数名は毎回変わる。このような場合には引数を使う。引数のある内部サブルーチンの記述形式は、引数のない場合と以下の点が異なる。

内部サブルーチンの形式 (引数あり)

(iii′) 引数のある場合の `subroutine` 文は、以下のようにサブルーチン名の後に、引数のリストをコンマで区切り、() でくくって並べる。 ここで引数として現れる変数を **仮引数** という。なお、仮引数と一般の変数とはカテゴリーが異なるので、呼び出し側のプログラムに仮引数と同名の変数があってもかまわない。

```
subroutine サブルーチン名 (仮引数 1[, 仮引数 2][, 仮引数 3]...)
```

(vi′) `call` 文は、サブルーチン名の後に、やはり引数のリストをコンマで区切り、() でくくって並べる。ここで引数として現れる変数を **実引数** という。

```
call サブルーチン名 (実引数 1[, 実引数 2][, 実引数 3]...)
```

引数結合

サブ・プログラムが呼び出されると、実引数と仮引数が並び順により結合する。このような対応関係を、**引数結合** (*argument association*) という。引数結合関係は関数の実行が終了し、呼び出し側に制御が返されるときに消滅する。引数結合が行われるために、実引数と仮引数には以下の制約がある。

(i) 実引数と仮引数は、個数が一致しなければならない。[*4]

(ii) 対応する実引数と仮引数は、型が一致していなければならない。そのためにサブ・プログラム側で仮引数に対して型宣言を行う必要がある。ただし、仮引数名は実引数名と一致しなくてもよい。 仮に一致したとしても、それらは別な変数として扱われる。

(iii) 一つの内部サブ・プログラムで、引数結合と親子結合を同時に使用することができる。 ただし、一つの変数に同時に複数の結合を行うことは禁止されている。 すなわち、呼び出し側で実引数となっている変数を、サブ・プログラム側でその値を参照したり、変更したりした場合の結果は保証されない。

例題 5-2 のプログラム例

```
1   program example5_02
2   ! 整数を 2 乗する内部サブ・プログラム (引数あり)
3       implicit none
4       integer :: i, j
5
6       i = 10
7       call nijou2(i)                ! サブルーチンの呼び出しと実引数リスト
8       write(*,*) i
9
10      j = 20
11      call nijou2(j)                ! サブルーチンの呼び出しと実引数リスト
12      write(*,*) j
```

[*4] 後述する optional 引数は例外である。

```
13      stop
14
15    contains                         ! contains 文
16
17      subroutine nijou2(k)           ! subroutine 文と仮引数リスト
18        integer, intent(inout) :: k ! intent については、次の授受特性の項を参照
19        k = k*k
20        return                       ! return 文
21
22      end subroutine nijou2          ! end subroutine 文
23
24    end program example5_02
```

この例題では、サブ・プログラムで宣言された、整数型の仮引数「k」に、第 7 行で実引数「i」が引数結合されて、「i」の値が「k」に渡されて処理が行われる。サブ・プログラムから呼び出し側に制御が返されると、この引数結合は解消する。さらに第 11 行では実引数「j」が「k」に引数結合されて、「j」の値が「k」に渡されて処理が行われる。

引数のデータ授受属性とそのチェック

引数はその役割から次の 3 種類に分類できる。

(1) 呼び出し側から値を受け取って処理を行う。呼び出し側には値を返さない。

(2) 呼び出し側からは値を受け取らず、サブ・プログラムで処理を行った結果の値を、呼び出し側に送る。

(3) 呼び出し側から値を受け取って処理を行い、その結果を呼び出し側に送る。すなわち (1)+(2)

これらの違いにより、呼び出し時の実引数には以下の制約が生じる。

(1) の場合、呼び出し時には実引数の値が確定していなければならない。一方、値を受け取る必要がないので、変数だけでなく定数、あるいは式も実引数にすることができる (ただしその型は仮引数の型と一致していなければならない)。

(2) の場合、値を受け取るために、実引数は変数でなければならない (ただし、呼び出し時にはその値が確定している必要はない)。さらに、実引数リストにこの種の引数が複数ある場合には、二重結合を避けるために、それらの変数は全て異なっていなければならない。

(3) の場合、(1)(2) の両方。すなわち全て異なる変数であり、その値が確定していなければならない。

しかしながら、一般にコンパイラはこれらの種類を区別することができない。たとえば実引数が定数である仮引数に代入文を実行してもエラーすら出さないのである。これはしばしば致命的な誤りを導く。

そのような事態を避けるには、仮引数の宣言文の属性に、ユーザが授受特性を指定してやればよい。そうすればコンパイラはその情報から、不適当な処理に対してエラーあるいは警告メッセージを発することができる (ただし、値が確定しているかどうかのチェックはしないようだ)。具体的には、属性に,intent(xxxx) を指定する。xxxx は

(1) なら 「in」

(2) なら 「out」

(3) なら 「inout」

である。上の例題では、サブ・プログラム側の「i」は (3) に相当するので、第 18 行のように、

```
intent(inout)
```

と指定して、チェックに役立てている。

3. 内部関数

サブルーチンと異なり、関数は数学でいう関数値にあたる**戻り値** をもつ。戻り値の型には、実数型、整数型、論理型、文字型などがある。 この戻り値の型を関数型といい、作成時に宣言しなければならない。

例題 5-3

「整数型変数の値を 2 乗してその値を返す」という処理を行う内部関数 `nijou3` を作成しなさい。ただし 2 乗される変数は 呼び出し側の主プログラムで宣言され、値を与えられているが、その変数名は一定でないものとします。

内部関数の記述形式を以下に記す。なお (i),(ii),(v),(vii) は、内部サブルーチンと共通なので省略する。

内部関数の形式

(iii′′) 内部関数の先頭は、以下の形式の `function` 文である。

`function 関数名 ([仮引数 1][, 仮引数 2][, 仮引数 3]...)`

特別な場合として、引数のない関数も定義できるが、この場合でも「`()`」は省略しない方がよい。

(iv′′) 内部関数の最後は、以下の `end function` 文である。

`end function 関数名`

(vi′′) 関数を呼び出すためには、式中の一つの項として、次の形式で呼び出す。

`··· 関数名 ([実引数 1][, 実引数 2][, 実引数 3]...) ···`

(viii′′) `function` 文のあとに、関数名に対して型宣言を行う。これが戻り値の型となる。

(ix′′) 戻り値は、関数名に対して代入文で代入することにより値を与える。

これらをまとめると、以下の形式となる。

```
program 主プログラム名
    implicit none
    ┌──────────────┐
    │ 主プログラム本体 │
    └──────────────┘
    .... 関数名 (実引数リスト)....
    ┌──────────────┐
    │ 主プログラム本体 │
    └──────────────┘
    stop
contains
  function 内部関数名 (仮引数リスト)
    型名 :: 内部関数名        (型宣言文による、関数型の定義)
    ┌──────────────┐
    │ 内部関数定義本体 │
    └──────────────┘
    関数名 = 式           (代入文)
    return
  end function 内部関数名
end program 主プログラム名
```

基本型関数の型宣言

関数型が整数型、数値型、論理型などの基本型の場合には、関数名にたいした型宣言する替わりに、

```
integer function 関数名
```

という書き方ができる。例題ではこの形の関数型宣言を行っている。

例題 5-3 のプログラム例

```
1   program example5_03
2   !    整数を 2 乗する内部関数
3       implicit none
4       integer :: i=8
5
6       write(*,*) nijou3(i)
7       write(*,*) nijou3(-3) ! 実引数は、整数定数や整数式でもよい。
8
9       stop
10
11  contains
12
13    integer function nijou3(k)
14      integer, intent(in) :: k    ! 仮引数は、型宣言しなければならない
15
16      nijou3 = k * k
17
18      return
19    end function nijou3
20
21  end program example5_03
```

関数の引数のデータ授受属性

文法的には関数の仮引数も、サブルーチンと同様に in,out,inout の 3 属性が許されている。しかしプログラム作成に当たっては、すべて in 属性にすべきである。なぜなら、もし実引数の値が内部関数で変更されると、f(x)+g(x) と g(x)+f(x) の値が異なる、という事態が起こる可能性があるからである。

また、最適化にも、計算の並列化にも不利である。

ユーザ定義手続きの優先

サブ・プログラムにはユーザ定義サブ・プログラムの他に、コンパイラにより標準で提供されたり、コンパイラメーカーにより独自に (商品の差別化のために) 提供される組込みサブ・プログラムがある。このときもし、ユーザの作成した手続き (関数・サブルーチン) 名が、組込み手続きと一致したならば、そのプログラム単位内ではユーザの作成した方の手続きが有効になり、組込み手続きの方は無視される。したがってユーザはすべの組込み手続きの名を覚える必要はない。

複数の内部サブ・プログラム

一つのプログラム単位で、複数個の内部プログラムを使用したいときは、contains 文のあとにそれらを連続して記述すればよい。このとき内部サブ・プログラムは、同じプログラム単位中の他の内部サブ・プログラムを呼び出すことができる。

一方、内部サブ・プログラムの中に、さらに他の内部サブ・プログラムを記述することはできない。

§5.3 サブ・プログラムに関する追加事項

1. 変数名のスコープ (有効範囲)

例題 5-4

次のプログラムでどのような値が出力されるかを予測し、実際にプログラムして確かめなさい。

```
1   program example5_04
2       implicit none
3       integer :: i = 0, j = 10
4
5       write(*,*) func1(i)
6       write(*,*) func2(i)
7       stop
8
9   contains
10
11    integer function func1(k)
12      integer, intent(in) :: k
13      func1 = k + j
14      return
15    end function func1
16
17    integer function func2(j)
18      integer              :: i = 100
19      integer, intent(in)  :: j
20      func2 = i + j
21      return
22    end function func2
23
24  end program example5_04
```

○ func1 の戻り値を決めているのは第 13 行であるが、ここで、第 12 行で宣言された「k」は、主プログラム第 3 行で宣言された「i」と、第 11 行で引数結合されているのでその値は 0 である。一方「j」は第 3 行で宣言された「j」と親子結合しているのでその値は 10 である。したがってその和は 10 である。
○ func2 の戻り値を決めているのは第 20 行であるが、ここで、第 18 行で宣言された「j」は、主プログラム第 3 行で宣言された「i」と、第 17 行で引数結合されているのでその値は 0 である。一方「i」は第 19 行の宣言により親子結合が解消され、その値は 100 である。したがってその和は 100 である。

変数名の有効範囲

　このように変数名には、その名前と実体 (記憶番地) とが結合される有効範囲がある。
(i) 主プログラム (プログラム単位) で宣言された変数は、そのプログラム単位全体で有効である。上の例では、第 3 行で宣言された変数「i」「j」は、親子結合により内部プログラムでも利用できる。
(ii) 内部サブ・プログラムで宣言された変数は、そのサブ・プログラムの中でのみ有効である。上の例では、第 12 行で宣言された変数「k」は、func1 の中でしか利用できない。
(iii) 主プログラムで宣言された変数と同名の変数を内部サブ・プログラムで宣言したときは、両者は別モノ (いわば同名異人) とみなされ、当該のサブ・プログラム中では後者が、それ以外では前者が使われる。
　このように変数名が有効な範囲を、変数の**スコープ** という。

2. 自動変数と静的変数

データ解析では、サブ・プログラムで計算した値を保持しておきたいことがよく起きる。このとき、呼び出し側にその値を返して保存すれば何も問題はないが、受け渡す引数が増える、あるいはサブプログラムの独立性が低下するということもあり、サブ・プログラム側で、値を保持しておきたいところである。しかしその際、注意しなければいけないことがある。

例題 5-5

以下のような処理を行う、1 個の整数型引数をとる整数型内部関数 isum を作成しなさい。

(1) 最初に実引数に「0」を入れて初期化する。戻り値は 0 を返す。

(2) 以後は、それまでの実引数を累積した値を戻り値として返す。

(3) 実引数を「0」にすると、再度初期化される。

まず、誤ったプログラムの例を示す。

例題 5-5 のプログラム例 (誤った例)

```
1   program example5_05
2       implicit none
3
4       write(*,*) isum(0)
5       write(*,*) isum(1)
6       write(*,*) isum(2)
7
8       stop
9
10  contains
11
12    integer function isum(k)
13      integer, intent(in) :: k      ! 仮引数にはデータ授受属性を付記する
14      integer             :: jsum   ! 内部関数内の変数
15
16      if( k==0 ) then
17        jsum = 0
18      else
19        jsum = jsum + k
20      endif
21      isum = jsum
22      return
23    end function isum
24
25  end program example5_05
```

このプログラムでは、まず第 4 行で実引数 0 で呼び出され、第 17 行で累積値 jsum を 0 に初期化する。その後は実引数が 0 でなければ、jsum に引数の値を加えて戻り値として返す。

一見問題なさそうに思える。実際コンパイルは警告もなく通る。しかし実行すると、第 5 行の呼び出し時にエラーが起き、「jsum が定義されていません」という意味のエラーがでる。すなわち累積値の処理に失敗している。

その理由は、サブ・プログラムの変数には、**自動 (割付) 変数** と**静的変数** の 2 種類があるからである。

自動 (割付) 変数 (automatic variable)

自動変数とは、サブ・プログラムが呼び出されるごとに新たに記憶領域が確保され、サブ・プログラムが終了するとその領域が解放される変数のことである。

上の例題では、最初に isum が第 4 行で呼び出されたときに、内部プログラム isum 内に変数 jsum の記憶領域が確保されるが、第 23 行で制御が戻る際にその領域は解放される。したがって isum が再び第 5 行で呼び出されたときには値は確定していない。そのときの値は処理系依存であり、この処理系では不確定値としてエラーになった。

自動変数は、計算機の初期にメモリが非常に高価だった時代には、記憶領域の有効利用という点で有利でありさかんに使われた。しかしメモリの安い現在ではむしろ、予期しない処理を行おうとしたときにエラーを起こしてユーザに知らせる、という役割の方が大きい。

ただしこの場合、例題のように宣言すると変数 jsum は自動変数になるが、累積値を保持するためには、次の静的変数として宣言しなければならない。

静的変数 (static variable)

静的変数とは自動変数と異なり、コンパイル時に記憶領域が確保されて、プログラムの終了まで解放されない変数のことである。この変数では、手続きの呼び出しに無関係に値が保持される。[*5]

jsum を静的変数として宣言するには、第 14 行の代わりに次のいずれかの方法によればよい。

(i) jsum の宣言文に save 属性を付加して、静的変数であることを明示する。

```
integer, save :: jsum  ! 静的変数として宣言する
```

(ii) 第 14 行の後に、次の save 文を挿入する。

```
save :: jsum  ! 静的変数を宣言する変数リスト
```

この場合、静的変数に指定する変数のリストを、コンマで区切って並べることにより一度に宣言できる。ここで、サブ・プログラムで宣言する仮引数をのぞく全ての変数を静的変数として宣言したいならば、次の文となる。

(ii′) 第 14 行の後に、次の save 文を挿入する。ダブルコロン「::」は略すことが多い。

```
save [::]    ! 全ての変数を、静的変数として宣言する
```

save 文は implicit none と異なり、サブ・プログラムごとに宣言しなければならない。

また、次のような場合にも変数は静的変数になる。

(iii) 第 14 行で、jsum を初期化式で初期化する (この場合値は適当でよい)。

```
integer :: k, jsum = 999
```

(iii′) 第 14 行の後に、data 文を挿入して jsum を初期化する (同上)。

```
data jsum/888/
```

このような宣言が行われない場合、サブ・プログラムで宣言された変数は全て自動変数となる。[*6] 一方、主プログラムで宣言された変数は、その性格上すべて静的変数になり、上のような宣言は無意味である。

演 習 問 題

演習問題 1

上の例題を、正しい結果を出すように修正しなさい。

解答 上記の (i)-(iii′) のいずれかの方法によって修正する。

[*5] この 2 種類の変数は、機械語レベルで命令が異なる。
[*6] これは Fortran の歴史的遺産で致し方ない。

3. オプショナル引数 (90/95)

親子結合による変数値の受け渡しは、それ以上の宣言文を必要としないという点で手軽ではあるが、例題 5-4 で示したようにスコープが不明瞭になり、紛れを生じやすい。また、作成した内部プログラムを他の主プログラムで再利用する場合にも、変数名の管理に手間がかかる。そこでこの方法はあくまで主プログラムに強く従属する短いサブ・プログラムで使うにとどめ、独立性・汎用性の高いサブ・プログラムでの変数値は、引数渡しにした方が確実である。

しかし、全ての変数を引数渡しにするとなると、往々にして引数リストに 10 や 20 もの変数が並ぶことになる。これは計算機に負担がかかることは仕方ないとしても、仮引数リストに対応してそれぞれ実引数を設定しなければならないユーザに対しても、多くの手間を要求することになる。

実際にはデータ解析では、サブ・プログラムを呼び出すたびにすべての実引数を変えるケースは少ない。引数のかなりの割合は変える必要はなく、いくつかの引数のみを毎回変えていくことの方が多い。そのような場合には、実引数の指定を省略できる、オプショナル引数を使うことによって、引数リストを短くして扱いやすくすることができる。[*7]

オプショナル引数のサブ・プログラム側の設定

(i) 仮引数リストのうち、オプショナル引数としたい仮引数に、属性 `optional` をつけて宣言する。ただし、その仮引数のデータ授受属性は `intent(in)` または `intent(inout)` でなければならない。

(ii) 仮引数リストの並び順は、すべてのオプショナルでない引数を先に並べ、その後にオプショナル引数を並べなければならない。[*8]

(iii) オプショナル引数が実引数で与えられているか否かをサブ・プログラム側で知るためには、組込み関数 `present` を使う。これは `present(`**オプショナル仮引数名**`)` が、実引数が指定されていれば「真」、省略されていれば「偽」になる関数であり、これを使って処理を行う。オプショナルでない仮引数に対して、`present` を使用してはならない。

これらをまとめると、以下のような形式になる (内部関数の場合。サブルーチンでも同様)。

```
function 関数名 ([仮引数 1,]...[オプショナル仮引数 1,][オプショナル仮引数 2]...)
  (関数とオプショナルでない仮引数の型宣言リスト)
  型名, intent(in), optional :: オプショナル仮引数 1
      |
 if(present(オプショナル仮引数名)) then ...
      |
```

呼び出し側の実引数の指定

(i) オプショナル引数に対する実引数は省略できる (そうでないものについてはもちろん省略できない)。

(ii) オプショナル引数に対して実引数を指定するときは、オプショナルでない実引数リストの後に、

```
仮引数名 = 実引数
```

の形式で指定する。この場合、オプショナル引数同士の実引数の指定順序は問わない。

(iii) 実引数を指定すべきオプショナル引数に対して、リストの最初からその引数までオプショナルかそうでないかにかかわらず、すべて実引数が指定してあれば、オプショナルでない引数と同様に指定できる。

[*7] データ解析では重要なテクニックの一つだが、数値解析やシミュレーションの Fortran 参考書ではあまり言及されない。

[*8] 配列でないオプショナル仮引数を、配列のオプショナル仮引数の前に並べた方がよい。

例題 5-6

以下の条件で処理を行う、5 個の整数型データからその平均を求める、実数型内部関数 average を作成しなさい。

(1) データは整数型の大きさ 5 の配列を実引数として与える。

(2) 有効データ数 n を指定したとき、平均をとるデータは最初の n($1 \leqq n \leqq 4$) 個までとする。

(3) 異常値 idex を指定したとき、idex を除いて平均する。

テストデータ

idata=(/ 11, 12, 13, 14, 999/)

n は指定せず、異常値として idex = 999 を指定する。

例題 5-6 のプログラム例

```
1   program example5_06
2       implicit none
3       integer :: idata(5) = (/11, 12, 13, 14, 999/)
4
5       write(*,*) average(idata, idex=999)
6
7       stop
8
9   contains
10
11    real function average(idata, n, idex)
12      integer, intent(in)           :: idata(5)
13      integer, intent(in), optional :: n, idex
14      integer                       :: kn, n_default = 5
15
16      kn = n_default
17      if(present(n)) kn = n
18
19      if(present(idex)) then
20        average = sum( idata(1:kn), idata /= idex)
21        average = average / count(idata(1:kn) /= idex)
22      else
23        average = sum( idata(1:kn) )/real(kn)
24      endif
25      return
26    end function average
27
28  end program example5_06
```

○ 第 5 行でオプショナル引数を指定している。もし n=4 を指定するなら average(idata, 4, 999)。

○ 第 16 行でいったん kn にサブ・プログラム側で定義された n_default の値 5 を代入しした後に、第 17 行で n に対する実引数が与えられたかどうかを調べ、与えられていた場合は kn の値を n に変更する。この n_default のような値を**デフォルト値** という。

○ 第 19 行で idex に対する実引数が与えられたかどうかを調べ、与えられていた場合は第 20,21 行の処理を、与えられていない場合は第 23 行の処理を行う。

§5.4 モジュール (90/95)

1. モジュールとは

モジュールとは Fortran90 によって追加された新たなプログラム単位であり、FORTRAN77 からの改訂の重要な柱の一つとなっている。特にデータ処理では、モジュールの使い方がプログラムの出来不出来を決める重要な要素となる。

一般に Fortran90/95 のプログラムは、1 個の主プログラムと複数のモジュールから構成されるが、モジュールは他の (複数の) プログラム単位から共通に参照されるための、変数・定数とサブ・プログラムの集合体であり、モジュール本体は特定の処理を行うことを目的としない。したがって、モジュール本体は宣言文のみで構成され、実行文を持たない (サブ・プログラム部では持つ)。

モジュールの形式

一般的なモジュールは、以下の形式である。

```
module モジュール名              ! module 文
    implicit none
    ┌────────────────────┐
    │ モジュール変数の宣言部 │
    └────────────────────┘
contains
    ┌──────────────────────────┐
    │ モジュール・サブ・プログラム部 │
    └──────────────────────────┘
end module モジュール名        ! end module 文
```

宣言文、およびサブ・プログラムの形式は、内部サブ・プログラムと同様である。

モジュール名は、プログラム全体で一意的でなければならない。すなわち、一つのプログラム上で、同名の変数、サブ・プログラム、他のモジュール等を宣言してはならない。[*9]

モジュールの参照

このように定義されたモジュールのモジュール変数、あるいはモジュール・サブ・プログラムを、他のプログラム単位から利用するためには、そのプログラム単位の開始文 (`program` 文等) の直後に、[*10] 次の `use` 文を記述して、該当のモジュールを参照することを指定する。[*11]

```
use モジュール名
```

複数のモジュールを参照する場合でも、1 行に複数の参照指定を行うことはできず、参照するモジュールごとに `use` 文が必要である。

モジュールが他のモジュールを参照することも可能である。ただしその参照関係は、次節で述べるように主プログラムを頂点として、上位モジュール (より頂点に近いモジュール) から下位モジュールを参照するという階層構成でなければならず、循環的な参照はできない。

[*9] すでに使っている名前と衝突したときの回避方法は後述するが、これはあくまで緊急的回避手段であり、望ましいとはいえない。また特に、同じディレクトリの他のプログラムで、同名で異なる内容のモジュールが使ってはならない。

[*10] その前に注釈文はあってもよいが、`implicit none` などの宣言文よりは前でなければならない。

[*11] 逆に言えば、そのプログラム単位で該当のモジュールを使いたくなければ、use 文を記述しなければよい。

2. モジュールのコンパイル

モジュールのコンパイル順序

モジュールは、かならずそれを参照するプログラム単位よりも先にコンパイルされていなければならない。*12なぜならば、モジュールがコンパイルされると通常のオブジェクト・ファイル「*.obj」に加え、モジュール情報ファイル「*.mod」が出力される。use 文でそのモジュールを参照するよう指定されたプログラム単位は、このモジュール情報ファイルを参照しつつ、コンパイルするからである。*13

プログラム例 5-a　モジュールから他のモジュールを参照する

```
1   module module3
2       implicit none
3       integer, parameter :: i = 1
4   end module module3
5
6   module module2
7       implicit none
8       integer, parameter :: j = 2
9   end module module2
10
11  module module1
12      use module2
13      use module3
14      implicit none
15      integer :: k = i + j
16  end module module1
17
18  program main
19      use module1
20      implicit none
21      write(*,*) i, k
22      stop
23  end program main
```

この例では主プログラム main はモジュール module1 を参照し、モジュール module1 は他のモジュール module2 および module3 を参照している。このとき、上述したコンパイル順の規則により、

・module1 は、main よりも先にコンパイルされていなければならない。
・module2 および module3 は、module1 よりも先にコンパイルされていなければならない。
・module2 と module3 は、どちらが先でもよい。

というコンパイル順序に関する条件が付くのでこのような順になる。

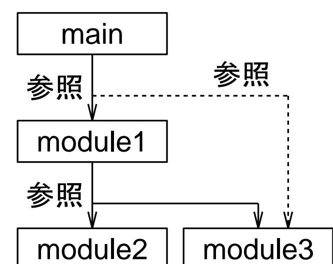


Fig.5.1

ここで main は、第 21 行で図の点線のように module3 の変数を直接参照しているが、module3 に対する use 文は不必要である。それは、module3 を use 文によって参照している module1 を参照するということが、module3 を間接的に参照しているとみなされるからである。

*12 ついでながら、この点について、「実践 Fortran90 プログラミング」に記載された例題は誤りである。その本を書くために使われた処理系では、同一ファイル内のモジュールの後置は許されたかもしれないが、一般にはそうでない。
*13 特に、修正後の再コンパイルの場合、順番を誤って修正前のモジュール情報ファイルを参照しないように注意。

モジュールの階層構成

モジュールの参照関係が複雑な場合、コンパイル順序を決めるためにモジュールの階層的分類が有効である。

右図は、主プログラムと 7 個のモジュールからなるプログラムであり、矢印が `use` 文による参照指定を表している。このようなプログラムのコンパイル順序を決めるため、次のようにモジュールのレベルを定める。

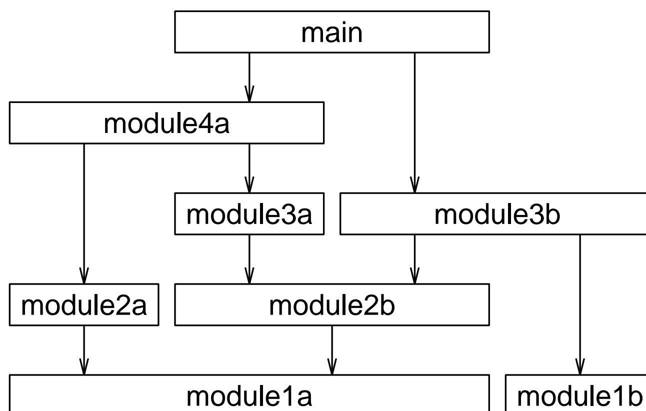


Fig.5.2

(i) `use` 文のないモジュールは第 1 レベル。
(ii) 第 1 レベルのモジュールしか参照しないモジュールは第 2 レベル。
(iii) 以後同様に、高々第 n レベルのモジュールしか参照しないモジュールは第 n+1 レベル。
(iv) 主プログラムは一番最後になり、レベルは最高となる。

このようにモジュールをレベルごとに分類しておき*14、次の規則に従って、順次コンパイルを行えばよい。
(1) 下位レベルのモジュールから上位レベルのモジュールへと、順番にコンパイルする。
(2) 同レベルのモジュール同士の順番は、どちらが先でもよい。
(3) 最後に主プログラムをコンパイル後、リンクして実行ファイルを生成する。

修正コンパイル

あるモジュールのソース・コードを修正したとき、再コンパイルはそのモジュールだけ行えばよいのではなく、そのモジュールを直接、間接に参照するモジュールは、主プログラムを含め、すべて再コンパイルしなければならない。上の例でいえば、もし `module2b` のソース・コードを修正したならば、`module2b` をコンパイルした後、`module3a`、`module3b` → `module4a` → `main` を、この順に再コンパイルしなければならない。*15

コンパイルの方法 (1) … 一括コンパイル

モジュールを順番にコンパイルするための最も簡単な方法は、一つのソースファイルに、下位のモジュールを先に、上位のモジュールを後に記述することである。これなら一回のコマンドで全プログラムをコンパイルでき、修正も容易である。

しかし、この方法は各モジュールが短い場合には便利であるが、一つ一つのモジュールが長くなると、デバッグや保守が容易でなくなるという弱点がある。これではそもそもパーツ化した意味が薄れる。

また、プロジェクトの複数のプログラムで、汎用性の高いモジュールを共通に使うときには、さらに問題である。万一そのモジュールを修正する必要ができたときには、そのモジュールが含まれているプログラムを全て同様に修正しなければならない、ということになる。そしてもしそれに失敗すれば、同じ名前で内容の異なるモジュールが存在する、というに好ましくない事態になる。

したがって、モジュールが極端に短くない限り、モジュールを 1 個ごとに 1 個のファイルに収めてコンパイルする方法が一般的である。

*14 実際のプログラムでは、3 レベルかせいぜい 4 レベルだろう。
*15 このとき実行ファイルが残っているとうまくコンパイルできない場合があるので、修正コンパイルの前に実行ファイルは消去しておくことが望ましい。

コンパイルの方法 (2) … 分割コンパイル

最も基本的な方法は一括コンパイルとは逆に、コマンドラインで 1 ファイル (1 モジュール) ずつコンパイルを行っていくことであり、これを分割コンパイルという。ただし、その方法は処理系によって少しずつ異なる。ここでは例として、Intel fortran でプログラム例 5-a をコンパイルする手順を示す。

(i) まず、プログラムをモジュールごとに「module3.f90」、「module2.f90」、「module1.f90」、「main.f90」の 4 個のファイルに分割する。

(ii) 次に、コンパイル・オプション「-c」をつけて、「main」を除くモジュールを、下位レベルから次々にコンパイルする。

```
ifort -c module3.f90
ifort -c module2.f90
ifort -c module1.f90
```

(iii) 最後に、コンパイル・オプション「-c」をつけずに、「main」をコンパイルして実行ファイルを作成する。このとき、main が参照しているモジュールのオブジェクト・ファイルをその前に指定しなければならない。

```
ifort module1.o main.f90
```

この方法は確実ではあるが、多数のコマンドを必要とするという欠点がある。ただし、make コマンドが使えれば、かなりの手間が削減される。

コンパイルの方法 (3) … 連結コンパイル

一括コンパイルと分割コンパイルの、双方の欠点を補う方法として、連結コンパイルがある。これは、モジュールごとに物理的に分割されたファイル群を論理的に連結し、一括コンパイルと同様の結果を得るものである。上の例ならば、次のようなコマンドを入力する。

```
ifort module3.f90 module2.f90 module1.f90 main.f90
```

これにより、modelue3、modelue2、modelue1、main をこの順に連結した 1 個のファイル、すなわちプログラム例 5-a をコンパイルする事と同じ結果が得られる。

連結コンパイルにより、一括コンパイル、分割コンパイルの弱点は解消されるが、新たに、コマンドラインが長くなって打ち間違いを起こしやすい、という問題が起きる。これには、

(i) Windows の Dos 窓、Linux の Gnome エディタとも、「↑」を入力すれば直前に実行したものと同じコマンドがプロンプトの後に表示される。これをそのまま、あるいは修正して実行すればよい。

(ii) Windows ならば、上のコマンドを「○○.bat」というテキストファイル (バッチファイル) に保存し、プロンプトの後に「○○」と入力すれば、そのコマンドが実行される。

(iii) Linux ならば、上のコマンドを適当な名前のファイルに保存し、実行可能属性を与えれば、そのファイル名を入力しただけで、そのコマンドが実行される。

コンパイルの方法 (4) … 統合環境でのコンパイル

Visual Studio や Plato3 のような統合環境では分割コンパイルと同様に、「main」を除くモジュールを、下位レベルから次々にコンパイルしていく。そして最後に main を読み込んで build により実行ファイルを作成し、Run で実行する。また、いくつかの手順をまとめて指定できるコマンドもある。

一般に統合環境では、オブジェクト・ファイルやモジュール情報ファイルの存在場所を指定する必要があるが、カレント・ディレクトリ (フォルダ) にあるならば指定は不要である。ただしこの場合、別なプロジェクトを同じディレクトリで行うと、module 名が重なることがあるので、その点は注意しなければならない。

3. モジュール変数との参照結合

たとえば円周率 π を、主プログラム main とモジュール module1 で使うとする。このとき、モジュール側で π の値を名前付き定数に定義しておき、主プログラムでそのモジュールを use すれば、主プログラム本体およびその内部サブ・プログラム中で、その名前付き定数の値を参照することができる。

また、データ数などの変数値をモジュールの変数に渡しておけば、他のモジュールからもその変数の値を参照することができる。

プログラム例 5-b　モジュール変数との参照結合

```
1   module module1
2       implicit none
3       real, parameter :: pai=3.14159265
4       integer         :: n
5       save    ! 多くのコンパイラでは不要
6
7   end module module1
8
9   program main
10      use module1
11      implicit none
12      real :: rectangle = 90.0
13
14      read(*,*) n
15      write(*,*) pai*real(n)/(2.0*rectangle)
16
17      stop
18  end program main
```

第 5 行の save 文は n の値を保持するために入れているが、多くのコンパイラではモジュール本体で宣言された変数についてはデフォルトで静的変数の属性を与えているので、不要なことが多い。

参照結合 (90/95)

主プログラム main では、implicit none 宣言があるにもかかわらず、宣言文のない pai や n という変数を使っている。これは、モジュール module1 側で宣言され、記憶領域が確保された pai および n と、use 文により結合されているからである。このような結合を**参照結合**という。

参照結合は、Fortran90 から導入されたものである。FORTRAN77 ではコモン (共通) ブロックの変数がその役割を果たしていたが、これは記憶域結合という形式であり、メインテが複雑であったため、代替するために導入された。

大域変数と局所変数

第 3 行で宣言された名前付き定数 pai や、第 4 行で宣言された変数 n は、use 文で module1 を参照指定する全てのプログラム単位で、その名前で利用できる。このように複数のプログラム単位で共通に利用できる変数を**大域 (的) 変数** (*global variables*) という。

一方、第 12 行で宣言された定数 rectangle は、main 以外のプログラム単位からは使えない。このように一つのプログラム単位でしか有効でない変数を総称して**局所変数** (*local variables*) という。[*16]

[*16] 大域変数という名称は、もともと FORTRAN77 で全プログラムをスコープとする変数のことを指していたのを流用したものである。しかし Fortran90/95 では、宣言されたより上位のモジュールでしか有効でないので、適切な使い方とはいえない。

4. モジュール・サブ・プログラム

他のプログラム単位と同様に、モジュールでも contains 文により内部サブ・プログラムを定義することができる。これをモジュール・サブプログラムという。モジュール本体に実行文は書けないが、モジュール・サブ・プログラム中には記述できる。そしてモジュール・サブ・プログラムはプログラム名を参照結合するにより、参照元の内部サブ・プログラムと同様に使用することができる。

例題 5-7

例題 5-5 を、jsum を静的変数に修正した上で、内部関数をモジュール関数に変更しなさい。

以下は、save 文により静的変数に指定した例である。

例題 5-7 のプログラム例

```
1   module module_sum
2       implicit none
3
4   contains
5
6     integer function isum(k)
7       integer, intent(in) :: k
8       integer             :: jsum
9       save       ! save 文は仮引数には無効
10
11      if( k==0 ) then
12        jsum = 0
13      else
14        jsum = jsum + k
15      endif
16      isum = jsum
17      return
18    end function isum
19
20  end module module_sum
21
22  program example5_07
23      use module_sum
24      implicit none
25
26      write(*,*) isum(0)
27      write(*,*) isum(1)
28      write(*,*) isum(2)
29
30      stop
31
32  end program example5_07
```

第 2 行および第 24 行の implicit none はプログラム単位の冒頭に、それぞれ 1 回宣言すればよいが、save 文はサブ・プログラムごとに宣言しなければならない。これが面倒ならば、コンパイラ・オプションで指定すれば、明示的に自動変数に指定されていない変数は、全て静的変数になる。

内部サブ・プログラムとモジュール・サブ・プログラムとの比較

内部サブ・プログラム名は local である。すなわちそのプログラム単位でしか使用できない。一方、モジュール・サブ・プログラム名は global である。すなわち、参照される全てのプログラム単位から利用できる (ただし、これは次節でで述べるように、便利な一方で名前の衝突を起こす可能性がある)。

変数や定数の引き渡しを、内部サブ・プログラムが親子結合で行うところを、モジュール・サブ・プログラムは、モジュール宣言部での参照結合で行うところが両者の違いである。一方、引き渡しを引数結合で行うならば、両者に使い方の差はない。

これより、モジュール・サブ・プログラムは、モジュールだけを切り離して、他の主プログラムで利用することに適しているといえる。変数の親子結合を行ってしまうと、これは容易でなくなる。したがって汎用的に利用されるサブ・プログラムはモジュール・サブ・プログラムにするべきである。一方、一回きりのサブ・プログラムであれば、内部サブ・プログラムの方が保守が容易である。ただしその場合でも、後に汎用化する可能性があるならば、できる限り親子結合は避けるべきである。

モジュール・サブ・プログラムの統合

データ解析では、短いサブ・プログラムを多数使用することが多い。さらには、わずかに異なるサブ・プログラムを交代に使うこともある。このような場合には、どのサブ・プログラムを参照するか、整理することがジョブの効率の向上のために重要な要素となってくる。

そこで、一つのモジュールに必要な定数類をまとめて宣言し、その後にそれらを使う関連するサブ・プログラムを列記すれば、参照の回数を少なくすることができる。このようなモジュールの構成は、Fortran90/95 プログラムを利用するために、きわめて重要である。

演習問題

演習問題 2

例題 5-1 と例題 5-2 のサブルーチンを、モジュール・サブルーチンに書き換えなさい。ただし例題 5-1 では、i はモジュール側で宣言され、主プログラム側で値を与えられるとします。

解答

```
module module1
    implicit none
    integer :: i
contains
  subroutine nijou
    i = i*i
    return
  end subroutine nijou
end module module1

program main
    use module1
    implicit none
    i = 10
    call nijou
    write(*,*) i
    stop
end program main
```

```
module module2
    implicit none
contains
  subroutine nijou2(k)
    integer, intent(inout) :: k
    k = k*k
    return
  end subroutine nijou2
end module module2

program main
    use module2
    implicit none
    integer :: i
    i = 10
    call nijou2(i)
    write(*,*) i
    stop
end program main
```

5. モジュールのスコープとその制限

モジュールはこのように強力なパーツであるが、その強力さゆえに困ることも起きる。それはモジュール変数名や、モジュール内部・サブプログラム名、またはモジュール名自身が、すでに使用している名前と衝突を起こすときである。

このような衝突を防止するには、以下に述べるようにいくつかの方法がある。

(1) モジュール側でのスコープ制限

モジュールで使われる変数・定数 (以下、合わせて変数という) は、目的により次の 3 種に分類できる。
(i) 外部から参照するために宣言された、モジュール変数。
(ii) モジュール内で、親子結合などで使われるために宣言された、モジュール変数。
(iii) モジュール・サブ・プログラム内で宣言された変数。

これらはそれぞれスコープが異なる。(i) のスコープは、当該のモジュールおよびそれを直接、間接に参照するプログラム単位であり、(ii) のスコープは、当該のモジュール全体である。また、(iii) のスコープは、それが定義されたサブ・プログラムの中だけである。

モジュール・サブ・プログラムも、同様に次の 2 種に分類できる。
(i′) 外部から参照するために定義された、サブ・プログラム。
(ii′) モジュール内で、他のサブ・プログラムから呼ばれるために定義された、サブ・プログラム。

(i′) のスコープは (i)、 (ii′) のスコープは (ii) とそれぞれ一致する。

このうち (iii) はモジュールの外部からは見えないので、外部で同名の変数が宣言されても、衝突を起こすことはない。逆に (i) および (i′) は当然外部から参照できなければならない。問題は (ii) および (ii′) が、モジュール外から参照できてしまうことである。これは意図しない名前の衝突を起こす可能性がある。これを予防するために、Fortran では (i) および (i′) の変数、サブ・プログラムには「public」、(ii) および (ii′) の変数、サブ・プログラムには「private」の属性を付加して、両者を区別する。具体的には、

(1) 変数またはサブ・プログラムの宣言文で、{public|private}属性を付加して宣言する。

```
型名, public  :: {変数名|サブ・プログラム名}
型名, private :: {変数名|サブ・プログラム名}
```

(2) モジュールの宣言部で、public 文、または private 文により、属性を付加したい変数名またはサブ・プログラム名をリストアップする。。

```
public  :: {変数名|サブ・プログラム名}[, 変数名|サブ・プログラム名]...
private :: {変数名|サブ・プログラム名}[, 変数名|サブ・プログラム名]...
```

(3) 上の (1) および (2) で属性指定されていない変数またはサブ・プログラムに属性をまとめて付加する。

```
{public|private}
```

デフォルトは、すべて public である。すなわち (3) で、

```
public
```

と指定されたことと同値である。

プログラム例 5-c　モジュール側でのスコープ制限の例

```
1   module module1
2       implicit none
3       private
4       real, parameter :: pai = 3.14159265
5       integer, public :: j = 2
6       public radian
7
8   contains
9     real function radian(x)
10      real, intent(in) :: x
11      real             :: rectangle_2 = 180.0
12      radian = x/rectangle_2*pai
13      return
14    end function radian
15  end module module1
```

上の例では、モジュール変数 `pai`、`j` およびサブ・プログラム `radian` に対してスコープを与えている。

・第 3 行で、明示的に `public` 宣言されていないものは、`private` 属性を与えるように変更している。

・第 5 行で、モジュール変数 `j` に、`public` 属性を与えている。

・第 6 行で、サブ・プログラム `radian` に、`public` 属性を与えている。

この結果、モジュール `module1` の外部から参照可能なのは `j` と `radian` となる。`pai` は参照できない。

誤って `private` を指定することを忘れるのを防ぐために、第 3 行のように暗黙に `private` を指定して、必要なものだけを `public` 指定した方が、逆の場合よりも望ましい。

(2) 参照する側でのスコープ制限

モジュールを呼び出す側で、あらかじめそのモジュールの一部しか使わないことがわかっていれば、使う部分だけを **`only` 句** で指定することができる。これにより、もしモジュールの使わない部分で名前の衝突が起きていても、それは参照しないということで回避できる。`only` 句は `use` 文でモジュールを指定すると同時に、次の形式で指定する。

```
use モジュール名, only : {モジュール変数名|モジュールサブ・プログラム名}[,...]
```

次のプログラムは、`module2` のモジュール変数のうち、`i` だけを参照するという例である。

プログラム例 5-d　参照する側でのスコープ制限の例

```
1   module module2
2       implicit none
3       integer :: i = 1, j = 2
4   end module module2
5
6   program main
7       use module2, only : i
8       implicit none
9       write(*,*) i
10  !    write(*,*) j
```

```
11        stop
12    end program main
```

第 10 行のコメントをはずすと、「j が未定義である」という実行時エラーを起こす。これは第 3 行の j が参照されていないためである。

(3) 参照する側での一時的名称変更

参照するプログラム単位中に、モジュールの名前と一致する名前があるとわかっているときに、緊急措置としてモジュール側の方の変数、サブ・プログラムの名前を一時的に仮の名前に変更して衝突を避けることができる。このときの仮の名前を**仮称** といい、仮称を設定する事を**仮称指定** という。仮称指定の形式は、次のとおりである。

```
use モジュール名, 仮称 => モジュール側での名前 [, 仮称 => モジュール側での名前]
```

次のプログラムでは、モジュール側にも参照する側にも i という変数があるため、モジュール側の i を一時的に iii に仮称指定している。

プログラム例 5-e　参照する側での仮称指定

```
1     module module3
2         implicit none
3         integer :: i = 1
4     end module module3
5
6     program main
7         use module3, iii => i
8         implicit none
9         integer :: i = 2
10        write(*,*) i, iii
11        stop
12    end program main
```

仮称指定したモジュール変数、サブ・プログラムに対して、参照側で型宣言してはならない。元の変数、サブ・プログラムはすでに宣言されているので、二重に宣言することになるからである。

仮称指定はあくまで暫定的な措置であり、そのままではデバッグなどで支障を来す。適当な時期にどちらかの変数名を変更するなどして、衝突を回避すべきである。

演 習 問 題

演習問題 3

(1) プログラム例 5-a を、モジュールごとに 4 個のファイルに分割し、コンパイル、実行しなさい。

(2) module3 の i=1 を i=3 に変更して、再コンパイルして実行しなさい。

解答

略 ··· 再コンパイルの際のコンパイル順に注意。

§5.5 組込みサブルーチン

組込み関数についてはこれまでに随所で述べてきたが、組込み手続きには関数以外にサブルーチンもある。ただし、その種類は 6 種類しかない上、通常使われるのはシステムの時刻を受け取るサブルーチン「date_and_time」および CPU 実行時間を受け取るサブルーチン「cpu_time」程度である。

例題 5-8

システムの日付と時刻を、画面に表示しなさい。

組込みサブルーチン date_and_time によって、PC の内蔵時計の時刻を受け取ることができる。

```
call date_and_time([date][,time][,zone][,values])
```

引数は次の形式で、すべて optional 引数である。

```
character(len=8),  optional, intent(out) :: date
character(len=10), optional, intent(out) :: time
character(len=5),  optional, intent(out) :: zone
integer,           optional, intent(out) :: values(8)
```

戻り値の内容は以下の通り、

date = "yyyymmdd"　→ yyyy : 年 mm : 月 dd : 日

time = "hhnnss.sss" → hh : 時 nn : 分 ss.sss : 秒

zone = "+0900"　→ : GMT からの時差

values は大きさ 8 の整数型配列であり、配列要素の内容は順に、

(1) 年、(2) 月、(3) 日、(4) 時差 (単位は分)、(5) 時、(6) 分、(7) 秒、(8) ミリ秒

以下は、values を使ってシステム時刻を受け取った場合である。

例題 5-8 のプログラム例

```
1   program example5_08
2   ! get system date and time
3       implicit none
4       integer :: sys_date(8)
5
6       call date_and_time( values = sys_date )
7
8       write(*,*) 'year     :',sys_date(1)
9       write(*,*) 'month    :',sys_date(2)
10      write(*,*) 'day      :',sys_date(3)
11      write(*,*) 'hour     :',sys_date(5)
12      write(*,*) 'minute   :',sys_date(6)
13      write(*,*) 'second   :',sys_date(7)
14      write(*,*) 'm second :',sys_date(8)
15      write(*,*)
16      write(*,*) 'GST+hours:',sys_date(4) / 60
17
18      stop
19  end program example5_08
```

5 章 の 章 末 問 題

【問題 5-1】

プログラムの開始時から、呼び出された回数を関数値として返す、整数型の内部関数を作成しなさい。

【問題 5-2】

西暦年、および年通算日を入力し、年、月、日に換算するプログラムを、閏年の判定部分を内部サブ・プログラムにすることにより、書き換えなさい。

【問題 5-3】

整数型の引数を受け取り、その数を除く約数の和を関数値として返す、整数型の内部関数を作成し、その関数を使って 10000 までの完全数をすべて出力しなさい。

【問題 5-4】

整数型の引数を受け取り、その数が素数ならば「真」そうでなければ「偽」を関数値として返す、論理型の内部関数を作成し、その関数を使って 100 以下の素数の個数を求めなさい。

【問題 5-5】

球の半径が引数として与えられたとき、その体積と表面積を返すモジュール・サブルーチンを作成しなさい。

【問題 5-6】

寸法 `n` の一次元実数型配列 `x` を引数と、実数型 `optional` 引数 `p` を受け取り、`p` が指定していなければその合計を、`p` が指定してあれば `x` の配列要素のうちで `p` 以上のものだけの合計を返すモジュール関数を作成しなさい。ただし、`n` はモジュールの宣言文で名前付き定数として与えるものとします。

第 6 章　データ型

§6.1 Fortran のデータ型概要

1. 型とは何か

ビット情報

計算機の中核部分は、電気的には二種類の TTL レベルの電圧を保持した素子の集合体であり、その二種類の電圧を、それぞれ [0]、[1] のデジタル信号として処理している。 すなわち素子一つについて、[0] か [1] かどちらかの情報を持つ。この素子の持つ情報を、1 **ビット** の情報と言う。

このように元々の情報としては 0 と 1 の二種類しかないが、この情報をどのように解釈するかによって、いろいろな意味を持たせることができる。 この解釈の方法を指定するのが**データ型** (*type*) である。

型による解釈の例

8 ビットのことを 1 **バイト** と言う。 通常の計算機はバイト単位で処理を行っていて、メモリには 1 バイトごとに**アドレス** という通番が振られている。 また、レジスタのビッド数は通常 8×2^n となっている。

いま、32 ビットのレジスタ*[1]に、以下のような信号が保持されているとする。ここで、右側が下位のビット、左側が上位のビットである。 また、8 ビット (=1 バイト) ごとに「|」でバイト区切りを示している。

```
01100100011000110110001001100001
|       |       |       |       |
```

この 32 個のビット・パターンは、

・整数型であると見なせば、→ `1684234849` という整数を表している。

・実数型であると見なせば、→ `0.167780E+23` という実数を表している。

・文字型であると見なせば、→ `"abcd"`という文字列を表している。

・論理型であると見なせば、→ `.true.` すなわち「真」を表している。

と、それぞれ解釈される。

なお、どのようなデータ型があるかは、プログラミング言語により異なる。

2. Fortran のデータ型の種類

Fortran のデータ型は、まずコンパイラの提供する**基本型** と、基本型を組み合わせてユーザが定義する**派生型** の二つに分けることができる。

・基本型は、数値を扱うものとそれ以外のものの二つに大別される。前者を総称して**数値型** 、後者を**非数値型** という。

　・数値型はまた、**整数型** と **実数型** の二つに分けられる。

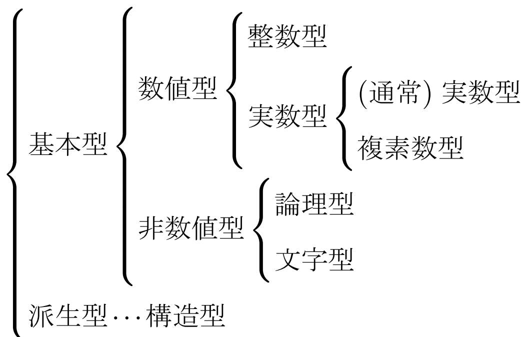
　　・実数型には、通常の実数型と**複素数型** の二つの sub type がある。

　・非数値型には、**論理型** と **文字型** がある。

・派生型は**構造型** といい、ユーザの定義によりその形態はさまざまである。

これらの関係を図で表すと、次のようになる。

*[1] CPU(中央演算処理装置) のさらに核心部分、演算を行う高速メモリのこと。



3. 種別

sub type

基本型の中には、バイト数の異なる、複数の sub type を持つものがある。一般にバイト数は 2 のベキ乗であり、バイト数が多いほど扱える数の範囲が広がる。または精度が向上する。その一方で、使用するメモリの量は増加する。また、計算時間がかかることもある。

たとえば整数型であれば、次のような sub type が一般的であり、それぞれ扱える整数の範囲が異なる。

・1 バイト整数型 · · ·　-128 から 127 まで
・2 バイト整数型 · · ·　-32768 から 32767 まで
・4 バイト整数型 · · ·　-2147483648 から 2147483647 まで
・8 バイト整数型 · · ·　-9223372036854775808 から 9223372036854775807 まで

ただし、どの sub type が扱えるかは処理系によって決まり、どのコンパイラでも上記すべてが利用できる、というわけではない。

このような sub type のことを、**種別** (*kind*) という。

基本種別と拡張種別

宣言文で単に型名だけを宣言した場合 (ex.　`integer ::`)、どの種別が選択されるかは、処理系によって基本型ごとにあらかじめ定められている。そのような種別を**基本種別** という。

基本種別以外の種別を、**拡張種別** という。拡張種別を使うためには、宣言文で型名と同時に**種別選択子**を指定しなければならない。

種別選択子

種別選択子については、Fortran90/95 では正の整数でなければならない、と規定しているだけで具体的な値は規格にない。したがって同じものに対しても、処理系によって異なった数が与えられている。

そこで拡張種別を使うには、まずその処理系で、求める種別にどのような種別選択子が割り当てられているかを知らなければならない。さらにいえば、特定の処理系のみで動作すればよいのならば、マニュアルを見ればよいが、処理系によらず動作をする可搬性のあるプログラムを作成するならば、プログラム上で以下の種別関数を使ってその選択子の値を取得しなければならない。

まず、変数 `a` の種別選択子の値 `k` は、次の種別関数 `kind` により求められる。

```
k = kind(a)
```

基本種別の種別選択子の値は、これにより求められる。次に拡張種別については、種別選択子の値が必要なのは、実際には整数型の扱える整数値の範囲と、実数型の精度についてである。これらについては、それぞれ対応する種別の種別選択子の値を求める関数が用意されているので、それぞれの項で説明する。

4. 変数・定数の型

Fortran で使用するすべての変数および定数は、いずれかの型 (種別を含む) を持たなければならない。

変数の型宣言

変数については、型はユーザが型宣言文 (*type declaration statement*) により指定する。基本型の変数に対する、Fortran90/95 の型宣言文の一般形は、次の形である (派生型については後述)。

```
型指定子 [, 属性指定子 1][, 属性指定子 2]・・・ :: データ要素宣言並び
```

・型指定子は、「型名 (kind=種別選択子の値)」である。ただし「kind=」は略すことができるので、「型名 (種別選択子の値)」でもよい。

型名は以下のキーワードで指定する。

```
integer          ... 整数型
real             ...(通常) 実数型
double precision... 倍精度実数型
complex          ... 複素数型
logical          ... 論理型
character        ... 文字型
```

・属性指定子は変数の属性を指定するもので、パラメータ属性、配列属性などがある。

・データ要素宣言は、「変数 = 初期値」の形の宣言を、コンマで区切って並べたものである。

拡張種別の使用宣言

拡張種別を使うためには、型指定子で種別選択子を指定しなければならない。しかしここで「kind=2」というように数値を指定すると、処理系によってこの数値は異なるので可搬性が失われてしまう。可搬性を保つためには、種別選択子の値を適当な名前付き定数に代入して宣言し、その名前付き定数で指定すればよい。たとえば、8 バイト整数を使うならば、まず 8 バイト整数の種別選択子の値を、主プログラムの先頭部分、あるいはモジュール宣言文 (この方が望ましい) の初期化式で、たとえば「i_8」という名前付き定数 (これ自体は基本整数型) に代入しておく、

```
integer, parameter :: i_8 = 8 バイト整数の種別選択子の値
```

そしてすべての 8 バイト整数型に対して、

```
integer(i_8) :: 変数名
```

という形式で宣言する。このプログラムを他の処理系に移植するときは、「i_8」の値を変えればよい。

定数の型

定数の型は、後述するようにその記述形式により決まる。たとえば「2」と記述すればこれは整数型定数であると解釈され、「2.0」と記述すれば実数型定数であると解釈される。

定数の種別

上の例のように、単に数値のみ記述したときは、種別は基本種別であると解釈される。拡張種別の定数を使いたいときは、数値部の終わりに「_種別選択子の値」を付加する。たとえば、8 バイト整数型変数「j」に 8 バイト整数型定数 20090101123000 を代入したいならば、まず変数と同様に適当な名前付き定数 (たとえば i_8) に種別選択子の値を入れておき、

```
j = 20090101123000_i_8
```

のように記述する。

§6.2 数値型

1. 整数型

整数型の種別

現在主流の 32 ビット OS 用のコンパイラでは、基本種別は主に 4 バイト (32 ビット) である。拡張種別としては、1 バイト、2 バイト、8 バイト整数型があり、これらのうちどれが使えるかは、処理系による。

一般に整数型では、OS のバイト数より大きいビット数の種別では計算は遅くなるが、バイト数より小さい種別で計算が速くなるわけではない。したがって、通常の計算は基本種別で行い、基本種別で扱えない大きな整数の場合のみ、バイト数の大きい拡張種別で行うことが合理的である。

2 バイト整数型は、PC の OS が 16 ビットであった時によく使われたので、そのプログラムとの整合のために使われる。また 1 バイト整数型は、通常の計算ではなく、後述するビット演算で使われる。

整数型の内部表現

次に、4 バイト整数型を例として、整数が計算機の内部でどのように表現されているかを示す。

・たとえば「11」をレジスタにロードすると、32 ビットの「0」「1」の列が、以下のように配置される。前の例と同様に、右側が下位のビット、左側が上位のビットである。

```
00000000000000000000000000001011
|       |       |       |       |
```

ここで「11」は 2 進数として表現されている。すなわち、最下位のビット、および 2 番目、4 番目のビットが「1」になっており、これは 2 進数で $2^3+2^1+2^0=11$ であることを表している。

・次に負の整数の表現を見る。 同じく 4 バイト整数型でので「-11」のレジスタ上の配置は、

```
11111111111111111111111111110101
|       |       |       |       |
```

となる。これは、符号を反転した「11」を加えたとき、レジスタの値が

```
00000000000000000000000000000000
|       |       |       |       |
```

すなわちゼロとなり、最上位のケタがあふれる (キャリー・フラグが立つ) ように決められている。このような形式で負の数を表現することを、**2 の補数表現** といい、ほぼすべての処理系で採用されている。2 の補数表現では最上位ビットは符号を表し、「0」のとき正またはゼロ、「1」のとき負である。

・この表現での最大値、および最小値は

```
01111111111111111111111111111111    最大値
|       |       |       |       |
10000000000000000000000000000000    最小値
|       |       |       |       |
```

と表現される。すなわち 最大値は $2^{31}-1$ であり、最小値は -2^{31} である。他の種別でも同様で、8 バイト整数型 n は、$2^{63}-1 \geqq n \geqq -2^{63}$ の範囲の整数を表すことができる。

整数型の種別関数

拡張種別を使うためには、その種別に対応する種別選択子の値を知らなければならないが、これは以下のようにプログラム中で求めることができる。すなわち整数型の場合、処理系の n ケタまで扱える整数型の種別のうちで、バイト数が最小である種別の種別選択子の値を i は、次の種別関数で求められる。

```
i = selected_int_kind(n)
```

たとえば n=2 の場合、1 バイト型整数で-99 から 99 までの値を扱うことができるので、もし処理系に 1 バイト型の整数があれば、その種別の種別選択子の値を返す。もしその処理系に 1 バイト型整数がなく、2 バイト型整数がバイト数最小の種別ならば、2 バイト型整数の種別選択子の値を返す。また n=5 ならば、-99999 から 99999、までを扱うことのできる、4 バイト型整数の種別選択子の値を返す。

さらにこの種別関数は、宣言文の初期化式の中で呼び出すことができる。たとえば「i_8」を 8 バイト整数型の、その処理系での種別選択子の値を持つ名前付き定数としたいならば、[*2]

```
integer, parameter :: i_8 = selected_int_kind(18)
```

という宣言文を、主プログラムの先頭部分かモジュール宣言部に置く。以後、8 バイト整数型の変数が必要ならば、

```
integer(i_8) :: 変数名
```

として宣言すればよい。

整定数

整数型の基本種別は、「**符号**」「**整数**」の形式である。正数の「+」記号は省略できる。

```
12    0    -45    +5
```

このとき、小数点を記入してはならない。小数点があると、それは実数型と判断される。

拡張種別の整定数は、基本種別と同様な符号と整数部に引き続いてアンダーバーと種別選択子の値を書く。 たとえば 400000 と-5000 とを 8 バイト整数型の定数とするならば、上で定義した「i_8」を使い、

```
400000_i_8      -5000_i_8
```

という形式で記述する。

符号なし整数型 (ビット型)

Fortran90/95 では、ビット操作関数やビット操作サブルーチンが、組み込み手続きとして導入され、ビットごとの演算が標準で可能になった。ただし、その対象は整数型に限られる。そこでビット操作を行うための整数型の変数や定数を、ここでは**符号なし整数型** 、あるいは**ビット型** と呼ぶことにする。

「符号なし整数型」の変数というものが Fortran にあるわけではない。形式上は通常の整数型の変数であり、通常の整数型として宣言する。

符号なし整数型の定数は、data 文で 2 進 (binary)、8 進 (octal)、16 進 (zone) のいずれかで定義する。[*3]その形式は、それぞれの頭文字、b、o、z に続けて、対応する表現を「’」でくくって表す。[*4]

[*2] この名前付き定数の名前には、他の変数名として使われそうもないものがよい。
[*3] 多くの処理系では、代入文あるいは宣言文の初期化式でも定義が可能である。
[*4] ftn fortran では、最上位ビットが 1 の符号なし整数は、デフォルトでは使えない。

```
data a/b'101'/    ! 2 進定数
data b/o'11'/     ! 8 進定数
data c/x'ff00'/   ! 16 進定数
```

ビット操作関数・サブルーチン

符号なし整数に対しては、通常の演算は行わず、ビット操作関数およびビット操作サブルーチンにより演算を行う。以下にそれらの手続きを示す。ここで引数は全部整数型で、i と j は同じ種別でなければならない。 またビットの順番は、最下位のビットを 0 として、上位に向かって数えていく。

○ bit_size(i) ··· i のビット数を返す。

○ not(i) ··· i のビットごとの否定 (0 ならば 1、1 ならば 0) を返す。

○ iand(i,j) ··· i,j のビットごとの論理積 (1 と 1 のとき 1、それ以外は 0) をとる。

○ ior(i,j) ··· i,j のビットごとの論理和 (0 と 0 のとき 0、それ以外は 1) をとる。

○ ieor(i,j) ··· i,j のビットごとの排他的論理和 (1 と 0、または 0 と 1 のとき 1、それ以外は 0) をとる。

○ ibclr(i,k) ··· i の k 番目のビットを 0 にする。

○ ibset(i,k) ··· i の k 番目のビットを 1 にする。

○ ibits(i,k,m) ··· i の k 番目のビットから m 個のビットを最下位まで移動し、残りのビットを 0 にする。

○ ishift(i,k) ··· i の全ビットを k>0 なら上位方向へ k、k<0 なら下位方向へ-k 移動させる。あふれたビットは捨てられ、反対側に空いたビットには 0 が詰められる。

○ ishiftc(i,k,m) ··· i の最下位から m 個のビットを、k だけ循環移動させる。移動の仕方は ishift と同じ、あふれたビットは反対側から詰められる。

○ btest(i,k) ··· i の k 番目のビットから 1 なら「真」、0 なら「偽」を返す。

○ subroutine mvbits(i,k,m,j,n) ··· i の k 番目から m 個のビットを、j の n 番目から m 個のビットにコピーする。 ビット移動は、変数 i,j のビット数以内に収まっていなければならない。

演 習 問 題

演習問題 1

次のプログラムを実行して、各自が今使用中の処理系における、それぞれの種別の種別選択子の値を調べなさい。なお、対応する種別が処理系にない場合には、関数は戻り値として-1 を返します。

```
program kind_of_integer
    implicit none

    write(*,*) "integer"
    write(*,*) " 1 byte", selected_int_kind(2)
    write(*,*) " 2 byte", selected_int_kind(4)
    write(*,*) " 4 byte", selected_int_kind(9)
    write(*,*) " 8 byte", selected_int_kind(18)

    stop
end program kind_of_integer
```

2. 実数型

実数型の種別

32 ビット OS 用のコンパイラでは、基本種別は通常 4 バイト (32 ビット) である。拡張種別としては、8 バイトの実数型がある。またそれ以上のバイト数の実数型が用意されている処理系もある。

整数型と異なるのは、現在の上位クラスの PC 用の CPU には、8 バイト (64 ビット) の実数計算を行うための専用のハードウェアが実装されており、8 バイト実数型の演算速度は 4 バイト実数型の演算速度と大きく変わらないことである。*5 それにもかかわらず、精度は 4 バイト実数型の 10 進 6 ケタに対して、8 バイト実数型は 10 進 15 ケタに達する。これらを考慮すると、4 バイト実数型よりは 8 バイト実数型を使う方が合理的である。

なお、8 バイトを超える実数型は、処理系がその型に対する数値演算ハードウェアを実装している場合を除き、大幅に速度が低下するので一般には実用的でない。また、そのようなハードウェアを実装している場合でも、可搬性が著しく低下するので、特別な場合を除き使用は避けるべきである。

倍精度実数型

上記の理由により、現在では CPU 能力と計算精度を考慮した場合、基本種別を 4 バイト実数型とするよりも、8 バイト実数型とした方が、本来は望ましいはずである。それにもかかわらず、4 バイト実数型を基本種別とする理由は、Fortran の規格の要求による。すなわち Fortran90/95 の規格では、sub type は各基本型について一つ以上あればよいのだが、実数型は例外であり、基本種別に加え、その倍のバイト数を占める拡張種別の二種類以上なければならないとされているからである。

これは主に歴史的な理由による。8 バイトの数値演算ハードウェアが一般的でない段階では、演算は速いが精度は悪い 4 バイト実数型と、演算は遅いが精度はよい 8 ビット実数型とを、場面に応じて使い分けることが合理的であった。その当時に作成されたプログラムと上位互換性を保つために、二種類の種別が要求された、と考えられる。

この基本種別の倍のバイト長を持つ拡張種別を、**倍精度実数型** と呼んで、基本型の一つのように扱う。型宣言での型名は、「`double precision`」である。なお、倍精度に対して、基本実数型を単精度ということがある。

ただし、倍精度型は Fortran90/95 では必ずしも必要ではない。多くの処理系では、コンパイラ・オプションにより基本種別を 8 バイトに変更できるので、そちらの方を使うことを推奨する。

実数型の内部表現

実数型の内部表現は、整数型とは異なり Fortran の規格が緩く、処理系により複数のパターンがある。しかしそれでは互換性において不便なので、最近は IEEE の規格に合わせた形式をとるコンパイラが多い。そこで、その規格について説明する。

IEEE の実数表現は、4 バイトの `S_floating`、8 バイトの `T_floating`、16 バイトの `X_floating` の 3 種類がある。このうち 4 バイトの `S_floating` は、ゼロでない実数について、次の式で表現する。

$$s \times 2^e \times \left(\frac{1}{2} + \sum_{k=2}^{p} f_k \times 2^{-k} \right)$$

・ここで、s は符号部で「`+1`」か「`-1`」のどちらかである。

・e は指数部で、2 のベキを表す。

・次の項は p ケタの 2 進小数部分で、f_k はその小数第 k ケタである。すなわち 2 進小数で書けば「$0.1f_2f_3\cdots f_p$」となる。ここで、小数第 1 位が常に 1 になるように、e の値を決めている。

*5 AMD の CPU では、64 ビット OS であればむしろわずかながら 8 バイト実数型の方が速い。

具体的なビットパターンは、以下のように配置される。

```
01000001000100000000000000000000
||      |                      |
S   E            F
```

・最初の 1 ビットが符号 s に対応し、「0」ならば正 (またはゼロ)、「1」ならば負を表す。
・次の 8 ビットが指数部に対応する。これを 1 バイトのビット型整数と見なすと、0 から 255 までの値をとることができるが、このうち 0 すなわち「00000000」の並びは、この並びが表す実数がゼロであることに対応する (ゼロの場合は、すべてのビットが「0」である)。また 255 すなわち「11111111」の並びは使われない。

残る 1 から 254 までの数から、126 を引いたものが、e の値となる。すなわち e は、-125 から +128 までの値をとることができる。$2^{-125} = 2.35 \times 10^{-38}$ であるから、ほぼ 10^{-37} から 10^{37} までの絶対値の数を表せることになる。
・最後部の 23 ビットが、2 進小数の小数第 2 位から小数第 24 位までの値である。この精度は、2 進での有効ケタが 23 だから、10 進での有効ケタは $23 \times \log_{10} 2 = 6.923$ となる。

これらより、例として図で表されたビットパターンに対応する実数を求める。まず s は「0」だから正である。次の 8 ビットはビット型整数として解釈すると 132 であるから、e の値は「$132 - 126 = 6$」である。また小数部は第 3 位のみが 1 であるから、省略された第 1 位の 1 を加えて「0.101」である。

これより図のビットパターンの表す数値は、$+2^6 \times \left(\frac{1}{2} + \frac{1}{2^3}\right) = 64 \times \frac{5}{8} = 40$ となる。

8 バイト実数型の T_floating では、S は 1 ビット、E の部分が 11 ビット、F の部分が 52 ビットとなる。この場合の精度は $52 \times \log_{10} 2 = 15.6$ ケタ、表せる絶対値は 10^{-307} から 10^{307} までである。

16 バイト実数型の X_floating では、S は 1 ビット、E の部分が 15 ビット、F の部分が 112 ビットとなる。この場合の精度は $112 \times \log_{10} 2 = 33.7$ ケタとなる。

実数型の種別関数

実数型の場合、処理系の 10 進で m ケタ以上の精度がある実数型の種別のうちで、バイト数が最小である種別の種別選択子の値を r とすると、r は以下の種別関数で求められる。

```
r = selected_real_kind(m)
```

上の記述より、6、13、33 を値を m として与えれば、4 バイト、8 バイト、16 バイトそれぞれの種別の種別選択子の値が取得できる。

たとえば、「r_8」を 8 バイト実数型の、その処理系での種別選択子の値を持つ名前付き定数としたいならば、整数の場合と同様に、

```
integer, parameter :: r_8 = selected_real_kind(13)
```

という宣言文を、主プログラムの先頭部分かモジュール宣言部に置く。以後、8 バイト実数型の変数が必要ならば、

```
real(r_8) :: 変数名
```

として宣言すればよい。

実定数

実数型の定数は、固定小数点表現と指数表現の 2 通りで定義できる。このうち基本種別に関しては、すでに §2.2 で示した。

拡張種別の固定小数点表現では、整定数と同様にアンダーバーの後に種別選択子の値を指定する。[*6]

`1.2_r_8`　　`-2._r_16`

拡張種別の指数表現では、基本種別の「e」の代わりに、倍精度型なら「d」、4 倍精度型なら「q」を使用する。

`1.2d3`　　`-.2d-2`　　`1d0`　　`-2q-3`

また、倍精度型の種別選択子の値を知りたければ、次の式で `d` に倍精度の種別選択子の値が代入される。

```
integer, parameter :: d = kind(1d0)
```

演 習 問 題

演習問題 2

次のプログラムを実行して、各自が今使用中の処理系における、それぞれの種別の種別選択子の値を調べなさい。なお、対応する種別が処理系にない場合には、関数は`-1` を返します。

```
program kind_of_real
    implicit none

    write(*,*) "real"
    write(*,*) " 4 byte",selected_real_kind(6)
    write(*,*) " 8 byte",selected_real_kind(15)
    write(*,*) "16 byte",selected_real_kind(20)

    write(*,*) "single precision", kind(1e0)
    write(*,*) "double precision", kind(1d0)

    stop
end program kind_of_real
```

演習問題 3

8 バイト実数型変数 `y` に、`0.1` を格納する場合、

(1) 基本実数型定数で代入する。

(2) 倍精度実数型定数で代入する。

(3) `read` 文で読み込む。

の 3 つの方法で行い、結果を出力して比較しなさい。

解答

(1) `0.100000001490116`

(2) `0.100000000000000`

(3) `0.100000000000000`

(1) は単精度、(2) と (3) は一致して倍精度となる。

[*6] ただし、この表現は紛らわしいので、あまり使われることはない。

3. 複素数型

Fortran では複素数型があり、複素変数や複素定数、複素数の四則演算を実数と同様に扱うことができる。他の言語で複素数型を扱えるものは少なく、Fortran が科学技術計算用のプログラミング言語といわれる理由の一つである。

複素数型の種別

一般に任意の複素数 c は、a,b を実数、i を虚数単位として、$c=a+bi$ の形式で表すことができる。ここで a を実 (数) 部、b を虚 (数) 部という。Fortran の複素数型は、この a,b の組により複素数をあらわす。ここで a,b は同じ種別の実数型であり、この実数の種別が、複素数型の種別となる。すなわち、複素数型の基本種別は、実数型の基本種別と同一であり、4 バイト複素数型、8 バイト複素数型があり、処理系によっては 16 バイト複素数型もある。32 ビット OS 用のコンパイラでは、通常 4 バイト (=32 ビット) である。

複素数型の内部表現

複素数型の内部表現は、連続した記憶領域上に実数部を表す実数型と、虚数部を表す実数型とを 2 個並べて表したものである。

実数部	虚数部

たとえば 4 バイト複素数型の記憶領域上に占めるバイト数は、4 バイト $\times 2 = 8$ バイトである。

複素数型の変数

複素数型の変数を使用するためには、型名は「complex」で宣言する。拡張種別を使いたいときは、実数型の種別選択子の値を使い、たとえば以下のように宣言する。

```
complex(r_8) :: 変数名
```

複素数型の定数

基本種別の複素数型変数 c に定数 $1+2i$ を代入するときは、

```
c = (1.0, 2.0)
```

のように、実数型定数を、実部、虚部の順に「,」で区切り、( ) でくくって代入する。 ただしこの書き方は実数型定数だけに限られ、$c=a+bi$ と実数型変数を代入するときには c = (a, b) とは書けず、

```
c = a*(1., 0.) + b*(0., 1.)
```

のように代入するが、あるいは次節に述べる組み込み関数 cmplx を使って代入しなければならない。

拡張種別の複素数型の場合も、対応する種別の実数型定数を同様に記述する。

```
c8 = (1.0_r_8, 2.0_r_8)
```

このとき「c8=(1.0,2.0)_r_8」のように書くと、いったん基本種別で複素数化されてから拡張種別に換算されるので。精度が落ちる。

複素数型の入出力

複素数型変数 c を入力するときは、実部、虚部の順に、二つの同種別の実数型定数を連続して入力する。たとえば、複素数型変数 c に対して、

```
read(*,*) c
```

に対して、 3.0 -1.0 と入力すれば、c には $3-i$ が代入される。

出力の際には、write(*,*) c により、複素定数型の出力がなされる。

複素数型の引数に対する数学関数の値

複素数 c が、実部 + 虚部形式で $c = a + bi$、極形式で $c = re^{i\theta}$ と表せるとする。

○ `abs(c)`

c の絶対値、$r = \sqrt{a^2 + b^2}$ が返される。

○ `sqrt(c)`

一般に複素数 c に対して、$\omega^2 = c$ の複素数根 ω は 2 個あり、それらは $\omega_1 = -\omega_2$ の関係にある。`sqrt` の値としては、そのうち偏角 ϕ が $-\pi/2 < \phi \leqq \pi/2$ の方が選択される。

○ `exp(c)`

$e^{a+bi} = e^a e^{ib} = e^a(\cos b + i \sin b) = e^a \cos b + i e^a \sin b$

○ `log(c)`

$\log re^{\theta i} = \log r + \log e^{\theta i} = \log r + \theta i$ 　（$\theta =$ `atan2(b,a)`）

○ `sin(c)`

$\sin c = \dfrac{e^{ic} - e^{-ic}}{2i}$ ここで、$e^{ic} = e^{-b+ia} = e^{-b} \cos a + i e^{-b} \sin a$、同様に $e^{-ic} = e^{b-ia} = e^b \cos a - i e^b \sin a$

したがって $\sin c = \dfrac{e^{-b} - e^b}{2i} \cos a + i \dfrac{e^{-b} + e^b}{2i} \sin a = i \sinh b \cos a + \cosh b \sin a$

○ `cos(c)`

`sin(c)` と同様に計算して、$\sin c = \dfrac{e^{ic} + e^{-ic}}{2} = \cosh b \cos a - i \sinh b \sin a$

演習問題

演習問題 4

次の Fortran の数式の値を答えなさい。

(1) `mod(3.0d0,1.2d0)`

(2) `mod(-3.0d0,1.2d0)`

(3) `atan(-1.0)`

(4) `atan2(1.0,-1.0)`

(5) `abs((1.0,-1.0))`

(6) `sqrt((0.0,-1.0))`

(7) `exp((0.0,-3.141593))`

(8) `log((-1.0,0.0))`

解答

(1) 0.6　　(2) -0.6　　(3) $-\dfrac{\pi}{4}$　　(4) $\dfrac{3}{4}\pi$

(5) $\sqrt{1^2 + 1^2} = \sqrt{2}$

(6) $\sqrt{e^{\frac{-\pi i}{2}}} = e^{\frac{-\pi i}{4}} = (\dfrac{\sqrt{2}}{2}, -\dfrac{\sqrt{2}}{2})$

(7) $e^0 \cos(-\pi) + e^0 \sin(-\pi) = (-1.0, 0.0)$

(8) $\log e^{\pi i} = \log 1 + \pi i = (0.0, \pi)$

Fortran での計算結果は、打ち切り誤差のために、必ずしも上の値とは一致しない。

4. 型変換

数値型を他の数値型に変換するには、組み込み関数を使う方法と、直接代入文で変換する方法がある。

組み込み関数による型変換

(1) 整数化

○ `int(x[,k])`　`!` x は整数式、実数式、複素数式

・引数の少数部分が切り捨てられて整数化される。 複素数型の場合は、実数部の小数部分が切り捨てられて整数化される。 k は、関数値の整数の種別を指定する。

・「`,k`」の部分は省略することができ、その場合関数値は基本整数型となる。(以下同じ)

・整数型を引数にして種別を変換することもできる。

○ `nint(x[,k])`　`!` x は実数式

・引数の少数部分が四捨五入されて整数化される。 規程では整数型、複素数型を引数にはできない。

(2) 実数化

○ `real(x[,k])`　`!` x は整数式、実数式、複素数式

・複素数が引数の場合、実部が (種別選択子の値が k の) 実数型に変換される。

○ `dble(x)`　`!` x は整数式、実数式、複素数式

・`real(x,kind(1d0))` と同じ、引数を倍精度実数型に変換する。

○ `aimag(x)`　`!` x は複素数式

・複素数の虚部を実数化する。

(3) 複素数化

○ `cmplx(x,y[,k])`　`!` x, y は実数式　または x が複素数式、y は省略

・x を実部、x を虚部として複素数化する。虚部が 0 のとき、y 以下は省略できる。

・複素数同士の種別変換のとき、x は変換元の複素数で、y は省略する。

・y を省略する場合、k の指定は、「`kind=k`」という形でなければならない。

(3′) 共役化

○ `conjg(x)`　`!` x は複素数式

・複素数 x の複素共役数を返す関数。$x = a + bi$ ならば、戻り値は $a - bi$ になる。

代入文による変換

代入文、

```
y = x  ! x は数値式
```

という構文において、

○ y が種別 k の整数型の場合、 `y = int(x,k)` が計算されて y に代入される。

○ y が種別 k の実数型の場合、 `y = real(x,k)` が計算されて y に代入される。

○ y が倍精度実数型の場合、 `y = dble(x,k)` が計算されて y に代入される。

○ y が種別 k の複素数型で、x が整数型または実数型の場合、 `y = cmplx(x,0,k)` が計算されて y に代入される。

また、これらの変換の際に

○ 同じ型の、バイト数の少ない種別に変換する場合、余ったビットは捨てられる。

○ 同じ型の、バイト数の多い種別に変換する場合、不足のビットは「0」で埋められる。

5. 混合演算

一つの計算式の中で、異なる型や種別の数値型を混在させることができる。結果は以下の規則に従う。

(1) 型についての規則

○ すべての変数、定数が一つの型 (整数型、実数型、複素数型) のときは、結果もその型になる。

```
7/3 -> 2          5.0 - 2.0 -> 3.0
```

○ 整数型と、実数型が混在していれば、結果は実数型になる。

```
5/2.0 -> 2.5
```

○ 実数型、複素数型が混在していれば、結果は複素数型になる。

```
(1.0,2.0)*2 -> (2.0,4.0)
```

○ 整数型と複素数型は、本来規則では混在できないが、ほとんどの処理系ではコンパイル・実行でき、結果は複素数型になる。

○ ベキ乗 `x**y` は、`exp(y*log(x))` を計算するので、本来 `x<=0` は許されないが、`y` が整数型の時は、`x` を `y` 回掛けるという計算を行うので許可される。このとき `y<=0` ならば、`x**abs(y)` の逆数となる。

(2) 種別についての規則

一つの式中で、結果の型の複数の種別をもつ変数、定数が混在するときは、結果の種別はその中でもっともバイト数の大きいものになる。このとき複素数型は、実数の一種と見なして、その実部・虚部を構成する実数の種別で比較する。[*7]

```
1.0 + 2.0d0 -> 3.0d0
```

演 習 問 題

演習問題 5

`n` が整数型、`a` が実数型、`c` が複素数型の変数とするとき、次の式の左辺の変数の値は、それぞれどうなるか。

(1) `n = -7/3`

(2) `n = -1.3`

(3) `a = (-1.0,1.0)`

(4) `c = 5/2`

解答

(1) `-2`　(2) `-1`　(3) `-1.0`　(4) `(2.0,0.0)`

演習問題 6

`k=kind(1d0)` とするとき、次の式の結果の値と種別を出力しなさい。(種別は `kind` 関数を使う)。

(1) `5.0/2.0_k`

(2) `5/2.0_k`

(3) `(5.0_k,0.0_k)/2.0`

(4) `(5.0,2.0)/2`

解答

(1) `2.5_k`　(2) `2.5_k`　(3) `(2.5,0.0)_k`　(5) `(2.5,1.0)_k`

[*7] ただし、`0.1+0.2d0` の値は `0.3d0` とはならない。これは `0,1` が単精度しかなく、すでに倍精度より大きな誤差を含んでいるからである。もっとも `0.1d0+0.2d0` も、厳密に言えば `0.3d0` となるとは限らない

§ 6.3 非数値型

1. 論理型

論理型変数とは、「真」あるいは「偽」の、どちらかの値をとる変数のことである。論理型変数に対して演算が定義できて、これを論理演算 (ブール代数) というが、その式の値も「真」あるいは「偽」の、どちらかの値をとる。

論理型の種別

32 ビット OS 用のコンパイラでは、基本種別は通常 4 バイト (32 ビット) である。しかし 2 通り、すなわち 1 ビットで十分な情報を、32 ビットも使って表すのはメモリのムダなので、1 バイトあるいは 2 バイトの論理型が用意されている処理系もある。*8

論理型の種別変換は、組み込み関数 `logical(b[,k])` で行う。ここで `b` は論理式、`k` は変換先の種別選択子の値であり、基本種別の場合は省略できる。

論理型の内部表現

論理型は、すべてのビットが 0 の場合「偽」、それ以外は「真」とみなされる。「真」を論理型変数に代入した場合の内部表現は処理系によって異なるが、すべてのビットを 1 にすることが多い。

論理型の変数

型名は「`logical`」である。

```
logical :: 変数名
```

論理型の定数

論理型の定数は、真を「`.true.`」、偽を「`.false.`」と表す。ex.「`logical :: p = .true.`」

論理演算

論理型の演算子には、次の 5 種類がある、`p1,p2` を論理型の変数、定数、あるいは式として、

- 「`.not. p1`」 ··· 否定　演算優先順位 1
- 「`p1 .and. p2`」 ··· 論理積　演算優先順位 2
- 「`p1 .or. p2`」 ··· 論理和　演算優先順位 3
- 「`p1 .eqv. p2`」 ··· 論理等価　演算優先順位 4
- 「`p1 .neqv. p2`」 ··· 論理不等価　演算優先順位 4

演算優先順位は小さいほど先に評価される。すなわち否定が一番先に評価され、論理等価または論理不等価が最後に評価される。演算優先順位が同じ時は、原則的に左から評価される。

演 習 問 題

演習問題 7

論理学でいう包含関係「p ならば q である」は、記号で「$p \to q$」と表される論理式である。この「$\to$」にあたる論理演算は、BASIC では「`Imp`」という演算子が定義されているが、Fortran では定義されていない。いま、「p」「q」を論理型変数とするとき、「$p \to q$」と同値の論理式を Fortran で記述しなさい。

解答

「p ならば q である」は「p であって q でない、ということはない」すなわち「p でないか、あるいは q である」と同値なので、論理記号では「$(\sim p) \vee q$」と表される。これを Fortran で表せば、

「`(.not.p) .or. q`」 となる。なお、「()」はわかりやすくするために加えたもので、なくてもよい。

*8 ただし、メモリは節約できるが、計算時間が増加するおそれがあるので、巨大な論理型配列を使わない限り、基本種別でよい。

2. 文字型

文字型はその名の通り文字を扱う型である。ただし Fortran では (C の char 型のように)1 文字だけを独立して扱うことはなく、常に文字列として扱う。この点で配列に似ているが、似て非なるものである。*9

文字型の種別

大部分の処理系では、文字型の種別は基本種別の 1 バイトのみである。実際、拡張種別があっても可搬性に乏しくきわめて使いづらい。それゆえに、文字型はすべて基本種別で使用するので「kind=」という記述は特に使う必要性はない。

文字型の内部表現

文字型のビット配列と文字との対応を**文字コード**という。文字コードについては、データ解析では特に重要なので、次節に独立させて扱うことにする。

文字型変数

文字型の型名は「character」であるが、型指定子は文字型のみ他の型と異なり、「len=n[,kind=k]」という形式をとる。「len=n」の部分は文字長、すなわち文字変数の長さである。

```
character(len=n[,kind=k]) :: 文字型変数名
```

ここで、「kind=k」の部分は上記の理由で省略される。また「len=」の部分も省略できる。たとえば文字長 10 の文字型変数 moji を宣言するならば、次のように簡略に記述できる。*10

```
character(10) :: moji
```

このように、Fortran の文字変数は「文字長」というパラメータを保持し、その長さは宣言文であらかじめ決められる (固定長という)。これは文字長が可変 (不定長) である BASIC や C などとは異なっている。

また、文字型変数名に文字パラメータ選択子をつけ、「moji*5」のように長さを指定することもでき、このときはこちらの方が型指定子よりも優先される。

```
character :: moji1*10, moji2*20
```

文字長を、どちらの形式でも指定しないときは、長さ 1 の文字長を指定したとみなされる。

どちらかというと、前者の書き方が望ましい。なぜなら、文字型にも配列が定義でき、たとえば、

```
character :: moji1(10), moji2(20)
```

は文字長 1 で寸法 10 の配列 moji1 と、文字長 1 で寸法 20 の配列 moji2 を宣言することを意味するが、これが後者の表現と紛らわしいからである。

文字型定数

文字型の定数は、シングル・クォーテーション「'」またはダブル・クォーテーション「"」で両端をくくって表す。これらの記号を定数に入れたいときには、2 個連続で書くか、もう一つの文字でくくる。

```
'abc'   "d-2"   "漢字"   "it's" (= 'it''s')
```

文字型演算

文字変数、文字定数、およびその連結演算子による結合を総称して、数式に対応して**文字式**という。文字式を扱う演算子は、文字式を連結する、連結演算子「//」のみである。たとえば、

```
"moji1" // '-' // "moji2"
```

は「 "moji1-moji2"」となる。

---

*9 この点は C の string 型と似ているが、string 型は char 型の配列そのものである点が異なる。

*10 FORTRAN77 との互換性を保つため、character*10 という形式での宣言も可能だが、廃止予定事項なので奨められない。

代入文

数式と同様に、等号「=」を使って、文字変数に文字式を代入することができる。

文字変数 = 文字式

このとき、双方の文字長により次のような処理がなされる。左辺の文字変数の文字長を n、右辺の文字式の文字長を m とすると、

(1)　n = m のとき、右辺の値がそのまま文字変数に代入される。
(2)　n < m のとき、右辺の文字式の先頭から n 文字が、文字変数に代入される。
(3)　n > m のとき、右辺の文字式が、文字変数の先頭から m 文字まで代入され、後ろの n-m ビットには空白が詰められる。

部分文字列

文字変数の一部分を取り出して、一つの文字変数のように扱うことができる。たとえば、文字長 n の文字変数 a の第 i 番目の文字から、第 j 番目の文字までの j-i+1 文字からなる文字列は、「:」を使って、

a(i:j)

で表される。これを**部分文字列** (*substring*) という。1 番目の文字から指定する場合は「1」は省略でき、最後の n 番目の文字まで指定する場合は「n」を省略できる。 i ≧ 1 かつ j ≦ n でない場合は、処理系に依存する。

このとき文字変数 a は、あたかも寸法 n の 1 バイト配列のように扱われる。ただし、do 三つ組、すなわち a(n1:n2:n3) のようにステップ n3 を指定することはできない (コンパイル・エラーになる)。

プログラム例 6-a　部分文字列

```
1   program substring
2       implicit none
3       character(10) :: c= "abcdefghij"
4       integer :: i, j
5
6       write(*,*) c(5:5)
7       write(*,*) c(:3)
8       write(*,*) c(6:)
9       i = 2 ; j = 8
10      write(*,*) c(i:j)
11      c(1:6) = c(3:8)  ! 部分文字列を作業領域に作成して、元の文字変数に代入される
12      write(*,*) c
13
14      stop
15  end program substring
```

上記プログラムの出力

```
e
abc
fghij
bcdefgh
cdefghghij
```

部分文字列を元の文字列に代入するときは、第 11 行のように、部分文字列はいったん取り出され、指定された位置に代入されて元の文字と置き換わる。

文字型を扱う組込み関数

文字型を扱う組み込み関数を、種類別に示す。以下 int は整数型の式、c は 1 文字の文字型、str は文字型の式とする。

(1) 文字コードに関するもの

○ char(int)　：整数式 int の処理系の文字コードに対応する 1 文字を返す。

○ ichar(c)　：文字 c の処理系の文字コードを、基本整数型で返す。

○ achar(int)　：整数式 int の ASCII 文字コードに対応する 1 文字を返す。[*11]

○ iachar(c)　：文字 c の ASCII コードを、基本整数型で返す。

(2) 文字長に関するもの

○ len(str)：str の文字長を返す。str の引数は文字配列も可能で、その場合は要素の文字長を返す。

○ len_trim(str)：str から末尾の空白を除いた文字長を返す。

(3) 文字列の操作に関するもの

○ trim(str)　：str の末尾の空白を削除した文字列を返す。

○ repeat(str,n)：str を `n(>=0)` 回繰り返した文字列を返す。文字型宣言文の初期化に使える。

○ adjustl(str)　：str の先頭の空白を取り除いて左詰めにし、除いた空白を末尾につける。

○ adjustr(str)　：str の末尾の空白を取り除いて右詰めにし、除いた空白を先頭につける。

(4) 文字列の検索に関するもの

○ index(str1,str2[,lb])　：文字列 str1 の中に、文字列 str2 が現れる位置を返す。現れないときは 0 になる。lb は論理式であり、.true. のとき文字列 str1 の後ろから検索する。

○ scan(str1,str3[,lb])　：文字列 str1 の中に、文字列 str3 の中に含まれる文字が初めて現れる位置を返す。str1 の文字がすべて str3 に含まれない文字であれば 0 になる。lb の働きは index と同じ。

○ verify(str1,str3[,lb])　：文字列 str1 の中に、文字列 str3 の中に含まれない文字が初めて現れる位置を返す。str1 の文字がすべて str3 に含まれれば 0 になる。lb の働きは index と同じ。

(5) 文字列の比較に関するもの

○ lgt(str1,str2)　：　`str1>str2` と同じ。 引数が両方とも長さ 0 のときは、戻り値は「真」になる。

○ lge(str1,str2)　：　`str1>=str2` と同じ。 引数が両方とも長さ 0 のときは、戻り値は「真」になる。

○ llt(str1,str2)　：　`str1<str2` と同じ。 引数が両方とも長さ 0 のときは、戻り値は「真」になる。

○ lle(str1,str2)　：　`str1<=str2` と同じ。 引数が両方とも長さ 0 のときは、戻り値は「真」になる。

文字列の大小

索引などを作るため、文字列には大小関係が定義されている。文字列 str1 と文字列 str2 の大小は、
(1) まず第 1 文字同士の文字コードを比較して、コードの大きい方が文字列の値が大きいとする。
(2) 第 1 文字が同じ場合は、第 2 文字同士の文字コードを比較して、同様に判定する。
(3) 一方の文字列が終了したならば、その文字列の方が小さいとする。

比較は数値型と同様に、比較演算子「`>,>=,<,<=,==,/=`」を使って行う。

[*11] PC Fortran のほとんどの処理系では、ASCII コードを使っているので、achar は char と同じ、iachar は ichar と同じであり、使う必要はない

3. 文字コード

1 バイト系コード

アルファベットの大小文字および特殊記号は、1 バイトで 1 文字を表す。1 バイトは 8 ビットであるから、$2^8 = 256$ 通りの文字をこれで表わすことができる。この文字に対応するビットパターンを文字コード、その対応方式を文字コード系と言うが、コード系は一通りではない。そのうちもっともよく使われているのは ASCII コードであり、一部に EBCDIC コードも使われている。*12

(1) ASCII コード

ASCII とは、American Standard Code for Information Interchange の略であり、パソコンからワークステーションまで幅広く使われているコード系である。ASCII コードは 16 進で $(00)_{16}$ から $(7E)_{16}$ までの、最上位ビットが 0 である 128 文字しか定義されていない。*13

ASCII のコード表を以下に示す。表は 16 進表示で、縦が上位のケタ、横が下位のケタを表す。 たとえば、$(41)_{16}$ というビットパターンに対応する文字は'A' であり、$(20)_{16}$ というビットパターンに対応する文字は、空白' ' である。

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
   -------------------------------------------------
 2|      !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
 3|   0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
 4|   @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
 5|   P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
 6|   `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
 7|   p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

この表に記されていない、$(00)_{16}$ から $(0F)_{16}$ まで、および $(7F)_{16}$ の計 17 コードは制御コード (制御文字) といい、Operating System で特別の意味を持つコードである（最上位ビットを 1 にした、対応する 17 コードも同様)。 たとえば、$(0A)_{16}$ は L/F（ラインフィード）といい、Linux では行の終わりを表すのに使われる。 これらは通常の文字としては扱われない。

また、$(10)_{16}$ から $(1F)_{16}$ までの 16 コードも、機種によりそれぞれ意味を持つコードであり、やはり通常の文字として扱われない。

(2) JIS コード

JIS コード系は日本だけで使われるコード系であり、ASCII コードの $(5C)_{16}$ の'\'（バックスラッシュ）を'¥' で置き換え、ASCII コードで定義されていない $(A0)_{16}$ から $(FE)_{16}$ までの部分に、半角カナなどを対応させたコードである。

日本語化された OS ならば使用可能ではあるが、国外では全く通用しないうえ、対応していないソフトも多いので、半角カナの使用は避けた方が無難である。

(3) EBCDIC

IBM 系の大型汎用計算機で使われた文字コードである。 現在はほとんど使われることはない。

(3′) EBCDIK

EBCDIC の使われていないビットパターンに、半角カナを対応させた日本限定のコードである。

---

*12 EBCDIC から ASCII へのコード変換は、Unix の dd コマンドで行う。たとえば、1 行 recl バイトの EBCDIC で書かれた infile というファイルを、ASCII コードに変換して outfile に出力するには、端末モードから次のように入力する。

```
dd conv=ascii,if=infile,of=outfile,ibs=recl
```

*13 最上位ビットはパリティ (偶奇性) のチェックに使われていた。

多バイト系コード

1 バイト系コードでは、最大 256 種類の文字しか表せないが、これでは漢字を表すには不足である。そこで、漢字や (全角) かな文字を表すために、複数のバイトからなるコードが使われる。ただしこれにはいくつもの種類があり、また相互に無関係に定義されているコードも多いためソフトウェアが対応しきれず、往々にしていわゆる「文字化け」を起こす。

(1) ISO2022JP コード系 (いわゆる JIS コード系)

JIS では漢字 1 文字に 2 バイトのコードを対応させる、漢字コード表を規定している。このコードはその漢字コード表を使っているが、そのまま使ったのでは処理系がデータが 1 バイトコード系なのか、2 バイトコード系なのかを判別できない。そこで漢字コードの前後に $(1B)_{16}$ [esc コード] から始まる識別コードを付け加えて、その識別コードに挟まれる部分が JIS 漢字コードであることを表示する。

このコード系では、n 文字の漢字を表すのに必要なバイト数は、2n+m となる。m の値は処理系によって決まる。たとえば Fedora-Linux で、JIS コード系での「漢字」という文字列の内部コードは、

$(1B)(24)(42)(34)(41)(3B)(7A)(1B)(28)(42)_{16}$ の 10 バイトであり、前後 3 バイトずつの識別コードを除いた、$(34)(41)(3B)(7A)_{16}$ の部分が JIS 漢字コードである。

このコード系は postscript の漢字コード、および mail-system のデフォルトコードとして使われる。

(2) Shift-JIS コード系

日本語 Windows の標準で使われているコード系である。

JIS コードをある規則に従って移動させ、1 バイト目を ASCII コードで定義されていない領域に割り振ったものが Shift-JIS コード系である。 処理系はそのコードを読み込むと、当該のバイトとその次のバイトを合わせて一つの文字を表していると解釈する。このため、n 文字の漢字を表すのに必要なバイト数は、2n バイトですむ。

Shift-JIS コード系での「漢字」の内部コードは、$(8A)(BF)(8E)(9A)_{16}$ の 4 バイトである。

(3) EUC コード系

UNIX マシンで使うために、Shift-JIS コード系と同じような考え方で ASCII コードを避け、全く新しく漢字コードを割り振ったものが EUC コード系である。n 文字の漢字を表すのに必要なバイト数は、やはり 2n バイトですむ。

EUC コード系での「漢字」の内部コードは、$(B4)(C1)(BB)(FA)_{16}$ の 4 バイトである。

(4) UNICODE コード系 (utf-8)

UNICODE にはいくつかのバージョンがあり、現在主に使われているのは utf-8 コード系である。日本語だけでなく、世界の主な文字を一つのコード系で表わすために作成されたコード系である。このためコード数が非常に多くなっており、n 文字の漢字を表すのに必要なバイト数は、3n バイトとなっている。

Linux 系のシステムでは、標準コード系を EUC から次第に UNICODE に移行しつつあり、日本語版 fedora も ver5 から utf-8 を標準コード系としている。[*14]

utf-8 コード系での「漢字」の内部コードは、$(E6)(BC)(A2)(E5)(AD)(97)_{16}$ の 6 バイトである。

漢字コードの相互変換

Windows システムでの変換は、適当なマルチコード対応のエディタで変換するのが簡単である。フリーのものとしては、「TeraPad」がある。

Linux システムでの変換は、「iconv」というコマンドで行う。ただし完全には対応し切れていない。

[*14] Windows は、現在各国の言語ごとにバージョンを出しているが、コスト削減のために UNICODE に変えようとしている。

§6.4 派生型

これまでに記述した、処理系によって用意された型のことを、**基本型** (*intrinsic type*) という。この基本型をユーザが組み合わせて作成した型のことを**派生型** (*derived type*) という。Fortran90/95 では構造型がこれにあたる。

1. 構造体

配列は同じ型、同じ種別のデータをひとまとめにして扱うものである。これに対して異なる型、異なる種別のデータをひとまとめにしたものが構造体である。たとえば試験の成績を集計するときに、一人分の氏名 (文字型)、それぞれの試験の点数 (整数型)、平均点 (実数型) をひとまとめにして扱うことができる。このとき、構造体を構成する個々の変数を、構造体の**成分** という。

ここでは以下の例題により、構造体の使い方を示すことにする。

例題 6-1

期末試験 3 科目での 6 人の学生の成績は、以下の通りでした。

```
arakawa   60 ,70 ,90
kurihara  72, 82, 78
manabe    80, 73, 75
miyakoda  60, 71, 68
ogura     85, 80, 82
yanai     75, 73, 78
```

これより各人の 3 科目の平均点を算出し、平均点の高い順に並べ替えて出力しなさい。
(並べ替えは、第 4 章の章末問題で既出)

構造型の定義

構造型は `type` 文で定義する。`type` 文と、`end type` 文の間に、各成分の定義を順番に宣言する。

```
type 構造型名
  各成分の宣言文
      ⋮
  各成分の宣言文
end type 構造型名
```

この順番には意味があり、各成分はこの順に並ぶ。なお、同じ型、種別の成分ならば、

```
integer :: i, j
```

と 1 行に書いても、`i`、`j` の順番に整列する。

例題の姓と 3 科目の点数を成分とする構造型を `examination` として定義するには、以下のように記述する。

```
type examination
  character(10) :: name
  integer       :: point(3)
end type examination
```

構造型の定義は、主プログラムのみの短いプログラムならば先頭の宣言部に記述するが、通常はモジュールの宣言部に記述して共通に参照する。

構造体変数の宣言

このように定義された構造型により、構造型の変数を宣言する。その形式は以下の通り、

```
type(構造型名) :: 構造体{変数名|配列名 (寸法宣言子)}
```

たとえば、上の examination 構造型を持つ構造体変数として、 results を定義するならば、

```
type(examination) :: results
```

のように宣言する。

構造型の入れ子

構造型の宣言の際に、成分として他の構造体を使うことができる。成分となる構造体の構造型宣言は、その文より前になければならない。*15

```
type record
  type(examination):: results
  real             :: average
end type record
```

構造体定数

構造体の定数は、以下ように構造体名に続けた「(　)」の中に、成分を順番に、最後まで「,」で区切って記述する。

```
構造型名 (成分 1, 成分 2,...)
```

注意しなければならないのは配列の場合である。配列はそれ一つで 1 成分であるから、配列あるいは配列構成子で記述しなければならず、配列要素をバラバラにして羅列してはいけない。たとえば「arakawa」君の成績を「examination」構造型の構造体定数として表せば、以下のようになる。

```
examination("arakawa",(/60,70,90/))
```

構造体成分の引用

構造体の成分は、構造体名と成分名の間に「%」をはさんで指定する。

```
構造体{変数名|配列要素}%{成分名|配列要素}
```

たとえば「record」型の構造体配列 personals の 2 番目の配列要素の、average 成分の値は、

```
personals(2)%average
```

で指定する。この形式で指定された成分からその値を引用でき、また指定された成分に対して代入を実行することができる。

構造型が入れ子になっている場合には、「%」を続けて指定する。たとえば、「record」型の構造体配列 personals の 3 番目の配列要素の、「examination」構造型 results 成分の、point 配列成分の 2 番目の配列要素の値は、以下のように指定する。

```
personals(3)%results%point(2)
```

*15 この場合、下の文の results 構造体は、他の場所で改めて宣言する必要はない。

例題 6-1 のプログラム例

```
1  program example6_01
2      implicit none
3
4      type examination
5        character(10)    :: name
6        integer          :: point(3)
7      end type examination
8
9      type record
10       type(examination):: results
11       real             :: average
12     end type record
13
14     integer, parameter :: n = 6
15     type(record)       :: personals(n), rw
16     integer            :: i, j, k
17
18     personals(1)%results = examination("arakawa", (/60,70,90/))
19     personals(2)%results = examination("kurihara",(/72,82,78/))
20     personals(3)%results = examination("manabe",  (/80,73,75/))
21     personals(4)%results = examination("miyakoda",(/60,71,68/))
22     personals(5)%results = examination("ogura",   (/85,80,82/))
23     personals(6)%results = examination("yanai",   (/75,73,78/))
24
25     do i =1, n
26       personals(i)%average = sum(personals(i)%results%point)/3.0
27     enddo
28
29     do i = 1, n
30       k=0
31       do j = 1, n-1
32         if ( personals(j)%average < personals(j+1)%average ) then
33            k = k + 1
34            ! ix(j) と ix(j+1) を交換 (swap)
35            rw = personals(j)
36            personals(j) = personals(j+1)
37            personals(j+1) = rw
38         endif
39       end do
40       if (k == 0) exit
41     end do
42
43     do i=1, n
44       write(*,*) personals(i)
45     enddo
46
47     stop
48 end program example6_01
```

§6.5 拡張データ型の引数

基本データ型以外の派生型 (構造型) および配列を併せて、ここでは**拡張データ型** あるいは拡張型と呼ぶことにする (これはこの冊子だけの名称で、一般的なものではない)。これらの拡張型をサブ・プログラムに引数として渡す場合には、基本型とは方式が異なる場合がある。

1. 引数が拡張データ型であるサブ・プログラム

構造型は、基本型と同様に実引数リスト、仮引数リストに記述できる。*16 その際、呼び出し側および呼び出される側の双方で (内部サブ・プログラムをのぞき) 構造型の定義が必要である。しかし 2 カ所に同一の定義を記述することは、煩わしいばかりでなく修正時のエラー要因となるので、通常はモジュール宣言部に記述しておいて、双方で参照を行う。

配列型も、その寸法が一定ならば、構造型と同様に呼び出し側、呼び出される側双方で配列宣言を行えば、基本型と同じく実引数リスト、仮引数リストに記述できる。このとき呼び出される側の仮引数である配列のことを **仮配列** という。これに対してこれまで扱ってきた仮引数以外の上 (下) 限が定められた配列のことを、**定数配列** という。

整合配列

問題は次の例題のように、配列の大きさが呼び出されるごとに異なる場合である。

例題 6-2

実数型の配列 x を引数として受け取り、その平均値を戻り値とする関数 ave_array を作成しなさい。ただし、配列の大きさは呼び出しごとに一定ではないものとします。

この場合呼び出される側で配列宣言を行うために、呼び出し側から配列の大きさを呼び出される側にやはり引数で渡せばよい。この場合の配列宣言の形式は (一次元配列の場合)、

```
型名 :: 仮配列名 ([下限値:] 上限値)
```

であり、これは定数配列の場合と同様であるが、上限値、下限値の指定が定数 (名前付き定数を含む) だけでなく、変数 (仮引数) でもよい、という点で異なる。このような仮配列を特に**整合配列** という。

例題 6-2 のプログラム例

```
1   program example6_02
2       implicit none
3       real :: a(5)=(/1.0,2.0,3.0,4.0,5.0/)
4       real :: b(4)=(/6.0,7.0,8.0,9.0/)
5
6       write(*,*) ave_array(a, size(a))
7       write(*,*) ave_array(b, size(b))
8
9       stop
10  contains
11
12    real function ave_array(x, n)
13      real, intent(in)    :: x(n)
```

*16 ただし、コンパイラによっては、基本型引数の後に拡張型引数を置かないとエラーを起こすことがあるようだ。とくに、省略可能な引数が混在する場合には注意が必要である。

```
14        integer, intent(in) :: n
15        integer             :: i
16
17        ave_array=0.0
18        do i =1, n
19          ave_array = ave_array + x(i)
20        enddo
21        ave_array = ave_array/real(n)
22        return
23      end function ave_array
24    end program example6_02
```

第 6 行と第 7 行で配列とともにその大きさを実引数として渡し、第 12 行でその値を変数 n として受け取り、第 13 行でその n を使って整合配列を宣言している。

形状引き継ぎ配列

整合配列のように上 (下) 限値を渡さず、配列の形状 (一次元、二次元等) だけを宣言することも可能である。これを **形状引き継ぎ配列** という。形状引き継ぎ配列の宣言は (一次元配列の場合) 次の形式である。

```
型名 :: 形状引き継ぎ配列名 (:)
```

整合配列との相違点は、

(i) 呼び出された側でわかるのは配列の大きさだけなので、下限値は引き渡せない。すなわち、下限値は常に 1 である。

(ii) 配列の添え字が、元の配列の範囲を超えた場合でも、警告されないことがある。

特に注意しなければならないのは (ii) の方である。これらの点で不安があるならば、整合配列を使った方がよい。

形状引き継ぎ配列を使うならば、上記のプログラム例の第 12 行から第 15 行を以下のように変更する。

例題 6-2 の修正

```
12      real function ave_array(x)
13        real, intent(in)    :: x(:)
14        integer             :: i, n
15        n = size(x)
```

文字長の一定でない文字型仮引数

仮引数が文字型であり、その文字長が呼び出しごとに変わるときも、配列と同様の宣言ができる。ただし、文字は常に一次元であり、その下限もつねに 1 なので、わざわざ文字長を別の仮引数として送るまでもない。そこで形状引き継ぎ配列に対応して、次のように宣言を行う。

```
character(*) :: 文字型仮引数名
```

呼び出された側からは、文字長は len(文字型仮引数名) で取得できる。

さらに文字長を len(文字型仮引数名) の形で、文字型を宣言することも可能である (章末問題)。

2. 基本型以外の関数値を返す関数

通常、数学で「関数」という場合は、数体から数体への写像を指す。すなわち関数 $f(x,y)$ では、x、y の定義域も、f の値域も数値である。

しかし、一般的な写像では、数体以外の集合が定義域にも値域にもなりうる。たとえば m 行 n 列の行列は、n 次元ベクトル空間から m 次元ベクトル空間への線形写像と同値である。Fortran の `function` もこれに似て、引数および関数値は基本データ型だけでなく、拡張データ型でも扱うことができる。[*17]

例題 6-3

寸法 3 の一次元実数型配列 `a`、`b` を引数として受け取り、その外積 `a × b` を返す、実数型の配列モジュール関数を作成しなさい。

このような場合、`function` 文の最後に戻り値を返す変数を、次の形式で `result()` の中に記す。これを **result 句** という。[*18]

```
function 関数名（引数リスト）result(戻り値の変数名)
```

そして型宣言はこの変数に対して行い、戻り値も関数名でなく、この変数に対して代入する。

例題 6-3 のプログラム例

```
1   module module1
2       implicit none
3   contains
4     function cross_product(a,b) result(cp)
5       real, intent(in) :: a(3), b(3)
6       real             :: cp(3)
7
8       cp(1) = a(2)*b(3)-a(3)*b(2)
9       cp(2) = a(3)*b(1)-a(1)*b(3)
10      cp(3) = a(1)*b(2)-a(2)*b(1)
11
12      return
13    end function cross_product
14  end module module1
15
16  program example6_03
17      use module1
18      implicit none
19      real :: a(3)=(/1.0,2.0,3.0/), b(3)=(/0.5,1.5,2.5/), c(3)
20
21      c = cross_product(a,b)
22      write(*,*) c
23      stop
24  end program example6_03
```

[*17] いうまでもないが、サブルーチンではこのような事態は起きない。

[*18] `result` 句自体は、基本型の関数でも同様に使用できる。そこで、基本型、拡張型にかかわらず、すべての戻り値は関数名でなく、`result` 句の変数で返すスタイルをとる人もいる。

§6.6 総称サブ・プログラム

1. 組込み総称関数

数学関数は一般に、決められた型および種別しか引数にできないが、abs のような関数は、複数の型・種別を引数にすることができる。その場合関数値は、引数の型・種別と同じものが返される。このような関数を **総称関数** という。*19 以下に総称関数の組込み関数を示す。

	整数型	基本実数型	倍精度型	複素数型
abs	○	○	○	○
mod, dim, modulo	○	○	○	—
sign	○	○	○	—
max, min	○	○	○	—
sqrt	—	○	○	○
exp, log	—	○	○	○
log10	—	○	○	—
sin, cos	—	○	○	○
asin, acos	—	○	○	—
tan, atan, atan2	—	○	○	—
sinh, cosh, tanh	—	○	○	—

引数が複数ある関数については、すべての引数は同じ型・種別でなければならない。

2. ユーザ定義総称サブ・プログラム

Fortran90/95 では、組込み関数だけでなく、ユーザの作成する関数、サブルーチンに対しても総称名での呼び出しが可能になった。データ解析では、これは特定の場合について大いに有効である。以下に例題を通してその使い方を解説する。

例題 6-4

ある地区で複数の風観測点の値から、風向の 16 方位別頻度分布図を作成することになった。
ところがこの地区の風向記録には、次の 3 種類の書式が混在している。
(a) 北北東から右回りに、1,2,3⋯16 と、16 方位の整数で表すもの。
(b) 0.0 から 360.0 まで、度単位で小数第 1 位までの実数で表すもの。
(c) 「NNE」「NE」「ENE」⋯ 「N」の 16 個の文字列で表すもの。
分布図の作成にあたって、これらを (a) の書式に統一したい。

(1) (a)(b)(c) それぞれの書式での風向を実引数で与えた場合に、関数値が (a) の書式で返されるような、モジュール関数 wdir_i(iw)、wdir_r(rw)、wdir_c(cw) をそれぞれ作成しなさい。

(2) 実引数 x に上のどの書式で風向を与えても、戻り値が (a) の書式で返るような、関数 wdir(x) を作成しなさい。

*19 FORTRAN77 までは上のような総称関数は規定されておらず、整数型ならば「IABS」、複素数型ならば「CABS」というように使い分けなければならなかった。これはプログラム作成者に余分な負担を強いていた。

(1) 作成にあたっては、最初からモジュール関数としても作成しても、最初は内部関数として作成し、完成したらモジュール化してもよい。ただし後者の場合は、パラメータ渡しは引数結合で行い、親子結合を使わないように注意しなければならない。

○ wdir_i(整数型引数)

この場合は関数側で処理する必要はなく、仮引数をそのまま wdir_i に代入すればよい。

○ wdir_r(実数型引数)

(a) の書式にするには、仮引数 = 22.5× i±11.25 となるような i を探さなければならない。そのために仮引数に 11.25 を加えた後に 22.5 で除す。その商が i となる。ただし仮引数が 11.25 未満の場合には i が 0 となってしまうので、その場合は 16 に修正しなければならない。

○ wdir_c(文字型引数)

サブ・プログラム側に文字型の配列を用意しておき、配列要素のどれに一致するかを調べて、その配列要素番号を返すのが手っ取り早い。このとき注意すべきことは、

(i) 文字型配列の配列要素に data 文または宣言文の初期化式で初期値を与える場合、すべての配列要素が同じ文字長でなければならない。そのため短い文字列には後ろに空白を補って、文字長をそろえる。[20]

(ii) 実引数の文字長が一定でないので、仮引数の文字長は「*」としなければならない。

(iii) 比較の際は、「trim」関数で、後ろの空白を取り去った同士で比較する。

これらの関数を、適当な名前のモジュールに組み込んで、呼び出し側で use する。これにより wind_i、wind_r、wind_c を個別に呼び出すことができる。この際にモジュール側でこれらの名称は public 属性である必要がある。

(2) これらの関数を 1 つの総称名で呼び出すためには、モジュール宣言部に**引用仕様宣言** を加える。引用仕様宣言は総称サブ・プログラムだけでなく、他の目的にも使われるが、今の場合に使われる形式を、**総称引用仕様** (*generic interface*) といい、次の形式である。

```
interface 総称名
  module procedure 個別名 [, 個別名][, 個別名]...
end interface
```

上の例題の場合ばらば、次のようになる。

```
interface wdir
 module procedure  wdir_i, wdir_r, wdir_c
end interface
```

総称名が呼び出されると、コンパイラは module procedure の個別名で指定されたサブ・プログラムの中で、仮引数リストの個数とそれぞれの引数の型が、実引数リストのそれと一致するサブ・プログラムを呼び出す。したがって、サブ・プログラムの仮引数リストの、どれとも個数と型が一致しない実引数リストを指定した場合はエラーとなる。また、個別名で指定されたサブ・プログラムは、互いに仮引数の個数か型のどちらかが異なっていなければならない。すべて一致するものがある場合にはエラーとなる。

総称名は個別名と異なってもよいし、個別名の一つと一致してもよい。前者の場合には、総称名も public 属性を持たなければならない。

[20] 文字が漢字 (多バイトコード) である場合は簡単でない。処理系によって 1 文字のバイト数が異なるため、補うべき空白の数が異なるからである。可搬性を保つためには data 文や初期化式は使えない。一方代入文ならば左辺の文字長より右辺の文字長が小さい場合は自動的に空白を補うので、こちらを使うことになる。

例題 **6-4** のプログラム例

```
1  module wind
2     implicit none
3     private
4     public wdir, wdir_i, wdir_r, wdir_c
5
6   interface wdir
7    module procedure  wdir_i, wdir_r, wdir_c
8   end interface
9
10 contains
11
12   integer function wdir_i(iw)
13     integer, intent(in) :: iw
14     wdir_i = iw
15     return
16   end function wdir_i
17
18   integer function wdir_r(rw)
19     real, intent(in) :: rw
20     wdir_r = int((rw+11.25)/360.0*16.0)
21     if(wdir_r == 0) wdir_r=16
22     return
23   end function wdir_r
24
25   integer function wdir_c(cw)
26     character(*), intent(in) :: cw
27     integer :: i
28
29     character(3) :: ccw(16)= &
30      & (/'NNE','NE ','ENE','E  ','ESE','SE ','SSE','S  ',&
31      &   'SSW','SW ','WSW','W  ','WNW','NW ','NNW','N  '/)
32
33     do i = 1, 16
34       if(trim(cw) == trim(ccw(i))) exit
35     enddo
36     wdir_c = i
37     return
38   end function wdir_c
39 end module wind
40
41 program example6_02
42     use wind
43     implicit none
44     write(*,*) wdir(10)
45     write(*,*) wdir(1.0)
46     write(*,*) wdir('SSE')
47     stop
48 end program example6_02
```

6 章 の 章 末 問 題

【問題 6-1】

係数 a、b、c が実数である二次方程式、

$ax^2+bx+c=0$

について、a、b、c をキーボードから読み込み、判別式により実根、虚根、重根の判定を行ったうえで、その根を出力するプログラムを作成しなさい。

【問題 6-2】

1 バイト文字の文字列を実引数とし、その中の小文字をすべて大文字に変換した文字列を戻り値とする関数 `ucase` を作成しなさい。

(ヒント)　コード表を利用すること。

【問題 6-3】

実引数に文字列を与え、文字列中のそれぞれの「文字」の度数を、A が何個、B が何個というようにリストするサブ・プログラム `Poe` を作成しなさい。ここで、

・度数を計算する「文字」はアルファベットだけとし、数字、特殊記号、空白は含めない。

・大文字、小文字は区別しない (前問の関数を利用してもよい)。

【問題 6-4】

実引数にベクトル `x`、`y` を与え、`y` の `x` に対する正射影 $\frac{(\vec{x}\cdot\vec{y})}{(\vec{x}\cdot\vec{x})}\vec{x}$ を返す関数 `proj` を作成しなさい。

【問題 6-5】

例題 6.4 で、書式が' 東北東'、' 南'、' 北西' のように漢字であった場合、プログラムをどのように修正すればよいか。

第 7 章　入出力

§7.1 ファイル

ユーザが計算を行うにあたっては、まず必要なデータを計算機に渡して計算を行い、次に計算結果を何らかの形で計算機から受け取る必要がある。このときのユーザと計算機との媒介を行う機構のことを、(Man-Machine) Interface という。Fortran では、これは**ファイル** (*file*) によって行われる。

1. ファイルの種類

Fortran で扱うファイルを入出力先から分類すると、次の 3 種類となる。
(1) デバイス・ファイル … 特定のデバイス自体を、ファイルとして入出力を行う。指定できるデバイスは、標準では出力としてコンソール画面、入力としてキーボード + 画面エコーバックの 2 つである。これらへの入出力を、**標準入出力** という。
(2) (外部) ファイル … 外部デバイス (周辺装置) 上にファイルを作成し、それに対して入出力を行う。Fortran に限らず、プログラミング言語ではもっとも一般的な入出力方式である。
(3) 内部ファイル … メモリ上の文字型変数あるいは文字型配列をファイルとみなして入出力を行う。

名称	入出力先	装置識別子	ファイル形式	データ形式
標準入出力	システム標準装置	*	順編成	テキスト
外部ファイル入出力	外部記憶装置	装置番号	順編成／直接編成	テキスト／バイナリ
内部ファイル入出力	主メモリ	文字変数	順編成	テキスト

2. データ転送

ファイルに対して入出力を行うための文を**データ転送入出力文** (data transfer i/o statement) という。単に「入出力文」と言った場合は通常こちらを指す。データ転送入出力文では、入出力の際の様々な条件の指定を行うことができる。データ転送入出力文は、以下の 3 種類である。
· `read` **文** … ファイルからの入力を行う。
· `write` **文** … ファイルへの出力を行う。
· `print` **文** … 標準出力ファイルへの並び出力 (処理系の標準形式での出力) を行う。

3. 入出力形式

一般的な `read` **文**および `write` **文**の形式は、以下のようになる。

```
{read|write}(装置識別子 [, 書式指定子]) 入出力並び
```

○ **装置識別子** (*unit identifier*) … 入出力先のファイルを指定する。標準入出力なら「`*`」、外部ファイル入出力なら装置番号、内部ファイル入出力なら文字型の変数である。
○ **書式指定子** (*format specifier*) … 入出力の形式を指定する。処理系の標準形式を使用するなら「`*`」、ユーザが指定するなら書式仕様を記入する。バイナリでの入出力の際は、何も書かない。
○ **入出力並び** (*input/output list*) … 入出力する変数等を、入出力する順に記述する。

§7.2 標準入出力 (*standard i/o (input/output)*)

現在の Fortran での基本的なプログラムへの入力は、キーボードからの入力であり、入力されたキーは、特殊文字以外は端末画面にエコーバックされる。プログラムからの基本的な出力は、一時的な格納場所にとストアされて、端末画面へと出力される。*1

1. 標準入出力でのデータ転送入出力文

標準入出力の実行は、第 1 項目が「*」、第 2 項目が書式指定子の、read 文または write 文で行う。

```
read(*, 書式指定子)[, 入力並び]
write(*, 書式指定子)[, 出力並び]
```

または、以下の簡略型の read 文、または print 文 でも行うことができる。

```
read （書式指定子)[, 入力並び])
print （書式指定子)[, 出力並び])
```

2. リダイレクションによる入出力先の切り替え

コマンド・プロンプト状態から、リダイレクション記号「<」によって、データを標準入力からでなく、「input.txt」という名のファイルから読み込むことができる。

```
実行ファイル名 < input.txt
```

同様に、リダイレクション記号「> ファイル名」によって、標準出力でなくファイルに出力することができる。また、リダイレクション記号「>> ファイル名」によって、既存のファイルに書き加えることもできる。

演習問題

演習問題 1

次の標準入力文で (1)-(6) のデータを読むとき、a に読み込まれる値は何か (error になることもある)。

```
real :: a
read(*,*) a
```

(1)　1　(2)　.0　(3)　0.　(4)　.　(5)　e　(6)　1e

解答

(1)　1.000000 (2)　0.0000000E+00 (3)　0.0000000E+00 (4))　0.0000000E+00 (5))　0.0000000E+00 (6)　1.000000

演習問題 2

次の標準入力文で (1)-(5) のデータを読むとき、i に読み込まれる値は何か (error になることもある)。

```
integer :: i
read(*,*) i
```

(1)　1.6　(2)　.0　(3)　0.　(4)　e　(5)　1e

解答

*1 Windows や UNIX では、これらをそれぞれ標準入力 (*stdin*)、標準出力 (*stdout*) という。

(1) 入力エラー (2) 入力エラー (3) 入力エラー (4) 入力エラー (5) 入力エラー

§7.3 外部ファイル入出力

ここまでファイルへの入出力は、標準入力と標準出力のリダイレクションで行ってきたが、ここではファイル名を指定して、ファイルの直接入出力を行う一般的な方法を述べる。この方法は標準出力のリダイレクションと比べて、以下の利点がある。

・ 同時に複数のファイルを扱うことができる。

・ 扱うファイルをプログラム内で、実行時に指定できる。

・ 直接編成や書式なしファイルを扱える。[*2]

1. 外部ファイルへの入出力手順

ファイル操作入出力文

Fortran では、入出力のときにはファイルをファイル名で直接指定するのではなく、装置番号で指定する。そのためには入出力を行う前に、まず装置番号とファイルとを関連づけておかなければならない。そのための文が、**ファイル操作入出力文** (file operation i/o statement) である。またファイルの属性についての指定も、ファイル操作入出力文で行う。

主なファイル操作入出力文としては、以下のものがある。

· `open` **文** ··· ファイルと装置番号を関連づける。その他ファイルの属性について諸々の指定を行う。

· `close` **文** ··· ファイルと装置番号の関連を解消する。ファイルの属性についていくつかの指定を行う。

これ以外のファイル操作入出力文には、`inquire` **文**、`backspace` **文**、`endfile` **文**、`rewind` **文**がある。

入出力の手順

一般に、Fortran でファイルを扱うときは、次の手順をとる。

(1) open 文でファイルを装置番号に関連づける。

装置番号は正の整数であり、その上限は処理系により決まる。一つの装置番号には一つのファイルが関連づけられる。また open 文では、ファイルを関連づける際に、ファイルの諸々の属性を指定できる。

(2) データ転送入出力文 (`read` 文、`write` 文または `print` 文) で、open 文での属性指定に基づき、データの入出力を行う。ただし、1 回のデータ転送入出力文に対して、1 回ずつ入出力動作を行うわけではなく、システムによって定められた量まで一時的な記憶場所 (入力バッファ・出力バッファ-buffer) にためてから、動作が行われる。

(3) `close` 文でファイルを装置番号との関連づけから解放する。必ずしも open 文と同じプログラム単位になくてもよい。

予約ユニット番号による標準入出力文

ほとんどの処理系では、あらかじめシステムによって、ユニット番号「5」番が標準入力に、「6」番が標準出力に予約されている。その場合は以下のような標準入出力も可能である。

```
read(5, 書式指定子) [, 入力並び]
write(6, 書式指定子) [, 出力並び]
```

この場合は、ユニット番号「5」あるいは「6」を後述の open 文で他のファイルに割り当てることによって、標準入出力を切り替えることができる。

これに加えて標準エラー出力 (*stderr*) が装置番号「0」に割り当てられている処理系もある。[*3]

[*2] これらの扱いについては、この章では触れない

[*3] 標準出力ファイルがリダイレクトされている場合には、プログラムが標準出力にメッセージを出力しても、画面には表示されず、ユーザーは即座に認識ができない。そのような場合は標準エラー出力 (*stderr*) に出力することにより、画面に出力する。また、標準エラー出力はバッファリングを行わない。

2. ハードディスク上の外部ファイルの形式とその構造

ファイルの始まりと終わり

ハードディスク上のファイルは一般に複数のセグメントに分かれて置かれているが、これらの物理的な位置はシステムが管理しているので、ユーザは知る必要はない。ユーザは論理的なパスを指定すればファイルにアクセスできる。ユーザから見たファイルは、一続きのバイトの並びで、その最後はファイル終了コードで終わる。これを `EOF` **コード** (End Of File) といい、ユーザはこのコードを読み出したことを検知することにより、ファイルの最後に到達したことを知ることができる。

改行コード

文字コードだけで書かれたファイルは、適当な長さの行に区切ることができる。そのために OS ごとに特定のコードを定め、そのコードが現れたならばそこまでを「1 行」ときめている。その特定のコードを改行コードあるいは `EOR` **コード** (End Of Record) という。[*4] このコードは実際にファイルに書き込まれているコードであり、Linux では「`0A`$_{(16)}$」(LF)、Windows では「`0D0A`$_{(16)}$」(CR LF) である。文字コードはコード表にこれらのコードを避けて配置されている。

テキストファイルとバイナリーファイル

このように、文字コードが行単位で書かれているファイルのことを、**テキストファイル** (text file) といい、それ以外のファイルを **バイナリーファイル** (binary file) といい、扱い方が異なる。[*5] これまで扱ってきた、並び入出力や書式仕様による入出力を行うファイルは、テキストファイルである。

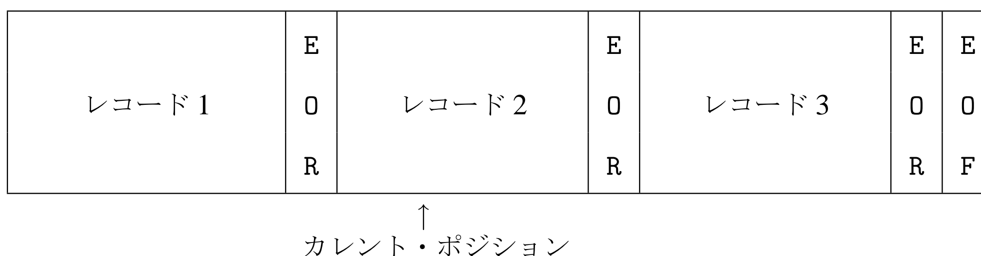
順編成ファイルと直接編成ファイル

これまで扱ってきたファイルでは、入出力を行うカレント・ポジション (current position) は最初はファイルの先頭にあり、入出力を行うとファイルの後ろへと順次進んでいった。このようにカレント・ポジションが順次移動するファイルを、**順編成ファイル** という。

Fortran ではこれ以外に、一定の長さのレコードに番号を付け、任意の番号へとアクセスできるファイルを扱うことができる。こちらは**直接編成ファイル** という。

順編成ファイルの構造

順編成ファイルは、(書式のあるなしにかかわらず) 下図のような構造をなしている。



カレント・ポジション

カレント・ポジションのあるレコードをカレント・レコード、その前のレコードを直前レコード、その後のレコードを直後レコード、という。カレント・ポジションが `EOR` にある時には、カレント・レコードが存在しないときもある。

前述したように、本章ではテキスト形式の順編成ファイルについてのみ解説を行う。

[*4] レコード (論理レコード) は、直接編成ファイルや書式なしファイルのようなバイナリーファイルにも設定される。ただし、テキストファイルについては「行」=「レコード」とみなしてよく。両者を区別する必要はない。

[*5] テキストファイルを、バイナリーファイルであるとみなして扱うことは可能である。この場合改行コードは単なるデータとみなされ、改行の制御は利かなくなる。逆にバイナリーファイルをテキストファイルとして扱うことは、一般に不可能である。

3. open 文

open 文の一般形式

順編成書式付きファイルの open 文は、次の形式である。

```
open([unit=]un[,para-list])
```

○「[unit=]*un*」は必須であり、省略することはできない。open(の直後の第一パラメータであるときには「unit=」の部分を省略でき、「*un*」だけでよい。

○「[,para-list]」は、「,」で区切られた「para=*parameter*」の形式の並びであり、必要に応じて指定する。「para=」の部分は省略することはできない。デフォルト値があるものは、下線で示す。

(1) 装置番号 [unit=]*un*

○「*un*」は装置番号をあらわす整数式であり、その値は正でなければならない。

· ファイルと装置番号の関連づけは、プログラム全体で有効である。すなわちどのプログラム単位でも、同じ装置番号に対して入出力をおこなえば、同じファイルに読み書きされる。

· すでに他の open 文で使用されている装置番号を指定すると、エラーになることがある。

·「*un*」の上限は処理系によって異なるが、可搬性を考慮して、99 以下にしておくことが無難である。

· 一ケタの装置番号は、処理系が予約している可能性があるので、やはり可搬性を考慮して、10 以上にしておくことが無難である。

→ すなわち 10-99 の、二ケタの装置番号が推奨される。

(2) ファイル名 file=*filename*

○「*filename*」はファイル名の記された文字定数または文字変数である。

· 一時ファイル (scratch file) 以外は必ず指定する。

· ファイル名の命名規則は、OS の規則に従う。[*6]

· パス名を指定できるが、絶対パス指定は可搬性を損なうので、相対パスにすべきである。

· すでに他の open 文で使用されているファイル名を指定すると、エラーになることがある。

· 他のプロセス (プログラム) で使用されているファイル名を指定すると、エラーになることがある。

(3) ファイル状態 status={ 'new' | 'old' | 'replace' | 'unknown' | 'scratch' }

○ ファイルの状態と、操作を指示する文字定数または文字変数で、上の並びの中から選択する。

· 'new' は出力ファイルに対して指定され、open 文が実行された時点でファイルが存在するとエラーになる。既存のファイルを誤って上書きしてしまうのを防ぐために指定する。

· 'old' を指定した場合、逆に open 文が実行された時点でファイルが存在しないとエラーになる。

· 'replace' は出力ファイルに対して指定され、open 文が実行された時点でファイルが存在しなければその名前のファイルを新しく生成し、存在すればそのファイルを消去して上書きする。

· 'unknown' を指定した場合、あるいは status パラメータを省略した場合、扱いは処理系依存になる。[*7]

· 'scratch' は**一時ファイル** (scratch file) に対して指定され、close 文が実行されるか、プロセスが終了するとファイルは消去される。status='scratch' を指定した場合、ファイル名を指定してはならない。

→ 一般には、'new' を指定するか、省略するかである。

(4) 動作モード action={ 'read' | 'write' | 'readwrite' }

○ 関連づけられたファイルに対して、入力、あるいは出力動作を制限する。

[*6] 一般に Fortran では大文字、小文字の区別はされないが、*filename* のような文字式の中では区別される。そこで、Windows のファイル名は大文字、子文字の区別はされないが、Linux では区別されることに注意せよ。

[*7] Intel Fortran では、ファイル存在しない場合は'new'、存在する場合は'old' として扱われる。

· 'read' を指定した場合、そのファイルに対して `write` 文や `print` 文による出力ができなくなる。[*8]
· 'write' を指定した場合、そのファイルに対して `read` 文による入力ができなくなる。
· 'readwrite' を指定した場合、そのファイルに対する入出力の制限はなくなる。
→ 'read' を指定することにより、目的のファイルを誤って上書きしてしまうことを防ぐ。[*9]

(5) 追加モードでのオープン `position=`{ 'rewind' | 'asis' | 'append' }
○ すでに存在するファイルに、追加出力を行うための指定子である。入力で指定するのは、かなり特殊な使い方である。
· 'rewind' を指定すると、ファイル・オープン時のカレント・ポジションは、ファイルの先頭に置かれる。
· 'asis' は、現在の Fortran i/o system では、'rewind' と同じである。
· 'append' を指定すると、ファイル・オープン時のカレント・ポジションは、EOF の直前に置かれる。この後で write 文を実行すれば、それまで存在したレコードの後に、出力したレコードが追加される。

(6) エラーによる分岐 `err=`*stmt*
○ *stmt* は文番号であり、open 文の実行時にエラーが起きたならば、文番号 *stmt* の文に分岐する。

```
    open(20, file='test2.txt', status='new', err=90)
                              |
 90 write(*,*) "file open error"
```

· open 文の実行時にエラーが起きたときに、`err` 指定子も、次の `iostat` 指定子もない場合、プログラムの実行は停止する。

(7) エラーコード検出 `iostat=`*ist*
○ open 文が正常に動作したかどうかを知るための指定子で、*ist* には基本種別の整数型変数を指定する。

```
    integer :: ier
    open(10, file='test1.txt', status='new', iostat=ier, err=99)
                              |
 99 write(*,*) "file open error code=",ier
```

· open 文が正常に実行された場合、*ist* には 0 が代入される。
· open 文の実行時にエラーが起きた場合、*ist* にはエラーコードを示す、正の値が代入される。その値は処理系依存である。
· open 文の実行時にエラーが起きたときに、`iostat` 指定子も、`err` 指定子もない場合、プログラムの実行は停止する。`iostat` のみ指定されている場合は、open 文の実行を中止して、次の文を実行する。[*10]

(8) 入力フィールド中の空白の扱いの指定 `blank=`{ 'null' | 'zero' }
○ 書式付き入力文を実行するとき、入力フィールド中の空白を「`null`」とみなすか、「`0`」とみなすかは、後述の書式仕様の BZ 指定子、BN 指定子によるが、どちらも指定されないときの扱いを定めるための指定子である。
· 'null' を指定すると、デフォルトで、BN 指定子が指定される。
· 'zero' を指定すると、デフォルトで、BZ 指定子が指定される。

[*8] 多くの場合、複数の open 文、あるいは他のプロセスが、同じファイルに対してすべて'read' モードでオープンしていれば、そのファイルは共有できる。
[*9] `status=`'new' は、誤って他のファイルを破壊しないため、`action=`'read' は、目的のファイルを破壊しないための指定であるので、混同しないように。
[*10] `iostat` 指定子があれば、エラーコードによってエラー・ハンドルを行えるので、`err` 指定子は不必要なように思えるかもしれないが、エラーコードは処理系依存なので、それでは可搬性が損なわれてしまう。

(9) 並び出力文での文字型出力の様式 `delim=`{ ’apostrophe’ | ’quote’ | ’null’ }

○ 並び出力「`*`」で文字式を出力した場合、文字列の中に区切り子「`,`」あるいは「␣」があると、並び入力ではそのまま読めなくなる。そこで、文字列の両端に「`"`」または「`'`」を付加して、文字列であることを示すための指定子である。なお、文字式の出力が書式付きの場合には効果がない。

· ’apostrophe’ を指定すると、文字列を並び出力する際に、アポストロフィー「`'`」で囲まれた形で出力する。文字列中に「`'`」が含まれていた場合には、二重「`''`」にして出力する。

· ’quote’ を指定すると、アポストロフィーの代わりに引用符「`"`」で同じ動作をする。

· ’null’ を指定すると、文字列をそのまま出力する。

(10) 書式付き入力で、レコードを越えるデータを要求した場合の処理 `pad=`{ ’yes’| ’no’ }

○ 書式付き入力で、書式仕様が要求するフィールド幅が、カレントポイントから EOR までのバイト長を上回ったときの処理を指定する。並び入力や、出力に対しては意味がない。

· ’yes’ 不足するバイト分だけの空白を付加する。

· ’no’ 何もしない (たぶん実行エラーになる)。

通常はデフォルトの pad=’yes’ のままで使うが、変数に予期できない値が代入されたりするとき、書式仕様の誤りを検出するために、故意に pad=’no’ にする、という使い方がある。

4. close 文

close 文の一般形式

close 文は、全てのファイル形式で共通で、次の形式である。

```
close([unit=]un[,para-list])
```

○「`[unit=]`*un*」は必須であり、省略することはできない。`open(`の直後の第一パラメータであるときには「`unit=`」の部分を省略でき、「*un*」だけでよい。

○「`[,para-list]`」は、「`,`」で区切られた「`para=`*parameter*」の形式の並びであり、open 文と使い方は同一だが、指定できるものは「`status`」、「`iostat`」、「`err`」の 3 個にすぎない。

○ close 文を省略しても、正常終了すれば end 文が代わりに関連を解放してくれるが、それまでの間に異常終了するとファイルが消去されてしまうので、アクセスする必要がなくなったファイルは明示的に close すべきである。

○ close した装置番号は、他のファイルに open 文で関連づけることもできる。close しないまま他のファイルに対して open するという誤りを防ぐためにも、必要のない関連づけは速やかに close すべきである。

(1) 装置番号 `[unit=]`*un*

○ *un* は装置番号をあらわす整数式であり、open 文でファイルに関連づけられた値である。

· 装置番号に対応した、ファイルとの関連づけを解消する。ファイル名を指定する必要はない。

(2) ファイルの後処理 `status=`{ ’keep’ | ’delete’ }

· ’keep’ を指定すると、close 文の実行後も、ファイルは残る。

· ’delete’ を指定すると、close 文の実行後に、ファイルは消去される。

· デフォルトは、open 文の `status` 指定子が、’scratch’ ならば’delete’、それ以外ならば’keep’ である。

(3) エラーコード検出 `iostat=`*ist*

open 文と使い方は同じである。

(4) エラーによる分岐 `err=`*stmt*

open 文と使い方は同じである。

5. inquire 文

inquire 文の用法

inquire 文は、装置番号あるいはファイル名のいずれかを指定し、それが現在使われているか、また使われている場合は open 文で指定したパラメータ値を取得するための文である。

inquire 文の一般形式

順編成書式付きファイルの inquire 文は、次の形式である。

`inquire([unit=]`*un*`[,para-list])` または

`inquire([file=]`*filename*`[,para-list])`

○「`[unit=]`*un*」と、「`[file=]`*filename*」はどちらか一方が必須である。両方を指定することはできない。
○「`[,para-list]`」は、「`,`」で区切られた「`para=`*parameter*」の形式の並びであり、必要に応じて指定する。「`para=`」の部分は省略することはできない。

問い合わせるパラメータには多くの種類が指定できるが、それらの詳細は文法書にゆだね、ここではその装置番号やファイルが使用中であるかどうかを問い合わせるパラメータのみを示す。

(1) 接続問い合わせ　`[exist=]`*ex*

○「*ex*」は接続状態を返す論理型変数であり、指定された装置番号あるいはファイルが接続されていれば「`.true.`」、接続されていなければ「`.false.`」を返す。

演 習 問 題

演習問題 3

次の要求を満たす open 文はそれぞれどう書けばよいか。
(1) ファイル「`'abc.txt'`」を、装置番号 10 番で読み込み専用モードで開く。
(2) ファイル「`'def.txt'`」を、装置番号 20 番で追加書き込みモードで開く。
(3) ファイル「`'ghi.txt'`」を装置番号 30 番で開き、並び入出力文での文字出力を、quote「`"`」で囲むように指定する。
(4) 装置番号 40 番で一時作業ファイルを開く。

解答

(1)　`open(10,file='abc.txt',action='read')`
(2)　`open(20,file='def.txt',position='append')`
(3)　`open(30,file='ghi.txt',delim='quote')`
(4)　`open(40,status='scratch')`

§7.4 内部ファイル入出力

1. 内部ファイル

文字変数に対する読み書き

Fortran では、文字変数、あるいは文字配列などを、あたかもファイルであるかのように、データ転送入出力文で読み書きすることができる。このようなファイルのことを**内部ファイル** という。

内部ファイルは特定の装置に関連づける必要はないため、open 文などのファイル操作入出力文は不要であり、いきなりデータ転送入出力文 (read 文、write 文) を書いてよい。ただしそのために、本来 open 文で指定されるべき、`blank`、`pad` などのパラメータが指定できない。これらのパラメータ値はすべてデフォルト値になる。

内部ファイルを使う目的は、主に文字型のデータと、数値型のデータとの内部変換である。

内部ファイルに使用できる文字型データ

内部ファイルとして指定できるものは、次のいずれかである。

· 文字変数
· 文字配列
· 文字配列の配列要素
· 部分文字列

2. 内部入出力文で利用できる指定子

内部入力文

内部ファイルに対する read 文は、以下のような形式である。

```
 character(10) :: str="abc1234567"
 integer :: i, ios

 read(str,'(i3)',err=91,iostat=ios) i
91 write(*,*) i, ios
 read(str,'(8x,i3)',end=93,iostat=ios) i
93 write(*,*) i, ios
```

利用できる指定子は、外部 read 文で利用できるもののうち、`fmt`、`iostat`、`err`、`end` の 4 個であり、`advance`、`eor`、`size` 指定子は使えない。なお、`fmt` に対して「`*`」も指定できる。

内部出力文

内部ファイルに対する write 文は、以下のような形式である。

```
 character(15) :: str
 integer :: ios
 real :: a=1.23

 write(str,'(5x,f5.2)',iostat=ios) a
 write(*,*) str, ios
 write(str,'(i3)',err=91,iostat=ios) a
91 write(*,*) ios
```

利用できる指定子は、外部 write 文で利用できるもののうち、`fmt`、`iostat`、`err` の 3 個であり、`advance` 指定子は使えない。なお、`fmt` に対して「`*`」も指定できるが、フィールド幅をユーザが指定できないので、あまり使うことはない。

§7.5 並び入出力 (list-directed i/o)

read 文、write 文あるいは print 文の書式指定子として、「*」を指定することを、**並び入力** (list-directed input) および**並び出力** (list-directed output) という。

1. 並び出力文

並び出力では、出力リストの式がそれぞれの型にしたがって、適当な書式指定により文字列に変換されて出力される。どのような書式が選択されるかは処理系によって異なる。それぞれの出力は空白「␣」で区切られる。

特に出力形式を指定する必要がなく、計算結果を確認するだけならば、通常は並び出力で十分である。出力形式を詳しく指定する場合には、並び出力ではなく、書式仕様を指定して出力する。

文字型の出力

デフォルト状態で文字を出力すると、囲み指定子である「'」や「"」は出力されない。また、「''」は「'」に、「""」は「"」に変換され、内部表現がそのまま出力される。

これは、この出力をそのまま並び入力で読みこもうとするとき、文字列の中に「␣」や「,」があれば、そこまでを 1 つのデータとして区切られて読み込まれることを意味する。このような場合には、書式付きの入出力を行わなければならない。[*11]

並び出力の改行

通常の (不定長) レコードへの出力では、改行を指定することができず、出力リストはすべて 1 行に出力される。したがって、1 つの出力文での出力並びの数は、適当な数で押さえておくことが望ましい。[*12]

2. 並び入力文

並び入力では、プログラムは、指定された場所から値を順次読み取り、入力リストの各変数の型にしたがって、処理系によってあらかじめ規定された方法にしたがって内部コードに変換して、対応する変数の番地に格納する。

しかし、入力並びの変数と、入力ファイルの項目の順序と型の対応には注意が必要である。正確に対応していれば問題はないが、現実には往々にして誤りが混入する。そうしたとき、並び入力の場合はプログラムにより適当に解釈されて、誤りが誤りと認識されないまま看過されることがある。特に、入力並びの変数が欠けて、対応がずれた場合はその可能性が大きい。

区切り子

並び入力によりデータを入力するには、入力項目並びにある変数の順番にしたがって、データを**区切り子** (value separator) で区切って入力していく。区切り子はコンマ「,」または (半角) 空白「␣」プラス任意の数の空白、である。 1 つのデータの途中に空白を入れてはいけない。とくに、文字列の中に「,」があると、その直前の文字が 1 つのデータの終わりとみなされるので注意すること。

データの反復

同じ値を繰り返して入力するためには、冒頭に繰り返し回数およびアスタリスク「*」をつける。

```
1.0, 3*2.0, 3
```

この場合、1.0, 2.0, 2.0, 2.0, 3.0 と入力したとみなされる。

[*11] あるいは、open 文で DELIM='apostrophe'、あるいは DELIM="quote"と、パラメータを指定すれば、並び入力で読めるように出力される。

[*12] open 文で recl 指定子を使えば、固定長ファイルに出力でき、出力は処理系によってレコード長以内の適当個数で折り返される。また文字は改行コードに続けて書かれる。しかし、固定長ファイルに (書式を指定せず) 並び出力を行うというのは、通常の出力とは言い難い。

入力並びの数と入力データの数が一致しないとき

入力並びにある変数の数が、現在入力対象としている行にある、区切り子で区切られたデータの個数よりも多いときは、プログラムは全ての変数を読み込むまで、次々に行を読みに行く。

逆に、入力並びにある変数の数が、現在対象としている行にある、区切り子で区切られたデータの個数よりも少ないときは、変数の数までで読み込みは打ち切られる。次の read 文は、読みかけの行の、次の行から読み込みを始め、読まれなかったデータは無視される。

データ読み込みの打ち切り

入力並びにしたがって、データを読み込んでいる途中で「/」を読み込むと、read 文はそこで入力動作を打ち切る。入力並びのそれ以降にある変数は不変のまま、制御は次の文に移る。

文字型データの読み込み

「,」、「/」、「'」、「"」、(半角) 空白を含まず、数字*で始まらない文字列は、区切り子で区切って普通に読むことができる。

これらのの文字を含むときには、「'」か「"」で囲まなければならない。このとき、囲った文字を二つ続けると、一つの文字として解釈される。またこの場合に限り、複数行にわたった入力が可能である。

'aaa''bbb' → 「aaa'bbb」

null 値

二つの「,」が「,,」のように連続した場合、あるいは行の開始直後に「,」がある場合、null 値という。このとき、その順番に対応する入力並びの変数は不変であり、以前の値が保持される。ただし、改行コードまたは「/」の直前に「,」がある場合は、その間は一つの null 値とはみなさない。

プログラム例 7-a　null 値

```
1  program null_value
2    implicit none
3    integer :: i, n, m(5)=0
4
5    read(*,*) n,(m(i),i=1,n) ! 4 までしか読み込まれない
6    write(*,*) m
7    read(*,*) m              ! 次の行から読み込み、改行直後の「,」は null 値
8    write(*,*) m             ! 13 以後は読み込まれない
9    read(*,*) m              ! 15 の後は null 値
10   write(*,*) m             ! 16 の後の改行直前の「,」は無視される
11
12   stop
13 end program null_value
```

入力データ

```
4,  1, 2, 3
4, 5, 6, 7, 8, 9, 10, 11
,12, 13 / 14
15,,16,
17, 18, 19
```

出力データ

```
           1           2           3           4           0
           1          12          13           4           0
          15          12          16          17          18
```

演 習 問 題

演習問題 4

次の並び入力文で (1)-(8) のデータを読むとき、c に読み込まれる値は何か。

```
character(10) :: c
read(*,*) c
```

(1) `""""`　(2) `abc,def`　(3) `"abc,def"`　(4) `"abc""def"`　(5) `'abc""def'`

(6) `"abc"def"`　(7) `'abc"def'`　(8) `"""` / `ab"` (2 行にわたる)

(注) 入力を間違えて入力待ちでハングしたときは、ctrl-C で JOB を中断しなさい。

解答

`write(*,*) c` での出力

(1) `"`　(2) `abc`

(3) `abc,def`　(4) `abc"def`

(5) `abc""def`　(6) `abc`

(7) `abc"def`　(8) `"ab`

§7.6 書式仕様による入出力

1. 書式仕様の指定

read 文、write 文あるいは print 文の書式指定子に、並び入出力の「*」の代わりに、**書式仕様** (format specification) を記述することにより、入出力の方法を詳しく指定することができる。

書式仕様とは、 **編集記述子** (edit descriptor) の集まりをカッコ「()」でくくったものであり、read 文あるいは write 文の「,」で区切られた、第 2 番目の項目に、次のいずれかの方法で指定する。

(1) 直接文字定数として記述する。

```
    write(*,"('x= ',f10.2)") x
```

(2) 書式仕様を記述した文字列を指定する。

```
    character(20) :: form="(2x,i5)"
    write(*,form) j
```

(3) 書式仕様を記述した文字配列を指定する。

```
    character(4) :: forms(2)
    forms(1)="(3x," ; forms(2)="i10)"
    write(*,forms) k
```

(4) 書式仕様を記述した format 文の文番号を指定する。

```
    read(*,200) i, j
200 format(2i5)
```

標準入出力文の書式仕様

print 文あるいは read 文のあとに、空白を開けて「()」でくくらずに、書式仕様を記述できる。

```
read 100, m, n
print "1x,2i5", m, n
```

ただし、標準入出力文で書式仕様を指定することは、入出力並びと紛らわしいので、あまり好まれない。

2. 編集記述子

編集記述子には、以下の三種類がある。

データ編集記述子

データ編集記述子 (data edit descriptor) は、入出力並びのある式と対応して、その内部表現 (ビット列) と外部表現 (文字列) を、記述子に指定された規則に従って、相互に変換する。したがって、一つの式に対して、一つのデータ編集記述子が対応する (逆は真ではない)。

文字列編集記述子

文字列編集記述子 (string edit descriptor) は、記述された内容を出力する。出力並びに対応する式はない。また、入力では使われない。

制御編集記述子

制御編集記述子 (control edit descriptor) は補助的な記述子であり、他の記述子の動作を限定したり、入出力の詳細を規定したりと、多用な役割がある。特定の入出力並びの式とは対応しない。

3. 書式仕様の規則

入出力動作と改行

入力の場合、ちょうど文字を次々と読むように、入力ポイントが入力ファイルの中を移動していく。一つの read 文が終了すれば、入力ポイントは次の行 (レコード) に移る。このとき、最後に読んでいたレコードにデータが残っていても読みとばされる。書式仕様で、明示的にデータを読み飛ばして次の行に移動することもできる。ただし、いったん読んだ行を後戻りしてもう一度読むことはできない。*13

出力の場合、出力ポイントは一般に出力ファイルの最後にある。一つの write 文あるいは print 文が終了すれば、改行コードが出力され、出力ポイントは次の行に移る。また、入力と同様に明示的に改行コードを出力することもできる。この場合も、一つの行の中で出力ポイントを一時的に戻すことは可能であるが、いったん出力してしまった行を、書式仕様で書き換えることはできない。

書式仕様との対応

入出力は、入出力並びの式を順番に、書式仕様のデータ編集記述子の並び順にしたがって、内部表現 ↔ 外部表現の変換を行っていく。 したがって入出力並びの式の型と、データ編集記述子の扱う型とは対応していなければならない。

```
integer :: i
real    :: r
character(4) :: c
read(*,'(i4,f10.0,a4)') i, r, c
```

書式仕様中の数字

書式仕様に表れる数字は、すべて 10 進数の定数であり、変数や名前付き定数は使用できない。また、種別も指定できない。もし書式仕様を、状況にしたがってプログラム中で変更したいならば、文字式に書式仕様を書き込んで、それを書式仕様として指定するという方法によるしかない。*14

入れ子の書式仕様と反復

書式仕様をカッコ「()」でくくってグループとして、他の書式仕様に含めることができる。これを入れ子の書式仕様、と言う。このとき、そのグループに対して反復回数を指定できる。

たとえば、　(3x,3(f6.1," m "),i5) という書式仕様は、

(3x,f6.1," m ",f6.1," m ",f6.1," m ",i5) という書式仕様と同等である。

この「()」でくくられた入れ子の書式仕様を、レベル 1 のグループという。また、このグループもグループを含むことができ、それをレベル 2 のグループという。同様に、レベル 3··· のグループも使用できる。

書式仕様の反復

入出力並びの式の個数と、書式仕様の中のデータ編集記述子の個数は必ずしも一致しなくてもよい。式の個数が、データ編集記述子の個数よりも少ない場合、空の式を評価しようとした時点で、入出力は終了する。式の個数が、データ編集記述子の個数よりも多い場合、以下のルールに従う。

(1) 書式仕様の終わりに達したところで改行される。すなわち入力の場合ならば、入力対象は次の行に移り、出力の場合なら、改行コードが書かれる。

(2) 書式仕様にグループがなければ、書式仕様の先頭に戻って、処理が行われる。

(3) 書式仕様にグループがあれば、レベル 1 のグループの中で一番右端のグループの先頭に制御が戻る。

(4) そのグループに反復回数指定があれば、優先して処理が行われる。

---

*13 標準入出力でなければ、backspace 文あるいは rewind 文の実行により、ポイントのある行を戻すことができる。出力でも同様。

*14 その方法については、内部ファイルの項を参照すること。

§7.7 データ編集記述子および文字列編集記述子

1. データ編集記述子全般について

フィールド幅

一般にデータ編集記述子は、**フィールド幅** w を、データ編集記述子の直後に指定しなければならない。フィールド幅とは、入出力データ 1 個の、入出力レコードに占める文字の幅である。*15

出力がフィールド幅に収まらないときは、フィールド幅全部に「*」が出力される。

I 型、B 型、O 型、Z 型と F 型では、出力に w に 0 を指定できる。この場合空白は詰めて出力される。*16

例外は文字列を扱う A 型編集記述子であり、このときはフィールド幅を省略することができる。このときの文字幅は、入力の場合対応する文字変数、出力の場合は対応する文字式の文字長となる。

反復

データ編集記述子の前に、**くり返し回数** r を指定することができる。省略されたときには 1(回) と解釈される。これは、その編集記述子を r 回連続して指定することを示す。

くり返し回数は正の整数である。また、くり返し回数が指定できるのはデータ編集記述子のみであり、文字列編集記述子や制御編集記述子には指定できない。

2. 整数型データ編集記述子

I 型編集記述子 ···10 進数の入出力

○ I 型編集記述子の一般形は、次の形である。

Iw [.m]

ここで m は出力の文字数であり、本来の出力の桁数が m に足りない場合は 0 が補われる。m はフィールド幅 w を越えない、すなわち $m \leqq w$ でなければならない。入力時には、指定しても無視される。

○ I 型編集記述子は、整数型の 10 進数の入出力に使われる。他の型に使われた場合の結果は保証されない。

○ I 型編集子での入出力で使える文字は「0123456789」の数字とそれに先行する符号「+-」である。

(例) 「␣␣1␣23456」 （␣ は空白を表す） というデータを、

```
read(*,'(3i3)') i1, i2, i3
```

というフォーマットで読めば、i1 には 1、i2 には 23、i3 には 456 が、それぞれ代入される。

(例) `write(*,'(i4, i4.3, i4)') 99, 99, -9999`

という文を実行すると、「␣␣99␣099****」と出力される。最後の****はフィールド幅の不足による。

B 型 、O 型 、Z 型 編集記述子 ···2, 8, 16 進数の入出力

○ 一般形は、それぞれ次の通りで、それぞれの使い方は、I 型編集子と同様である。

Bw [.m]　　Ow [.m]　　Zw [.m]

○ B 型は 2 進数、O 型は 8 進数、Z 型は 16 進数の整数型の入力に使われる。使われる文字は以下の通り。

· B 型編集記述子 → 「01」

· O 型編集記述子 → 「01234567」

· Z 型編集記述子 → 「0123456789」と「abcdef」

○ 出力の場合、どの型にも使うことができ、内部表現を 2 進、8 進、16 進で出力できる。

*15 多くの処理系では、フィールド幅を省略してもエラーにならず、処理系が適当な値を補ってくれるが、それは行うべきではない。それくらいなら並び入出力を使用すべきである。

16 これを指定することにより、Excel の「.CSV」ファイル形式の出力ができる。

3. 実数型 (複素数型) データ編集記述子

実数型データの入力

入力の場合、フィールド幅に実数型定数が収まっていれば、以下のどの実数型編集記述子を指定して入力してもよく、変数に入力される値は同一である。編集記述子の選択は、出力の場合のみ意味を持つ。ただし、種別指定子「_kind」は入力データ中にあってはいけない。

小数部、指数部のケタ数の指定

実数型データの編集記述子には、小数部のケタ数 d と、指数部のケタ数 e というパラメータがある。

○ d は小数部のケタ数を表し、すべての編集記述子で必須である。その値はゼロ以上で上限は編集記述子の種類とフィールド幅によって決められる。出力でゼロを指定した場合、最後に小数点が出力される。

○ e は指数部のけた数を表し、F 型,D 型以外の編集記述子で指定でき、デフォルト値は $e=2$ である。F 型は指数部がないので不必要であり、D 型は $e=2$ に固定されている。

F 型編集記述子 ··· 指数部のない実数型の入出力

○ F 型編集記述子の一般形は、次の形である。

F $w.d$

○ 入力の場合は、フィールド幅を超えない、すなわち $d \leqq w$ であればよい。

○ 出力の場合は、小数点以下 d ケタに加え、負の場合の符号出力 1 ケタ、整数部の数字がすくなくとも 1 ケタ、および小数点の出力が必要なため、$d \leqq w-3$ でなければならない。*17

○ フィールド内に小数点が 1 個あれば (2 個以上あればエラー)、その小数点が d の指定よりも優先される。

○ フィールド内の文字が、「.」だけ、または「e」だけの場合は、0 とみなされる。

(例) 「1234567890-.234␣1.2␣␣␣13␣␣.␣␣␣1.2e1」 というデータを、

```
read(*,'(7f5.2)') r1, r2, r3, r4 ,r5, r6, r7
```

というフォーマットで読めば、r1 には 123.45、r2 には 678.90、r3 には −0.23、r4 には 1.20、r5 には 0.13、r6 には 0.00、r7 には 12.00 が、それぞれ代入される。

(例) また、上で読み込んだ変数を、

```
write(*,'(7f5.2)') r1, r2, r3, r4 ,r5, r6, r7
```

で出力すれば、「**********-0.23 1.20 0.13 0.0012.00」 と出力される。*****は、以前の例と同様に、値を書き込むには、フィールド幅が足りないことを表す。

E 型編集記述子 ··· 指数部のある実数型の入出力

○ E 型編集記述子の一般形は、次の形である。

E $w.d$[Ee]

○ 入力の場合、$d+e \leqq w$ でなければならないが、そもそも Ee の部分は、入力では動作しないので、指定する意味はない。

○ 出力の場合、$d \leqq w-7$ でなければならない。すなわち、小数部の d ケタに加え、符号に 1 ケタ、「0.」に 2 ケタ、「E」に 1 ケタ、指数部の符号に 1 ケタ、指数部が 2 ケタ (単精度のけた数は 100 を越えない) の計 7 ケタ分が必要である。

*17 F 型編集子による入力の場合、入力フィールドの実数定数の型を指定することはできないが、Intel Fortran の場合は、倍精度で読み込まれてから、入力並びの変数の型に変換されるようである。

○ デフォルトの指数部のケタ数は 2 であり、e に 3 以上を指定すれば、必要なフィールド幅はその分増加する。逆に e に 1 を指定して、出力する数のベキが 2 ケタであれば、`E` が出力されなくなる。
○ 単精度の有効ケタ数は 7 なので、区切りのための空白を 1 ケタ加えた `e15.7` で出力されることが多い。
(例) `a1` が -3×10^{50} であるとき、「`write(*,'(e15.7)') a1`」を実行すると、

「␣`-0.3000000E+51`」と出力される。

D 型編集記述子 ··· 指数部のある倍精度実数型の入出力

○ D 型編集記述子の一般形は、次の形である。

D *w*.*d*

○ 出力形式は E 型と同じであるが、倍精度であることを示すため、指数部の「`E`」が「`D`」に変わる。
○ ただし、ベキのケタ数 e を指定できないため、仮に $b1 = -2\times10^{100}$ を「`d15.7`」の形式で出力すると、

「␣`-0.2000000+101`」

となり、D が消える。この表現を避けたいならば、E 型編集記述子を使い「`e16.7e3`」で出力すれば、

「␣`-0.2000000E+101`」

と出力することができる。

ES 型編集記述子 ··· 有効数字形式実数型 (science) の入出力 (90)

○ ES 型編集記述子は、実数を有効数字形式で出力するための記述子で、その形式は次の形である。

ES *w*.*d* [E*e*]

○ 出力の場合、123.45 および −12.2 を「`es15.7`」で出力すると、

「␣␣`1.2345000E+02`」、「␣`-1.2200000E+01`」

のように、整数部が 1 ケタで出力される。

EN 型編集記述子 ··· 工学形式実数型 (engineering) の入出力 (90)

○ EN 型編集記述子は、単位を 3 ケタごとに定める、SI 形式で出力するための記述子で、その形式は次の形である。

EN *w*.*d* [E*e*]

○ 出力の場合、123.45 および −12.2 を「`en17.7`」で出力すると、

「␣␣`123.4499969E+00`」、「␣␣`-12.1999998E+00`」

のように、指数部が (負も含め)3 の倍数、整数部が 3 ケタ以内で出力される。この場合は、整数部が 3 文字になることがあるので、$d \leqq w-9$ でなければならない。

複素数型の入出力

複素数型の入出力に対する、特別な書式はなく、二つの実数型により、実部、虚部の入出力を行う。この場合、実部と虚部のデータ編集記述子は同一である必要はなく、その間に制御編集記述子があってもよい。

4. 論理型データ編集記述子

L 型編集記述子 ··· 論理型の入出力

L 型編集記述子は、論理型の入出力のための記述子で、その形式は次の形である。

L*w*

∘ 入力の場合、フィールド中に「`.true.`」、または「`.T`」、「`.t`」、「`T`」、「`t`」から始まる文字列があれば、「真」とみなされる。

また、フィールド中に「`.false.`」、または「`.F`」、「`.f`」、「`F`」、「`f`」から始まる文字列があれば、あるいは全てのカラムが空白であれば、「偽」とみなされる。

∘ 出力の場合、左から $w-1$ 個の空白が置かれ、右端のカラムに「`T`」あるいは「`F`」が対応する論理式の「真」「偽」によって出力される。

5. 文字型データ編集記述子

A 型編集記述子 ··· 文字型の入出力

A 型編集記述子は、文字型の入出力のための記述子で、その形式は次の形である。

Aw

対応する文字型変数「`c`」の文字長を l とするとき、

∘ 入力の場合、

・$w > l$ ならば、入力フィールドの最後から l 文字が取り出され、「`c`」に代入される。

・$w < l$ ならば、入力フィールドの w 文字が取り出され、「`c(1:w)`」に代入され、その後に l-w 個の空白が付加される。

・$w = l$、あるいは w が省略されたならば、入力フィールドの l 文字が取り出され、「`c`」に代入される。

∘ 出力の場合、

・$w > l$ ならば、「`c`」が入力フィールドの最後から l 文字に出力され、左に空白が詰められる。

・$w < l$ ならば、、「`c`」の先頭から w 文字が出力される。

・$w = l$、あるいは w が省略されたならば、「`c`」全体が出力される。

∘ フィールドの読み込み中にレコード (行) の終わりを検出したときは、足りない分は空白で補われる。*18

6. 組込型全般のデータ編集記述子

G 型編集記述子 ··· 全ての組込型の入出力

∘ G 型編集記述子の一般形は、次の形である。

G $w.d$ [Ee]

∘ 対応する入出力並びの式または変数が整数型のときの動作は、Iw と同じである。

∘ 対応する入力並びの式または変数が実数型のときの動作は、F$w.d$ と同じである。出力の場合は、その値によって F 型または E 型として出力される。すなわち出力値 r 値が、

$$0.1 \leqq |r| < 10^m$$

の範囲にあれば、有効数字を失わなうことなく少数だけで出力できるので F 型になり、右端の 4 カラムには空白が出力される。 $|r|$ がこの範囲にないときは、E 型で出力される。

∘ 対応する入出力並びの式または変数が論理型のときの動作は、Lw と同じである。

∘ 対応する入出力並びの式または変数が文字型のときの動作は、Aw と同じである。

7. 文字列編集記述子

文字定数編集記述子 ··· 文字定数の出力

∘ 「`'`」と「`'`」、あるいは「`"`」と「`"`」で囲まれた文字定数を出力する。入力には使われない。

∘ すでに述べたようにくり返し記号は使えないが、「`()`」でくくれば、反復させることはできる。

*18 `open` 文で「`pad='no'`」を指定してあれば、残りの文字数を次の行に読みに行く。

演習問題

演習問題 5

π を円周率とするとき、倍精度型の変数に 4π を代入し、以下の書式で出力すると、どのように出力されるか。

(1)「(f15.7)」 (2)「(e15.7)」 (3)「(d15.7)」 (4)「(es15.7)」 (5)「(en17.7)」 (6)「(g15.7)」

解答

(1) 「␣␣␣␣␣12.5663706」 (2) 「␣␣0.1256637E+02」 (3) 「␣␣0.1256637D+02」
(4) 「␣␣1.2566371E+01」 (5) 「␣␣␣12.5663706E+00」 (6) 「␣␣␣12.56637␣␣␣␣」

演習問題 6

次のデータを、それぞれ以下のプログラムで入出力を行ったとき、どのように出力されるか。

```
real :: a
read(*,'(7f5.2)') a
write(*,'(7f5.2)') a
```

(1) 「12345」 (2) 「-.234」 (3) 「␣1.2␣」 (4) 「␣␣13␣」 (5) 「␣.␣␣␣」 (6) 「1.2e1」

解答

(1) 「*****」 (2) 「-0.23」 (3) 「␣1.20」 (4) 「␣0.13」 (5) 「␣0.00」 (6) 「12.00」

演習問題 7

(1) 文字長 10 の文字型変数 str に、「abcdefghij」というデータを、「(a8)」、「(a12)」という書式でそれぞれ読み込んだとき、str には何が入るか。
(2) 文字長 10 の文字型変数 str に「abcdefghij」という値が入っているとき、「(a8)」、「(a12)」と言う書式でそれぞれ出力すると、何が出力されるか。

解答

(1) 「(a8)」→「abcdefgh␣␣」、「(a12)」→「cdefghij␣␣」
(2) 「(a8)」→「abcdefgh」、「(a12)」→「␣␣abcdefghij」

演習問題 8

整数型配列「j(1:10)」には、1 から 10 までの自然数が順に格納されている。これを以下の書式仕様で write 文を実行したとき、どのように出力されるか。

(1) write(*,'(i5(3i5))') j
(2) write(*,'(i5(3i5/))') j
(3) write(*,'(i5(3i5/2i5))') j
(4) write(*,'(i5,(3i5/2i5))') j
(5) write(*,'(i5,(3i5))') j
(6) write(*,'(i5,(2i5),i5)') j

解答

(1)
```
␣␣␣␣1␣␣␣␣2␣␣␣␣3␣␣␣␣4
␣␣␣␣5␣␣␣␣6␣␣␣␣7␣␣␣␣8
```
(2)
```
␣␣␣␣1␣␣␣␣2␣␣␣␣3␣␣␣␣4
_
```
(3)
```
␣␣␣␣1␣␣␣␣2␣␣␣␣3␣␣␣␣4
␣␣␣␣5␣␣␣␣6
```

```
    ␣␣␣␣9␣␣␣10
```

```
    ␣␣␣␣5␣␣␣␣6␣␣␣␣7␣␣␣␣8
-
    ␣␣␣␣9␣␣␣10
```

```
    ␣␣␣␣7␣␣␣␣8␣␣␣␣9␣␣␣10
```

```
(4) ␣␣␣␣1␣␣␣␣2␣␣␣␣3␣␣␣␣4
    ␣␣␣␣5␣␣␣␣6
    ␣␣␣␣7␣␣␣␣8␣␣␣␣9
    ␣␣␣10
```

```
(5) ␣␣␣␣1␣␣␣␣2␣␣␣␣3␣␣␣␣4
    ␣␣␣␣5␣␣␣␣6␣␣␣␣7
    ␣␣␣␣8␣␣␣␣9␣␣␣10
```

```
(6) ␣␣␣␣1␣␣␣␣2␣␣␣␣3␣␣␣␣4
    ␣␣␣␣5␣␣␣␣6␣␣␣␣7
    ␣␣␣␣8␣␣␣␣9␣␣␣10
```

§7.8 制御編集記述子

1. 位置付け (Tab) に関するもの

T 型編集記述子

○ T 型編集記述子は、行内で入出力を開始する位置を指定する記述子であり、次の形式である。

T*n*

ここで、*n* は 1 以上の十進数字であり、省略できない。

○ T 型編集子を使えば、入出力並びの順番にかかわらず、任意の順番でデータを入出力できる。

TL 型編集記述子

○ TL 型編集記述子は、直前に入出力を終えた位置より、左へ *n* カラムだけ移動した位置から、次の入出力を行うための記述子であり、次の形式である。

TL*n*

○ TL 型編集子を使えば、一つのデータから複数の変数の読みとりを行うことができる。

TR 型編集記述子

○ TR 型編集記述子は、直前に入出力を終えた位置より、右へ *n* カラムだけ移動した位置から、次の入出力を行うための記述子であり、次の形式である。

TR*n*

X 型編集記述子

○ X 型編集記述子は、*n* 個の空白を出力する記述子であり、次の形式である。

X*n*

○ X 型編集子が書式仕様の最後に書かれたときは、空白は出力されずに無視される。

これらの移動により、すでに書き込まれたフィールドに再び書き込むことになったときは、上書きされるので注意が必要である。

(例) $i = 111111$、$j = 222222$、$k = 333333$ であるとき、

「`write(20,'(t11,i10,tl20,i10,tl5,i10)') i, j, k`」

で出力すると、結果は「`␣␣␣␣2␣␣␣␣33333311111`」となる。

つまり最初の「`t11,i10`」で出力バッファの 11-20 カラムに *i* が「`␣␣␣␣111111`」と書き出され、出力ポイントは次の 21 カラムに移動する。次の「`tl20,i10`」で出力バッファの第 1 カラムに出力ポイントが戻り、1-10 カラムに *j* が書き出され、「`␣␣␣␣222222␣␣␣␣111111`」となる。最後に「`tl5,i10`」で出力ポイントが 11 から 6 に戻り、*k* が書き出された後、バッファが出力に送られて、上記の結果が表示される。

2. 正符号 (+) に関するもの

SP 型、SS 型、S 型編集記述子

○ 数値出力のとき、出力値が正であったときに「+」符号を出力するかどうかを指定する編集子であり、意味は次の通りである。

・ SP ⋯「+」を出力する。

・ SS ⋯「+」を出力しない。

・ S ⋯「+」を出力するかどうかを、処理系のデフォルトで決まる。

○ 書式仕様が開始されたときは、S 型編集子が選択された状態であり、上の三つのうちどれかが現れると、次にまたどれかが現れるか、書式仕様が終了するまで、その状態を保つ。

3. 空白の解釈に関するもの

BN 型、BZ 型編集記述子

○ 数値入力のとき、空白を「null」とみなすか、「0」とみなすかを指定する編集子であり、意味は次の通りである。

・ BN … 空白はすべて取り除き、あとは順に右詰めにし、空いた左側には空白を埋めたものと解釈して、データ編集子で読み込む。

・ BZ … 空白はすべて「0」であると解釈して、データ編集子で読み込む。

○ 書式仕様が開始されたときの状態は、BN の状態である。*19

(例) 「␣1␣2␣␣1␣.0␣21」　というデータから、整数型変数「i」と実数型変数「r」を、

「read(*,'(bn,i5,f8.0)') i, r」

で読みこめば、データは「␣␣␣12␣␣␣1.021」と解釈され、$i = 12$、$r = 1.021$ が読み込まれるが、

「read(*,'(bz,i5,f8.0)') i, r」

ならば、「01020010.0021」と解釈され、$i = 1020$、$r = 10.0021$ と読み込まれる。

4. 小数点の移動に関するもの

P 型編集記述子

○ 実数の入出力のとき、小数点の位置を実際の位置から移動させる編集子であり、次の形式である。

nP

n は 1 以上の整数か、または負の整数である。

○ この編集記述子に限り、nP と次の編集記述子の間の「,」は省略することができる。

○ P 型編集子の効果は外部表現 (入出力データ) が固定小数点表現の場合と、指数表現の場合とで異なる。

・ 外部表現が固定小数点表現のとき、実数型編集子 (F,E,D,ES,EN,G 型) による入力では、小数点が左に n だけずれたものとして、読み込まれる。

(例) 「␣␣␣1.2」 を、「2p,f6.1」で読み込むと、対応する変数には「0.012」として読み込まれる。

・ F 型編集子により固定小数点表現の出力を行うときは、小数点が右に n だけずれたものとして、書き出される。すなわち入力と効果が逆になる。

(例) 「1.1」を、「2p,f6.1」で書き出すと、「␣110.0」と書き出される。

ただし、G 型編集子による出力で、固定小数点表現になったときには、P 型編集子による影響はない。

・ 外部表現が指数表現のとき、実数型編集子による入力では、P 型編集子による影響はない。

・ 外部表現が指数表現のとき、E,D,G 型編集子による出力では、指数部の整数が n だけ引かれ、数値部の小数のケタが右に n ケタ移動する。すなわち数値としては不変である。

(例) 「e15.7」で書き出すと、「␣␣␣.1234567e+04」と出力される変数を、「1p,e15.7」で書き出すと、「␣␣1.2345670e+03」と書き出される。

このとき、小数点以下のけた数 d に対して、n は $-d < n < d+2$ の範囲になければならない。

○ 設定されたケタ移動幅は、次の P 型編集記述子が指定されるか、書式仕様の終わりまで効果を現す。

このように、P 型編集記述子は複雑な動作をするので、誤りの原因になりやすく、自分からは使用すべきでない。もともとは「1pg15.7」などとして、有効数字を表すためによく利用されたが、Fortran90/95 では ES 型編集記述子が利用できるので、それを使った方がよい。

---

*19 open 文で blank='zero' が指定されていれば、デフォルトは BZ となる。

5. 改行に関するもの

斜線編集記述子

入出力の際に、改行を指定するための編集記述子で、次の形である。

```
/
```

- 「/」の前後の「,」は省略できる。
- 入力の場合、「/」があれば、その行からの入力を打ち切り、以下の入力は次の行から行う。
- 出力の場合、「/」があれば、改行コードを出力し、以下の出力は次の行へ行う。
- 2 行以上続けて改行したいときは、連続して指定する。「//」なら、1 行空行になる。
- 書式指定の先頭にあった場合、空行のまま 1 行改行する。

コロン編集記述子

入出力の際に、最後の入出力並びの処理が終わった時点で、入出力を打ち切るための編集記述子で、次の形である。

```
:
```

- 「:」の前後の「,」は省略できる。

(例)

```
write(*,'(3(i5,","))') i, j, k
```

と書式仕様を指定すると、最後の k の内容を出力した後に「,」を出力してしまうが、

```
write(*,'(3(i5:","))') i, j, k
```

と指定すれば、「i5」を変換した後に出力を終了する。

§7.9 データ転送入出力文

この節では、順編成書式付きファイルの、データ転送入出力文の一般形について述べる。

1. read 文

read 文の一般形式

順編成書式付きファイルの read 文は、次の形式である。

read([unit=]*un*[,fmt=]*form*[,para-list]) input-list

○「[unit=]*un*」は必須であり、省略することはできない。read(の直後の第一パラメータであるときには「unit=」の部分を省略でき、「*un*」だけでよい。

○「[fmt=]*format*」は順編成書式付きファイルでは、省略することはできない。その位置が、read 文の第二パラメータであり、第一パラメータが「unit=」の部分を省略した「*un*」であるときには、「fmt=」の部分を省略でき、「*format*」だけでよい。

○「[,para-list]」は、「,」で区切られた「para=*parameter*」の形式の並びであり、必要に応じて指定する。「para=」の部分は省略することはできない。デフォルト値があるものは、下線で示す。

○「input-list」は、変数および do 型くり返しより構成される。

(1) 装置番号 [unit=]*un*

○「*un*」は open 文でファイルに結合された装置番号を表す整数式、または「*」。

・「*」は標準入力を指定する記号である。

(2) 書式仕様の指定 [fmt=]*format*

○「*format*」で、書式仕様を指定する。指定の方法は書式仕様の項を参照すること。

(3) ファイル終了記録の検出 end=*stmt*

○入力動作開始直後に、EOF (ファイル終了記録) を検出した場合、入力動作を中止して文番号 *stmt* の文に制御を移す。

```
  n=0
  do
    read(12, '(i5)', end=90) a(n+1)
    n=n+1
  enddo
 90 write(*,*) n, " data read"
```

・read 文の実行開始時に EOF に遭遇したとき、end 指定子も、iostat 指定子もない場合、および read 文の実行中にに EOF に遭遇したとき、iostat 指定子がない場合、プログラムの実行は停止する。

(4) 入力エラーの検出 err=*stmt*

○入力動作中に、何らかの原因で入力が失敗した場合、入力動作を中止して文番号 *stmt* の文に制御を移す。

```
    read(21,'(i5)',err=99) j
                |
 99 write(*,*) " data read error"
```

・read 文の実行時にエラーを起こしたとき、err 指定子も、iostat 指定子もない場合、プログラムの実行は停止する。

(5) 停留入力 advance={ 'yes' | 'no' } (90/95)

○ read 文の実行がレコードの途中まで読み込んだところで終了すると、通常はそのレコードのその後のデータは読み飛ばされる。advance 指定子により、これを抑止することができる (出力についても同様の

指定がある)。これを**停留入出力** という。

· 'yes' を指定すると、read 文の終了後、入力のカレント・ポジションは、次の EOR の直後に移動する。すなわち次の read 文は、次の行の先頭から読み始める。

· 'no' を指定すると、read 文が終了しても、入力のカレント・ポジションは移動しない。次の read 文は、レコードにデータが残っていれば、それから読み始める。

· 停留入力は、並び入力文では指定できない。書式付き入力文でしか指定できない。

· 停留入力中に記録が終了したとき、eor 指定子も、iostat 指定子もない場合、プログラムの実行は停止する。

(6) 読み込んだバイト数 size=*ic*

○ 停留入力文により読み込んだバイト数を得るための指定子で、*ic* には基本種別の整数型変数を指定する。

· 停留入力文が正常に終了したとき、読み込まれたバイト数を *ic* に返す。

(7) 改行コードの検出 eor=*stmt*

○ 停留入力中に、改行コード (EOR) を検出した場合、入力動作を中止して文番号 *stmt* の文に制御を移す。[*20]

```
  n=0
  do
    read(10, '(i5)', eor=90, advance='no', size=ic) a(n+1)
    n=n+1
    write(*,*) n, ic
  enddo
 90 write(*,*) " n data read"
```

· 停留入力中に記録が終了したとき、eor 指定子も、iostat 指定子もない場合、プログラムの実行は停止する。

(8) エラー・コードの検出 iostat=*stmt*

○ read 文が正常に動作したかどうかを知るための指定子で、*ist* には基本種別の整数型変数を指定する。

```
  integer :: j, ios
  open(20, file='test3.txt')
  do
    read(20, '(i5)', eor=90, end=99, iostat=ios, advance='no') j
    write(*,*) j
    cycle
 90 write(*,*) "eor encounterd",ios
  enddo
 99 write(*,*) "eof encounterd",ios
  close(20)
```

· read 文が正常に実行された場合、*ist* には 0 が代入される。

· read 文の実行時にエラーが起きた場合、*ist* にはエラーコードを示す、正の値が代入される。その値は処理系依存である。

· read 文は正常に終了したが、EOR か EOF を読み込んだとき、*ist* には負の値が代入される。その値は処理系依存である。[*21]

[*20] open 文で pad='no' を指定したとき、改行コードを読み込むとエラーになる。そのときは、eor 指定子ではなく、err 指定子でエラー・ハンドルを行わなければならない。

[*21] Intel Fortran の場合、EOR を読めば「-2」、EOF を読めば「-1」が *ist* に代入される。

2. 入力並び

入力並びは、入力項目を「a, b, c」のように、コンマ「,」で区切って並べたものである。入力項目としては、以下の項目が許される。

スカラー変数

スカラー変数とは配列でない変数のことで、整数型、実数型、複素数型、論理型、文字型などの変数が項目として可能であるが、定数や式は不可である。

配列変数

配列変数も入力項目として可能である。この場合配列の全要素を、添え字の順序に従って入力することになる。

また、配列要素単独での入力も可能である。この場合、その項目の入力が実行されるまでに、添え字の値が決まっていればよい。すなわち、「2,1,12」というデータに対して、

```
integer :: i,j,m(2,2)
         |
read(*,*) i,j,m(i,j)
```

という read 文を実行すると、m(2,1) に 12 が代入される。

DO 型入力並び (implied-DO list)

次の形式も入力項目として可能である。

（変数 1[, 変数 2…], DO 制御変数=初期値, 終値 [, 増分]）

DO 制御変数の動作は、DO-LOOP と同じである。また、変数としては、配列要素も許され、以下のような文が可能である。

```
read(*,*) (a(i), b(2,i), i=1,5,2)
```

この場合、a(1)、b(2,1)、a(3)、b(2,3)、a(5)、b(2,5) の順に、入力ファイルから読み込まれる。

また、DO の初期値等の値は、その項目の入力が実行されるまでに決まっていればよいので、

```
read(*,*) n, (a(i), i=1,n)
```

という文が可能である。この場合、n、a(1)、a(2)、···、a(n) の順に、入力ファイルから読み込まれる。

入力並びの禁止事項

入力の場合、次のことは禁止されている。

・ 同じ変数が、一つの入力並びの中に複数回現れること
・ (使用中の) DO 制御変数が入力並びの中に現れること

3. write 文

read 文の一般形式

順編成書式付きファイルの write 文は、次の形式である。

write([unit=]*un*[,fmt=]*form*[,para-list]) output-list

○「[unit=]*un*」は必須であり、省略することはできない。write(の直後の第一パラメータであるときには「unit=」の部分を省略でき、「*un*」だけでよい。

○「[fmt=]*format*」は順編成書式付きファイルでは、省略することはできない。その位置が、write 文の第二パラメータであり、第一パラメータが「unit=」の部分を省略した「*un*」であるときには、「fmt=」の部分を省略でき、「*format*」だけでよい。

○「[,para-list]」は、「,」で区切られた「para=*parameter*」の形式の並びであり、read 文と使い方は同一だが、指定できるものは「err」、「advance」、「iostat」の 3 個だけである。「eor」、「end」、「size」は指定できない。
○「output-list」は、式および do 型くり返しより構成される。

(1) 装置番号 [unit=]*un*
○ read 文と使い方は同じである。

(2) 書式仕様の指定 [fmt=]*format*
○ read 文と使い方は同じである。

(3) 出力エラーの検出 err=*stmt*
○ read 文と使い方は同じである。

(4) 停留出力 advance={ 'yes' | 'no' } (90/95)
○ write 文の実行が終了すると、通常はそこで改行する。advance 指定子により、これを抑止することができる。
· 'yes' を指定すると、read 文の終了後、改行コード (EOR) を出力し、出力のカレント・ポジションは、その直後となる。すなわち次の write 文は、改行して次の行の先頭から書き始める。
· 'no' を指定すると、read 文が終了しても、改行コードを出力しない。したがって出力のカレント・ポジションは移動しない。次の write 文は、同じレコードに続けて書き始める。
· 停留出力は、並び出力文では指定できない。書式付き出力文でしか指定できない。

(5) エラー・コードの検出 iostat=*stmt*
○ write 文が正常に動作したかどうかを知るための指定子で、*ist* には基本種別の整数型変数を指定する。

```
 integer :: i, ios
 open(30,file='test3.out')
 do i=1, 10
   write(30, '(i2)', err=90, iostat=ios, advance='no') i
   cycle
90 write(*,*) "write error at ",i , "error code= ", ios
 enddo
 close(30)
```

· read 文が正常に実行された場合、*ist* には 0 が代入される。
· read 文の実行時にエラーが起きた場合、*ist* にはエラーコードを示す、正の値が代入される。その値は処理系依存である。

4. 出力並び

出力並びの項目は、基本的には入力並びと同様であるが、以下の違いがある。
・ 出力項目は、変数だけでなく、定数や式も可能である。
・ 入力並びの禁止事項は適用されない。
したがって、以下のような文も可能である。

```
write(*,*) (a, i*2, i=1,2)
```

この文を実行すると、a(の内容)，2，a(の内容)，4 の順に出力される。

7 章 の 章 末 問 題

【問題 7-1】

(1)「abcde」という文字列をファイルから読み込み、1 文字 (1 バイト) ずつ分解して、コードを 16 進で出力するプログラムを作成しなさい。

(2)「漢字」という文字列をファイルから読み込み、同じく 1 バイトずつ分解して、コードを 16 進で出力するプログラムを作成しなさい。

(3) Linux の iconv、あるいは Windows のエディタで、「漢字」の漢字コードを JIS(ISO2022JP)、SJIS、EUC(EUCJP) に変換して、その漢字コードを同様に調べなさい。

【問題 7-2】

停留出力文を使い、「5*6=30」の形式を、横に 9 列、縦に 9 行並べて、下の九九の表を出力しなさい。

```
1*1= 1 2*1= 2 3*1= 3 4*1= 4 5*1= 5 6*1= 6 7*1= 7 8*1= 8 9*1= 9
1*2= 2 2*2= 4 3*2= 6 4*2= 8 5*2=10 6*2=12 7*2=14 8*2=16 9*2=18
1*3= 3 2*3= 6 3*3= 9 4*3=12 5*3=15 6*3=18 7*3=21 8*3=24 9*3=27
1*4= 4 2*4= 8 3*4=12 4*4=16 5*4=20 6*4=24 7*4=28 8*4=32 9*4=36
1*5= 5 2*5=10 3*5=15 4*5=20 5*5=25 6*5=30 7*5=35 8*5=40 9*5=45
1*6= 6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36 7*6=42 8*6=48 9*6=54
1*7= 7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49 8*7=56 9*7=63
1*8= 8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64 9*8=72
1*9= 9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```

【問題 7-3】

1 行に 5 ケタの数値がいくつか記されているファイル、「exercise2.txt」がある。

(例)　「␣␣␣12␣␣␣34␣␣␣56␣␣␣78」

停留入力文と eor 指定子を使って、データの個数「n」を調べ、整数型配列「k」に入力データを格納しなさい。最後に「n」および入力した「k」を出力しなさい。

【問題 7-4】

前問と同じ処理を、停留入力文を使用せずに、1 行を文字型変数に読み込むことにより、内部入出力文を使って行いなさい。

【問題 7-5】

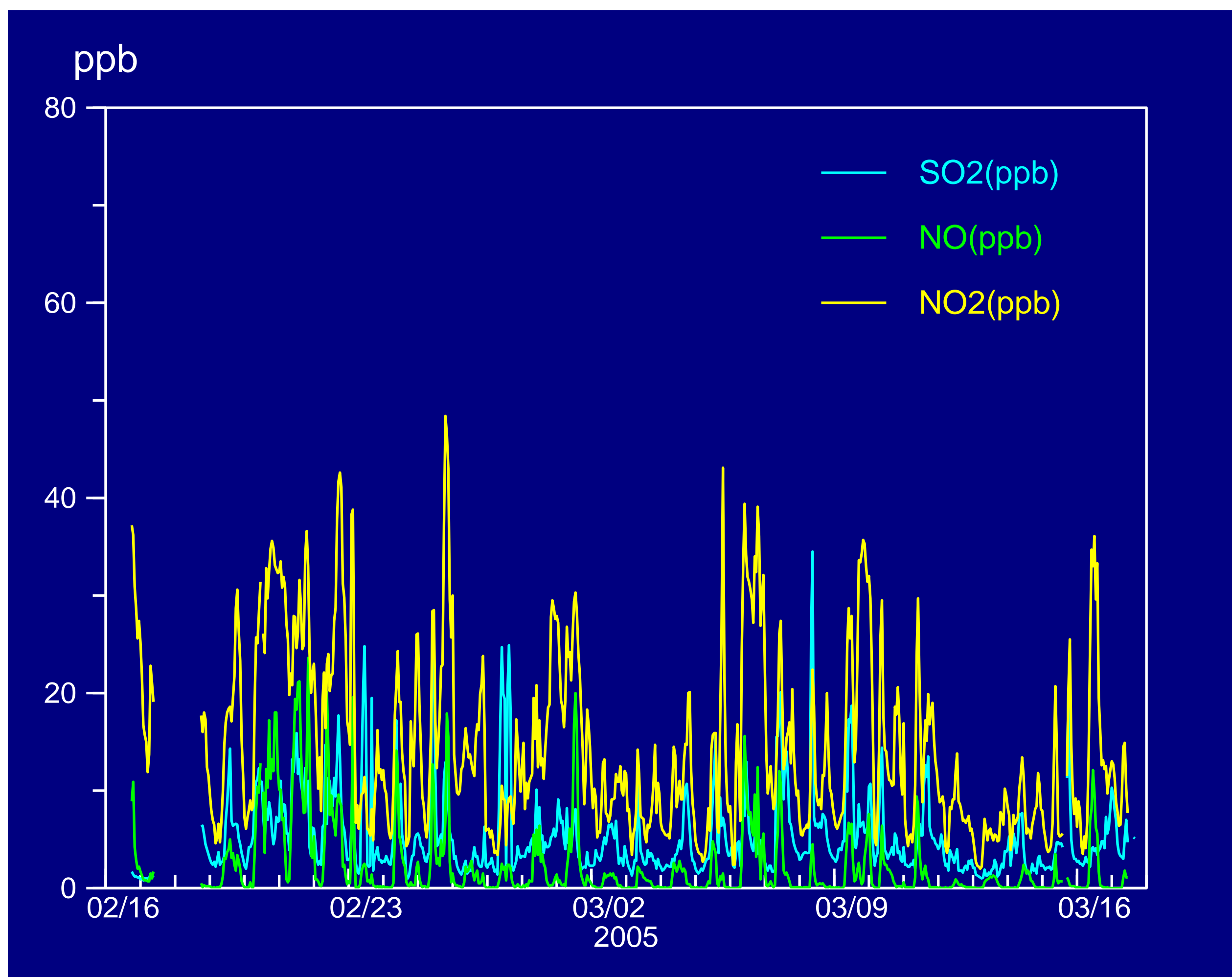
「exercie5.csv」は、データがコンマで区切られたファイルであるが、「/」があるため並び入出力文では読めない。

(例)　abc,def,g/h

書式付き read 文で 1 行を読みとり、コンマの位置を調べて分割し、3 つの文字列をそれぞれ出力しなさい。

eps_plot Command Reference ver.2.24

Fortran90/95 Graphic Subroutine Package



凡例

- ○ [xxx] ··· optional 引数であり、省略可能
- ○ {yyy|zzz} ··· alternative, どちらかを選択
- ○ integer、real、character、logical ··· それぞれ、基本整数型、基本実数型、文字型、論理型
- ○ structure、integer(i_8)、real(r_8) ··· それぞれ、構造型、長整数型、倍精度実数型
- ○ 「→aaa」··· optional 引数を省略した場合のデフォルト値

水色の枠のある文字列をクリックすれば対応するアンカーに飛ぶ。

目　次

■ 1　画像ファイルの設定

plots (図形描画の開始)　1
factor (座標系の設定 [1])　2
xviewp (座標系の設定 [2])　3
plote (図形描画の終了)　4

■ 2　直線・曲線の描画

plot (指定した座標までペンで線分を引く)　5
pline(1) (始点と終点の座標から線分を引く)　6
pline(2) (x、y 座標から折れ線を書く)　7
pbezier (二次、三次ベジェ曲線を描く)　8

■ 3　ペン番号と線の属性変更

newpen (ペン番号を指定して属性変更)　9
penmode (実線・破線の線種を指定)　10
penmodeset (別名 dashs) (線種の定義、登録)　11
pencolor(1) (RGB 整数値による線色の指定)　12
pencolor(2) (h、s、b 値による線色の指定)　13
pencolor(3) (R、G、B 値による線色の指定)　14
hatchset (斜線パターンの定義、登録)　15
dotset (点パターンの定義、登録)　16

■ 4　文字、数字、記号を書く

symbol (文字を書く)　17
kanji (漢字を書く)　18
number (数値を与えて数字を書く)　19
pfont (フォントの変更)　20
center (センター・シンボルを書く)　21

■ 5　図形の描画 (1) 円、楕円

pcircle (円・円弧の描画)　22
pellipse (楕円・楕円弧の描画)　23
tcircle (三点を通る円弧の描画)　24

■ 6　図形の描画 (2) 多角形

plinec (多角形の x、y 座標による描画)　25
prect (長方形の描画)　26
ptriangle (二等辺三角形の描画)　27
polygon (多角形の相対座標による描画)　28

■ 7　図形の描画 (3) 矢印

arrow (始点と方向を指定した矢印の描画)　29
pvector (始点と終点を指定した矢印の描画)　30

■ 8　二次元グラフと座標 (1) 標準

paxis (任意方向の軸の設定-通常)　31
pxaxis (x 軸の設定-通常)　32
pyaxis (y　軸の設定-通常)　33
xygrid (x 軸、y 軸に平行な格子線を描く)　34
xyline (x 軸、y 軸による二次元グラフ)　35

■ 9　二次元グラフと座標 (2) 常用対数軸

laxis (任意方向の軸の設定-常用対数)　36
lxaxis (x 軸の設定-常用対数)　37
lyaxis (y　軸の設定-常用対数)　38
○片対数、両対数グラフの使用例　40

■ 10　二次元グラフと座標 (3) 時間軸

txaxis (時間軸-横軸-の設定)　41
tygrid (時間軸、y 軸に平行な格子線を描く)　42
tyline (t 軸、y 軸による二次元グラフ)　43

■ 11　その他

subframe_begin (一時的な座標移動の開始)　44
subframe_end (一時的な座標移動の終了)　44
clip_begin (多角形領域へのクリッピング開始) 45
clip_end (多角形領域へのクリッピング終了)　45

■ 12　暦関係の副プログラム

grtowd (年月日時分秒から、Date 型への変換関数)　46
wdtogr (Date 型から、年月日時分秒への変換関数)　46
ldtogr (長整数型から、年月日時分秒への変換関数)　47
add_time (年月日時分秒型時刻データの加算)　47
day_month (指定された年、月の日数)　47

■ 1 画像ファイルの設定 ■

□ plots (図形描画の開始)

1. 呼び出し形式

```
call plots({[width][,height]|[,paper]}[,file][,backcolor][,unitnum][,code])
```

2. 引数の説明

○ `width` (real,optional) ··· 図の横幅 (単位 cm)、 最大 3000pt.=105.8cm。

○ `height` (real,optional) ··· 図の縦の長さ (単位 cm)、 最大 4000pt=141.1cm。

· width と height のどちらかを負の値にすると、図の枠が出力される。

· どちらかが省略されたときの図の大きさは、次の `paper` パラメータできまる。

○ `paper` (character,optional → [letter size]) ··· ··· 図の大きさを JIS の名前で指定する。

· { `"A4"|"A4L"|"B4"|"B4L"|"B5"|"B5L"` } から 1 つ選択。`L` が最後に付くものは横置き、他は縦置きである。大文字、小文字は問わない。

○ `file` (character,optional → `"eps_plot.eps"`) ··· 図を出力するファイル名

○ `backcolor` (integer,optional → `RGB_white` [白色]) ··· 背景色。

○ `unitnum` (integer,optional → 3) ··· 図形ファイルを出力する機番。

○ `code` (character,optional → `"utf8"`) ··· 漢字コード。

· { `"jis"|"sjis"|"euc"|"utf8"` } から 1 つ選択。大文字、小文字は問わない。

3. 使用法

· 図形出力を開始する際に、他の `eps_plot` サブルーチンを呼ぶ前に、必ず 1 回だけ `call` する。

4. 使用例

eps_plot

subroutine package

Fig.1-1 program plots_example

目次へ

□ factor (座標系の設定 [1])

1. 呼び出し形式

```
call factor(factx[,facty])
```

2. 引数の説明

○ `factx` (real) ··· 図形を横方向に、現在の大きさの `factx` 倍にする。

○ `factx` (real,optional → `=factx`) ··· 図形を縦方向に、現在の大きさの `facty` 倍にする。

3. 使用法

· 初期の図形の `factx,facty` 倍でなく、現在の図形の `factx,facty` 倍である。

· 最後に `call plot(x,y,-3)` で決められた、座標原点を中心に拡大・縮小する。

· eps 形式のベクトルデータは x、y 方向の比率を保ったまま拡大・縮小できるので、その場合特に `factor` を使用する必要はない。どちらかの倍率だけを変える際に使う。

· factor を call すると、図形の比率だけでなく、座標の比率も変わってしまうので、注意が必要である。

· これらのことから、`plots` 直後以外には、あまり call すべきではない。

4. 使用例

Fig.1-2 program factor_example

目次へ

□ xviewp (座標系の設定 [2])

1. 呼び出し形式

```
call xviewp(xmin,ymin,xmax,ymax)
```

2. 引数の説明

○ `xmin` (real) ··· 用紙の左側の x 座標。

○ `xmax` (real) ··· 用紙の右側の x 座標。

○ `ymin` (real) ··· 用紙の下側の y 座標。

○ `ymax` (real) ··· 用紙の上側の y 座標。

3. 使用法

· 用紙に対して、引数の説明で述べたように、x、y 座標を定める。

· 実行後の結果は `factor` と似ているが次の違いがある。`factor` は図形を全体として拡大、縮小するので、線幅が変わる。`xviewp` は座標だけを拡大、縮小するので、線幅は不変である。これらの相違を、使用例で示す。一番左が元の図形、中が `factor`、右が `xviewp` により拡大した図である。

4. 使用例

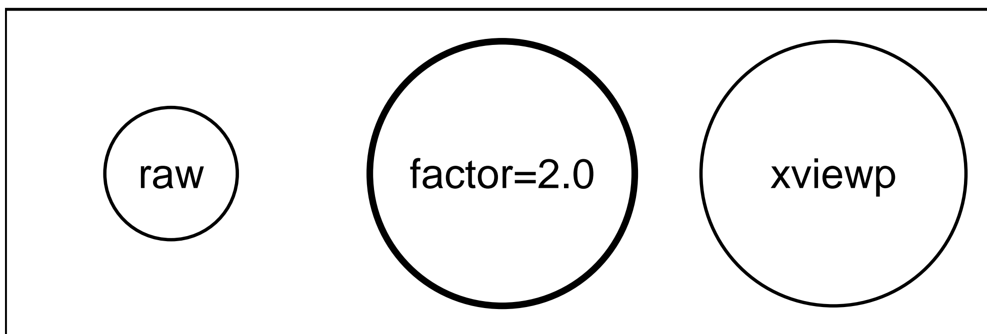


Fig.1-3 program xviewp_example

目次へ

□ plote (図形描画の終了)

1. 呼び出し形式

```
call plote([end_char])
```

2. 引数の説明

○ `end_char` (character,optional) · · · 描画終了時に、標準出力に `end_char` を出力。

3. 使用法

· 描画の終了時に 1 回だけコールする。

· `call plot(x,y,999)` でも同じ動作をする。

· 1 つのプログラムで連続して複数枚の図を書きたいときは、いったん `plote` をコールして描画出力を終了させ、改めて別ファイルで `plots` をコールする。

省略 (他のサブルーチンの使用例を参照)

目次へ

■ 2 直線・曲線の描画 ■

□ plot (指定した座標までペンで線分を引く)

1. 呼び出し形式

```
call plot(x,y[,mp])
```

2. 引数の説明

○ x (real) ··· 線を引く先の点の x 座標

○ y (real) ··· 線を引く先の点の y 座標

○ mp (integer,optional → 2) ··· ペン移動モード

· mp = 2 → ペンダウンモード。カレント・ポイントから (x,y) まで線を引く。動作後、カレント・ポイントは (x,y) に移る。

· mp = 3 → ペンアップモード。カレント・ポイントは (x,y) に移るが、線は引かれない。

· mp = -3 → 原点移動。(x,y) が新しい原点となる。カレント・ポイントは消滅する。

· mp = 999 → call plote と同じ。ある種のプロッター・サブルーチン・パッケージとの互換用。これをコールしたときは call plote をコールしてはいけない。

3. 使用法

· まず引きたい線の始点の座標を mp=3 でコールして、カレントポイントを作成する。

· 次に引きたい線の終点の座標を mp=2 でコールして、線を引く。

· 連続して plot を mp=2 でコールして、次々に線を延ばしていくことができる。このとき、接続点には端点処理が行われ、また破線 (penmode) のパターンも継続する。間に他の処理 (下図の場合はペン色の交換) を入れた場合には、そのような処理は行われない。

4. 使用例

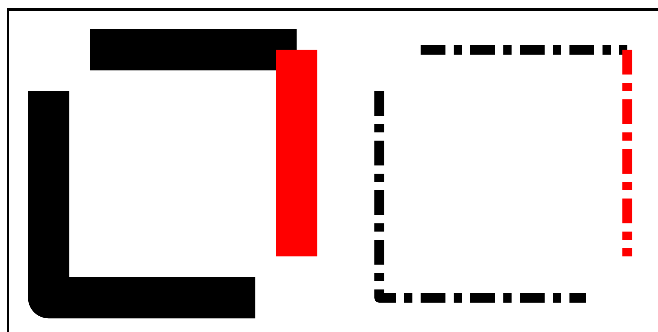


Fig.2-1 program plot_example

目次へ

□ pline(1) (始点と終点の座標から線分を引く)

1. 呼び出し形式

```
call pline(x1,y1,x2,y2[,penmode][,pencolor][,penwidth])
```

2. 引数の説明

○ `x1` (real) ··· 始点の x 座標。

○ `y1` (real) ··· 始点の y 座標。

○ `x2` (real) ··· 終点の x 座標。

○ `y2` (real) ··· 終点の y 座標。

○ `penmode` (integer,optional → unchanged) ··· 線種の指定 (User's manual 参照)。

○ `pencolor` (integer,optional → unchanged) ··· 線の色の指定 (User's manual 参照)。

○ `penwidth` (integer,optional → unchanged) ··· 線の幅の指定 (User's manual 参照)。

3. 使用法

· 座標 $(x1, y1)$ を始点、座標 $(x2, y2)$ を終点とする線分を、`pencolor` 色で描く。連続して `pline` を呼び出しても、端点処理や破線パターンの継続は行われない。それには `pline(2)` を call しなければならない。

`penmode`、`pencolor`、`penwidth` の指定は、このコマンドだけで有効であり、次には引き継がれない。

4. 使用例

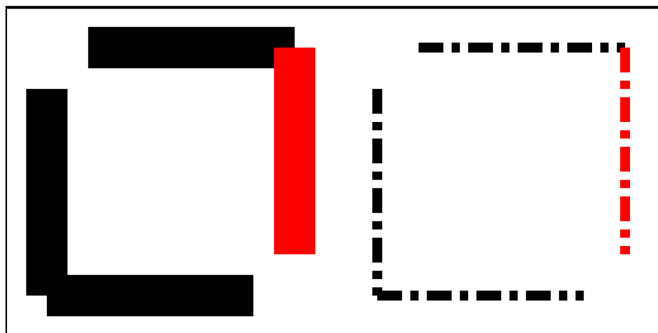


Fig.2-2 program pline1_example

目次へ

□ pline(2) (x、y 座標から折れ線を書く)

1. 呼び出し形式

(a) `call pline(x_array,y_array[,n][,penmode][,pencolor][,penwidth][,fillcolor][,hatch])`

または、

(b) `call pline(xy_array[,n][,penmode][,pencolor][,penwidth][,fillcolor][,hatch])`

2. 引数の説明

○ `x_array` (real,array) ··· x 座標の 1 次元配列。

○ `y_array` (real,array) ··· y 座標の 1 次元配列。

○ `xy_array` (structure,array) ··· 構造体 `coord_xy` の 1 次元配列。

○ `n` (integer,optional → [min(size(x_array),size(y_array))]) ··· 使用する座標の組の個数。

○ `penmode` (integer,optional → unchanged) ··· 線種の指定 (User's manual 参照)。

○ `pencolor` (integer,optional) ··· 線の色 (User's manual 参照)。

○ `penwidth` (integer,optional → unchanged) ··· 線の幅の指定 (User's manual 参照)。

○ `fillcolor` (integer,optional) ··· 塗りつぶし色 (User's manual 参照)。

○ `hatch` (integer,optional) ··· テクスチャ・パターン番号 (同上)。

3. 使用法

(a) · 座標 `(x_array(1),y_array(1))` を始点として、`(x_array(2),y_array(2))`、`(x_array(3),y_array(3))` ··· `(x_array(n),y_array(n))` まで連続した折れ線を引く。

· `n` が指定されていない場合、配列 `x_array`、`y_array` の次元の小さい方が `n` となる。

(b) · 座標 `xy_array(1)` を始点として、`xy_array(2)`、 ··· `xy_array(n)` まで連続した折れ線を引く。

· 構造体 `coord_xy` は以下のように、座標 (x,y) を要素に持つ構造体である。

```
type, public :: coord_xy
  real       :: x, y
end type coord_xy
```

· `n` が指定されていない場合、配列 `xy_array` の次元が `n` となる。

(共通) · その間端点処理や破線パターンの継続が行われる。

· `fillcolor` または `hatch` のどちらかが指定された場合、終点と始点が結ばれ、どちらも端末処理される。

· 塗りつぶしやテクスチャの貼り付けなしに終点と始点を結びたいならば、`fillcolor=-99` とダミーカラーを指定する。あるいは `plinec` を call する。

4. 使用例

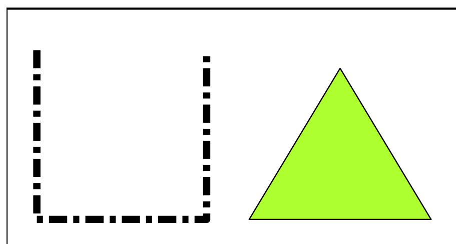


Fig.2-3 program pline2_example

目次へ

□ pbezier (二次、三次ベジェ曲線を描く)

1. 呼び出し形式

```
call pbezier(x1,y1,x2,y2,x3,y3[,x4][,y4])
```

2. 引数の説明

○ x1 (real) · · · 始点の x 座標。

○ y1 (real) · · · 始点の y 座標。

○ x2 (real) · · · 制御点 1 の x 座標。

○ y2 (real) · · · 制御点 1 の y 座標。

○ x3 (real) · · · 終点の x 座標。x4,y4 が与えられたときは、制御点 2 の x 座標。

○ y3 (real) · · · 終点の y 座標。x4,y4 が与えられたときは、制御点 2 の y 座標。

○ x4 (real,optional) · · · 終点の x 座標。

○ y4 (real,optional) · · · 終点の y 座標。

3. 使用法

· x1,y1,x2,y2,x3,y3 の 6 個の引数が与えられたときは、座標 (x1,y1) を始点、座標 (x2,y2) を制御点、座標 (x3,y3) を終点とする、二次のベジェ曲線を書く。

· x1,y1,x2,y2,x3,y3,x4,y4 の 8 個の引数が与えられたときは、座標 (x1,y1) を始点、座標 (x2,y2) と座標 (x3,y3) を制御点、座標 (x4,y4) を終点とする、三次のベジェ曲線を書く。

4. 使用例

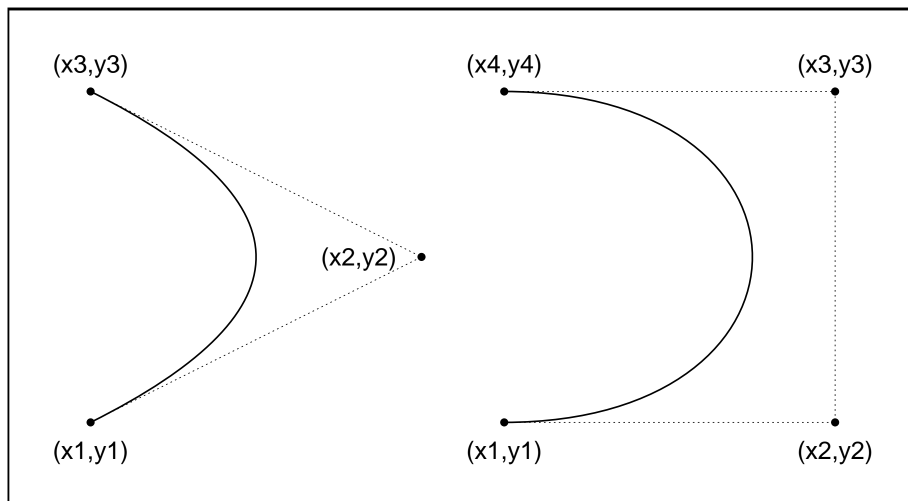


Fig.2-4 program pbezier_example

目次へ

■ 3 ペン番号と線の属性変更 ■

□ newpen (ペン番号を指定して属性変更)

1. 呼び出し形式

```
call newpen([ipen][,penmode][,pencolor][,penwidth] &
& [,linecap][,linejoin][,miterlimit])
```

2. 引数の説明

○ ipen (integer.optional → current_pen (現在使用しているペン)) ··· 属性を変更するペン番号を、1-50 の範囲で指定する。指定を省略した場合は現在のペンに対して適用、0 を指定した場合はすべてのペンに対して適用する。

○ penmode (integer,optional) ··· 線種の指定。詳細は subroutine penmode の説明を参照。

○ pencolor (integer,optional) ··· 線の色の指定。詳細は subroutine pencolor の説明を参照。

○ penwidth (integer,optional) ··· 線の幅の指定。単位は 0.01cm。

○ linecap (integer,optional) ··· 線の端点の形状の指定。数値の意味は、User's manual を参照。

○ linejoin (integer,optional) ··· 線の接続の形状の指定。数値の意味は、User's manual を参照。

○ miterlimit (integer,optional) ··· linejoin=0 の場合のマイター接続からベベル接続に変わる限界を指定。数値の意味は、User's manual を参照。

3. 使用法

· User's manual で述べたように、機械式プロッタ的な使い方と、プリンタ・プロッタ的な使い方がある。

· 機械式プロッタ的な使い方では、1-50 のペンにあらかじめ newpen の引数で属性を定義し、必要に応じてペンを newpen で呼び出して描画する。

· プリンタ・プロッタ的な使い方では、newpen の引数は使わず、線幅以外の属性の指定はすべてのペンに対して同時に行う。このため、newpen で呼び出すペン番号が、線幅指定と同値になる。

4. 使用例

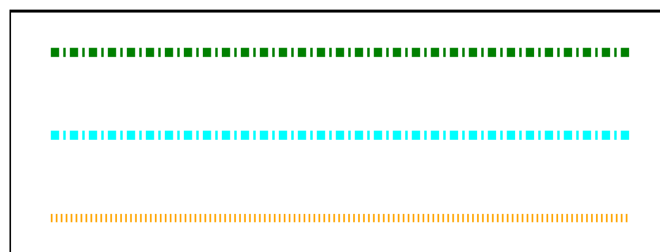


Fig.3-1 program newpen_example

目次へ

□ penmode (実線・破線の線種を指定)

1. 呼び出し形式

```
call penmode(mode)
```

2. 引数の説明

○ `mode` (integer) · · · 線種を 1-20 の範囲で指定する。

3. 使用法

· `penmode=1-10` は、あらかじめプログラムで規定されている (下図)。

· `penmode=11-20` はユーザ定義領域であり、ユーザはサブルーチン `penmodeset`(別名 `dashs`) を呼び出して作成した破線パターンをこの範囲に登録しておいて呼び出す。`plots` が呼ばれた直後には、`penmode=1` と同じパターン、すなわち実線が指定されている。

4. 使用例

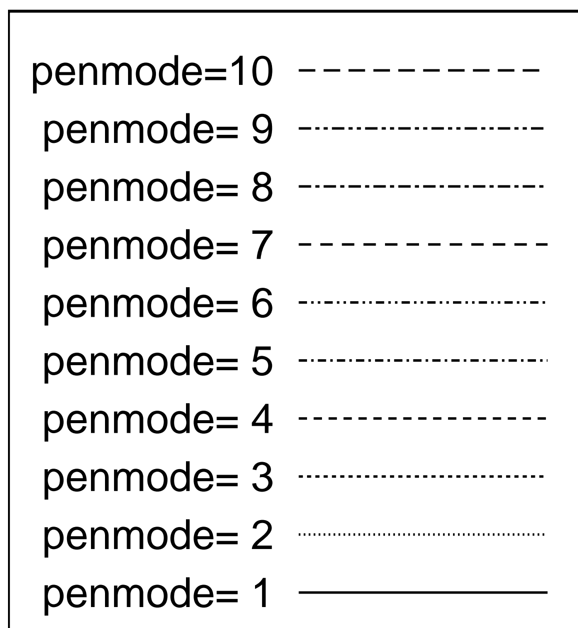


Fig.3-2 program penmode_example

目次へ

□ penmodeset (線種の定義、登録)　(別名 dashs)

1. 呼び出し形式

```
call penmodeset(d_array[,nd][,penmode])
```

または

```
call dashs(d_array[,nd][,penmode])
```

2. 引数の説明

○ `d_array` (real,array) ··· 破線のパターンを格納しておく配列。

○ `nd` (integer,optional → size(d_array)) ··· d_array のうち、パターン定義に使用する要素数。

○ `penmode` (integer,optional) ··· 定義したパターンを登録する番号 `1-20`。

3. 使用法

· `dashs` は CALCOMP 系のアプリケーションに整合するためにつけた別名である。

· `d_array` は破線パターンを定義する配列であるが、二つの定義方法がある。

· CALCOMP 系の定義方法では、`d_array` に、線を引く部分の長さをマイナス値で、間隔を開ける部分の長さをプラス値で、cm 単位で交互に配列要素に代入しておく。

· Postscript の定義方法では、`d_array` に、線を引く部分、間隔を開ける部分、線を引く部分 ··· の順に、長さを cm 単位で交互に配列要素に代入しておく。

· どちらの定義方法であるかは、マイナス値があるかどうかで判断する。

· `d_array` のうち添え字 1 から添え字 `nd` までの配列要素が、パターン定義に使われる。

· `nd` を省略した場合、`d_array` の可能な全配列要素がパターン定義に使われる。

· `penmode` は、作成した破線パターンを保存したい場合に、その番号を指定する。

· 前出のように、`penmode` を保存する番号は 1 から 20 までである。指定された番号のパターンが、作成されたパターンに置き換わる。

4. 使用例

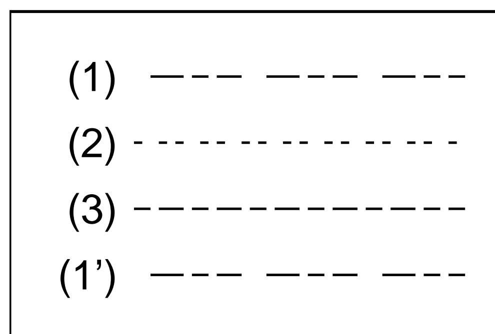


Fig.3-3 program penmodeset_example

目次へ

□ pencolor(1) (RGB 整数値による線色の指定)

1. 呼び出し形式

```
call pencolor(icode[,usercolor])
```

2. 引数の説明

○ `icode` (integer) · · · 変更する線色を示す基本型整数。

○ `usercolor` (integer,optional) · · · 変更後の線色を、ユーザー指定色として登録する。

3. 使用法

· 変更する色を、RGB 表現でそれぞれ 0 から 255 までの 256 段階で表したものをそれぞれ、`rr`、`gg`、`bb` とするとき、`icode=rr*256*256+gg*256+bb` と対応する。

· `usercolor` は、$-3 >$ `usercolor` $\geqq -32$ の範囲で指定する。

4. 使用例

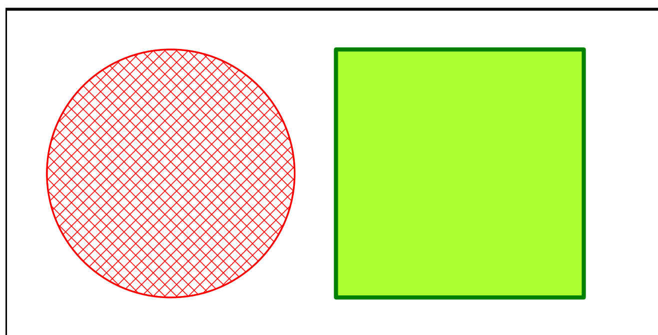


Fig.3-4 program pencolor1_example

目次へ

□ pencolor(2) (h、s、b 値による線色の指定)

1. 呼び出し形式

```
call pencolor(hue, saturation,brightness,usercolor[,usercolor])
```

2. 引数の説明

○ `hue` (real) ··· 変更する色の `hue` 値。

○ `saturation` (real) ··· 変更する色の `saturation` 値。

○ `brightness` (real) ··· 変更する色の `brightness` 値。

○ `usercolor` (integer,optional) ··· 変更後の線色を、ユーザー指定色として登録する。

3. 使用法

· `hue,saturation,brightness` は、それぞれ 0.0 から 1.0 の範囲で指定する。

· `usercolor` は、$-3 >$ `usercolor` $\geqq -32$ の範囲で指定する。

4. 使用例

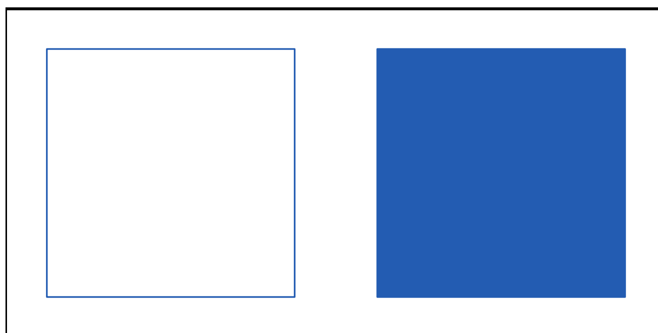


Fig.3-5 program pencolor2_example

目次へ

□ pencolor(3) (R、G、B 値による線色の指定)

1. 呼び出し形式

```
call pencolor(ired,igreen,iblue[,usercolor])
```

2. 引数の説明

○ `ired` (integer) · · · Red の強さを示す基本型整数。

○ `igreen` (integer) · · · Green の強さを示す基本型整数。

○ `iblue` (integer) · · · Blue の強さを示す基本型整数。

○ `usercolor` (integer,optional) · · · 変更後の線色を、ユーザー指定色として登録する。

3. 使用法

· 変更する色を、RGB 表現でそれぞれ 0 から 255 までの 256 段階で表し、それぞれ、`ired`、`igreen`、`iblue` とする。

· `usercolor` は、$-3 >$ `usercolor` $\geqq -32$ の範囲で指定する。

4. 使用例

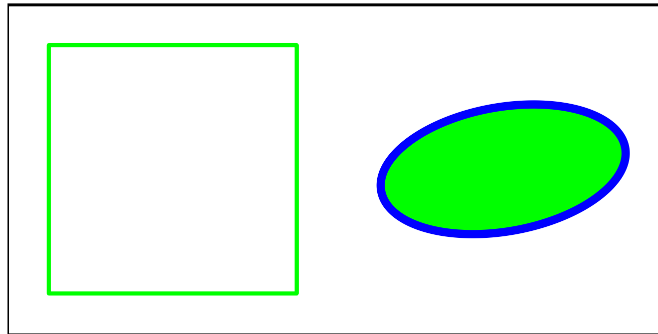


Fig.3-6 program pencolor3_example

目次へ

□ hatchset (斜線パターンの定義、登録)

1. 呼び出し形式

```
call hatchset(no[,penmode][,pencolor][,penwidth][,interval][,ang1][,ang2])
```

2. 引数の説明

○ no (integer) ··· 定義した斜線パターンを格納する番号 (1-60)。

○ penmode (integer,optional) ··· 斜線パターンを引くペンの penmode。

○ pencolor (integer,optional) ··· 斜線パターンを引くペンの pencolor。

○ penwidth (integer,optional) ··· 斜線パターンを引くペンの penwidth。

○ interval (real,optional) ··· 斜線パターン同士の間隔。

○ ang1 (real,optional) ··· 斜線パターン 1 の傾き、x 軸から左回りに計る。

○ ang2 (real,optional) ··· 斜線パターン 2 の傾き、x 軸から左回りに計る。

3. 使用法

· no 番目の斜線パターンを再定義する。指定しなかったパラメータは変更されない。

· 斜線は角度を変えて 2 回描かれる。ang2 を負の値にしておけば、2 回目は描かれない。

4. 使用例

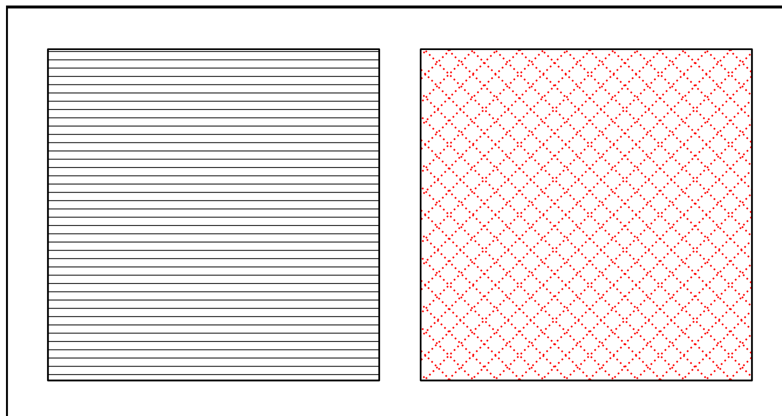


Fig.3-7 program hatchset_example

目次へ

□ dotset (点パターンの定義、登録)

1. 呼び出し形式

```
call hatchset(no[,pencolor][,size][,interval][,ang])
```

2. 引数の説明

○ `no` (integer) · · · 定義した点パターンを格納する番号 (1-20)。

○ `pencolor` (integer,optional) · · · 点パターンを引くペンの `pencolor`。

○ `size` (real,optional) · · · 点の半径。

○ `interval` (real,optional) · · · 点パターン同士の間隔。

○ `ang` (real,optional) · · · 点パターンの傾き、`x` 軸から左回りに計る。

3. 使用法

· `no` 番目の点パターンを再定義する。指定しなかったパラメータは変更されない。

4. 使用例

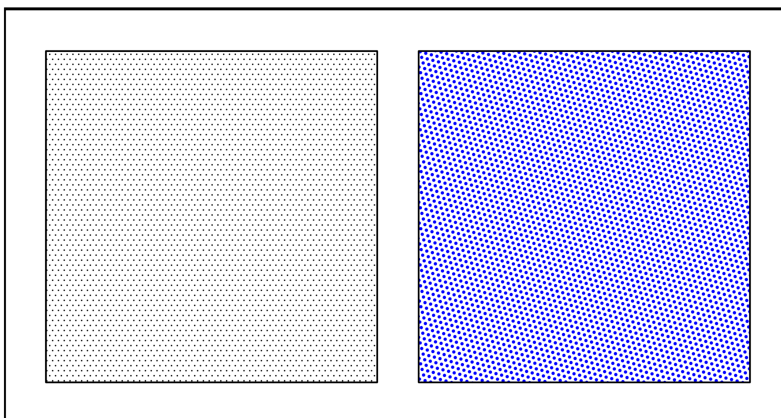


Fig.3-8 program dotset_example

目次へ

■ 4　文字、数字、記号を書く　■

□ symbol (文字を書く)

1. 呼び出し形式

```
call symbol(x,y,h,str[,ang][,n][,location][,boardcolor][,font][,code])
```

2. 引数の説明

○ `x` (real) ··· 文字作画開始点の x 座標。

○ `y` (real) ··· 文字作画開始点の y 座標。

○ `h` (real) ··· 文字の高さ。

○ `str` (character) ··· 作画する文字列。

○ `ang` (real,optional → 0.0) ··· 文字列の傾き、x 軸から左回りに計る。

○ `n` (integer,optional → 全文字列) ··· 描画する文字数

○ `location` (integer,optional) ··· 作画開始座標と文字列との位置関係

○ `boardcolor` (integer,optional → 背景板なし) ··· 背景板の色。

○ `font` (integer,optional) ··· フォントの番号。

○ `code` (characterr,optional) ··· 漢字コードを文字型で指定。

3. 使用法

· `x` または `y` が 999.0 以上の場合、(可能ならば) 直前に書いた文字に続けて書く。

· `str` には多バイト文字も書けるが、必ず `code` を指定しなければならない。

· `location`、`boardcolor`、`font`、`code` の詳細については、User's Manual を参照。

· `font`、`code` はサブルーチン限り有効で、保持されない。

4. 使用例

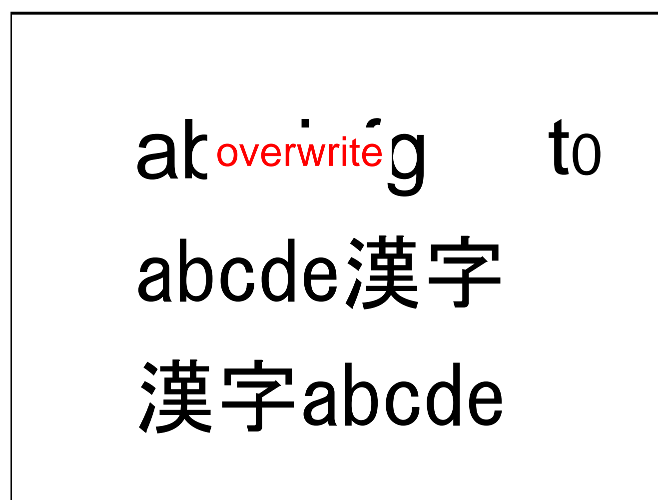


Fig.4-1 program symbol_example

目次へ

□ kanji (漢字を書く)

1. 呼び出し形式

```
call kanji(x,y,h,str[,ang],[n],[location][,boardcolor][,font][,code])
```

2. 引数の説明

○ x (real) · · · 文字作画開始点の x 座標。

○ y (real) · · · 文字作画開始点の y 座標。

○ h (real) · · · 文字の高さ。

○ str (character) · · · 作画する文字列。

○ ang (real,optional → 0.0) · · · 文字列の傾き、x 軸から左回りに計る。

○ n (integer,optional → 全文字列) · · · 描画する文字数

○ location (integer,optional) · · · 作画開始座標と文字列との位置関係

○ boardcolor (integer,optional → 背景板なし) · · · 背景板の色。

○ font (integer,optional) · · · フォントの番号。

○ code (characterr,optional) · · · 漢字コードを文字型で指定。

3. 使用法

· x または y が 999.0 以上の場合、(可能ならば) 直前に書いた文字に続けて書く。

· str には多バイト文字を指定する。code を指定しないときは、カレント・コードが適用される。

· 他の引数については、symbol と同じ。

4. 使用例

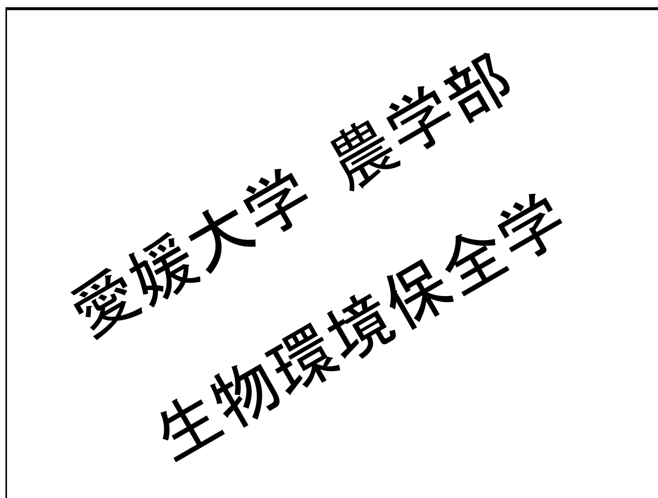


Fig.4-2 program kanji_example

目次へ

□ number (数値を与えて数字を書く)

1. 呼び出し形式

```
call number(x,y,h,val[,ang][,n][,location][,boardcolor][,font])
```

2. 引数の説明

○ `x` (real) · · · 数字作画開始点の x 座標。

○ `y` (real) · · · 数字作画開始点の y 座標。

○ `h` (real) · · · 数字の高さ。

○ `str` (real — integer) · · · 作画する数。

○ `ang` (real,optional → 0.0) · · · 数字列の傾き、x 軸から左回りに計る。

○ `n` (integer,optional → 0 出ない最後のケタ) · · · 小数点以下のケタ数

○ `location` (integer,optional) · · · 作画開始座標と文字列との位置関係

○ `boardcolor` (integer,optional → 背景板なし) · · · 背景板の色。

○ `font` (integer,optional) · · · フォントの番号。

3. 使用法

· `x` または `y` が 999.0 以上の場合、(可能ならば) 直前に書いた文字に続けて書く。

· `str` には基本実数型、あるいは基本整数型を指定する。

· `n>0` のとき、小数点以下 `n+1` ケタを丸めて、`n` ケタまで書く。`n` ＝ 0 のとき、整数部と「`.`」だけを書く。
`n<0` のとき、整数部が`-n` ケタに足りない場合、上のケタに 0 を`-n` ケタまで補う。

· 他の引数については、`symbol` と同じ。ただし `code` は指定できない。

4. 使用例

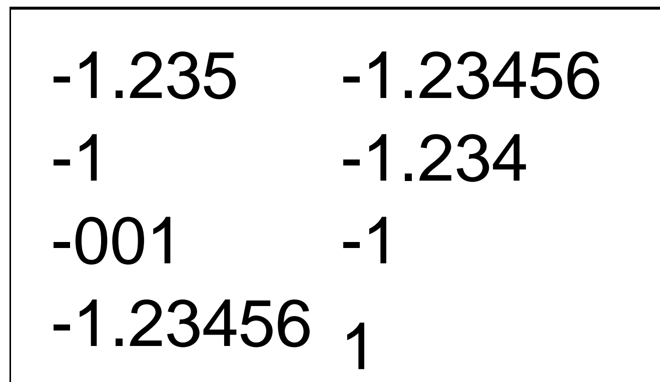


Fig.4-3 program number_example

目次へ

□ pfont (フォントの変更)

1. 呼び出し形式

```
call pfont_num(ifont)
```

2. 引数の説明

○ `ifont` (integer) ··· 変更するフォントの番号。

3. 使用法

· 欧文フォント (`ifont>0`) と、漢字フォント (`ifont<0`) は独立に指定され、保持される。

4. 使用例

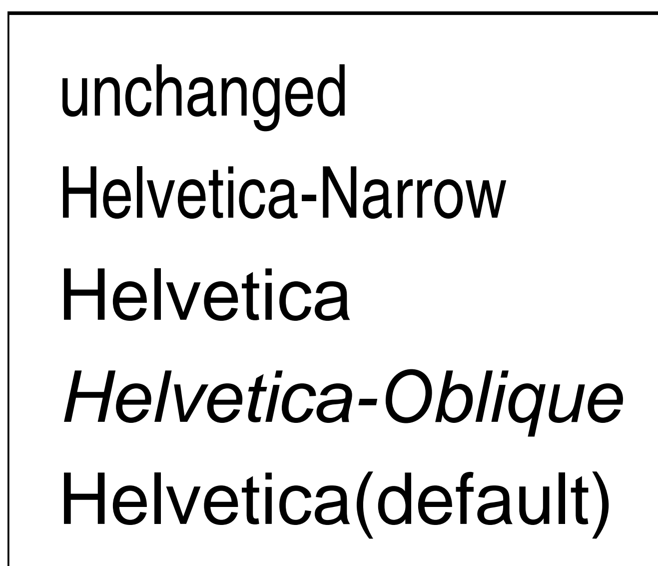


Fig.4-4 program pfont_example

目次へ

□ center (センター・シンボルを書く)

1. 呼び出し形式

```
call  center(x,y,h,n)
```

2. 引数の説明

○ x (real) ··· 記号中心の x 座標。

○ y (real) ··· 記号中心の y 座標。

○ h (real) ··· 記号の高さ。

○ n (integer) ··· 記号の種類 (下図参照)。

3. 使用法

· 下の例図にある 20 種の記号を、座標 (x,y) に高さ h で書く。

· どの記号で書くかは、引数 n で指定する。

· 1,4,7,··· は透明、2,5,8,··· は背景色で塗りつぶした記号である。

4. 使用例

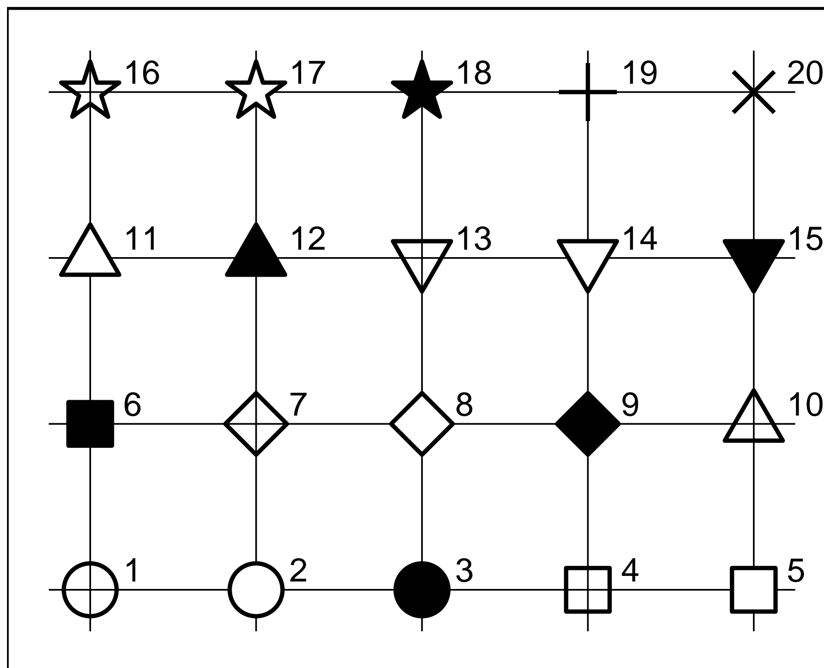


Fig.4-5 program center_example

目次へ

■ 5 図形の描画 (1) 円、楕円 ■

□ pcircle (円・円弧の描画)

1. 呼び出し形式

```
call  pcircle(x,y,r[,ang1][,ang2][,asp][,ang][,fillcolor][,hatch])
```

2. 引数の説明

○ x (real) ··· 円の中心の x 座標。

○ y (real) ··· 円の中心の y 座標。

○ h (real) ··· 円の半径。

○ ang1 (real,optional) ··· 円弧の書き始めの角度 (x 軸方向から左回りに度単位)。

○ ang2 (real,optional) ··· 円弧の書き終わりの角度 (x 軸方向から左回りに度単位)。

○ asp (real,optional) ··· x 方向に対する y 方向の扁平率。

○ ang (real,optional → 0.0) ··· x 方向の軸の傾き、x 軸から左回りに計る。

○ fillcolor (integer,optional) ··· 塗りつぶし色を、基本整数型で指定。pencolor 参照。

○ hatch (integer,optional) ··· 斜線パターン番号。Use's Manual 参照。

3. 使用法

· ang1、ang2 のどちらかが指定されなかったとき、全部の円を描く。

· ang1、ang2 が指定されたとき、ang1 から ang2 まで左回りに円弧を書く。

· fillcolor または hatch が指定されたとき、円弧を書くと両端が中心と結ばれ、扇形になる。

· r<0 のときも円弧は扇形になる。この場合の半径は|r|になる。

4. 使用例

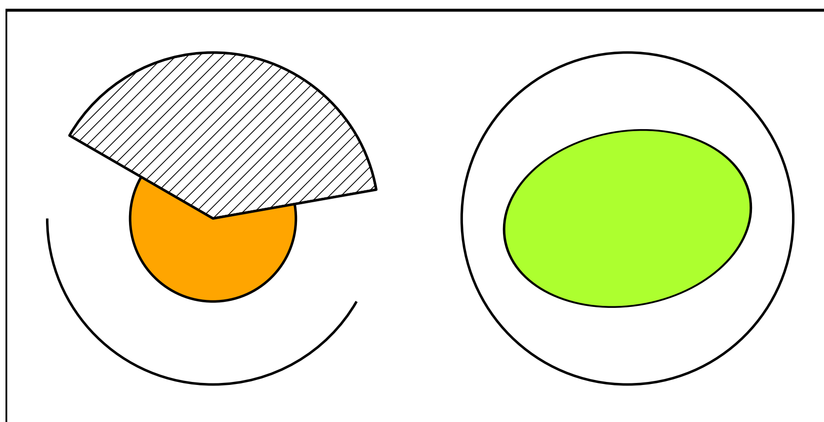


Fig.5-1 program pcircle_example

目次へ

□ pellipse (楕円・楕円弧の描画)

1. 呼び出し形式

```
call pellipse(x,y,r[,ang1][,ang2][,asp][,ang][,fillcolor][,hatch][,idiv])
```

2. 引数の説明

○ `x` (real) · · · 楕円の中心の x 座標。

○ `y` (real) · · · 楕円の中心の y 座標。

○ `h` (real) · · · 楕円の x 軸方向の半径。

○ `ang1` (real,optional) · · · 楕円弧の書き始めの角度 (x 軸方向から左回りに度単位)。

○ `ang2` (real,optional) · · · 楕円弧の書き終わりの角度 (x 軸方向から左回りに度単位)。

○ `asp` (real,optional) · · · x 方向に対する y 方向の扁平率。

○ `ang` (real,optional → 0.0) · · · x 方向の軸の傾き、x 軸から左回りに計る。

○ `fillcolor` (integer,optional) · · · 塗りつぶし色を、基本整数型で指定。`pencolor` 参照。

○ `hatch` (integer,optional) · · · 斜線パターン番号。Use's Manual 参照。

○ `idiv` (integer,optional → 360) · · · 楕円を `idiv` 角形で近似する。

3. 使用法

· `ang1`、`ang2` のどちらかが指定されなかったとき、全部の楕円を描く。

· `ang1`、`ang2` が指定されたとき、`ang1` から `ang2` まで左回りに楕円弧を書く。

· `fillcolor` または `hatch` が指定されたとき、楕円弧を書くと両端が中心と結ばれ、扇形になる。

· `r<0` のときも楕円弧は扇形になる。この場合の x 軸方向の半径は`|r|`になる。

· `pcircle` でも楕円は描ける (青本) が、その場合の楕円は円を斜めから見た形式になる。[*1] この場合、線の太さが変化するし、斜線の角度や間隔も変わってくる (下図左)。これを避けるために、`pellipse` では楕円を `idiv` 角形で近似して描く。楕円の一部だけを拡大して描く場合、`idiv` を大きくする必要があるが、あまり大きくしない方がかえってきれいに描けるようである。

4. 使用例

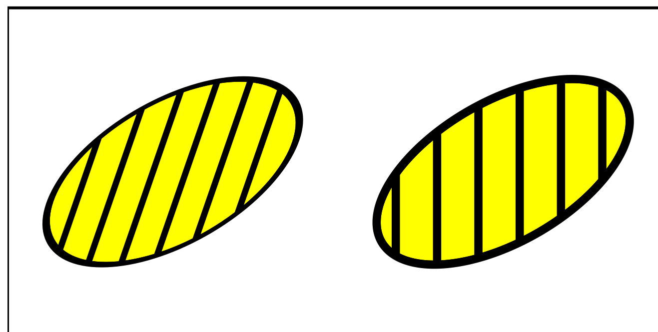


Fig.5-2 program pellipse_example

目次へ

[*1] これは postscript がデザインを主な目的にしていることに起因する。自然界にせよ人工物にせよ本来楕円形をしているものは少なく、楕円形に見えるものは円を斜めから見た場合がほとんどだからである。

□ tcircle (三点を通る円弧・円の描画)

1. 呼び出し形式

```
call  tcircle(x1,y1,x2,y2,x3,y3[,fillcolor][,hatch][,full])
```

2. 引数の説明

○ `x1` (real) · · · 始点の x 座標。

○ `y1` (real) · · · 始点の y 座標。

○ `x2` (real) · · · 中間点の x 座標。

○ `y2` (real) · · · 中間点の y 座標。

○ `x3` (real) · · · 終点の x 座標。

○ `y3` (real) · · · 終点の y 座標。

○ `fillcolor` (integer,optional) · · · 塗りつぶし色を、基本整数型で指定。`pencolor` 参照。

○ `hatch` (integer,optional) · · · 斜線パターン番号。Use's Manual 参照。

○ `full` (logical,optional → `.false.`) · · · `.true.` ならば、円を描く。

3. 使用法

· `(x1,y1)` を始点とし、`(x2,y2)` を通り、`(x3,y3)` を終点とする円弧を書く。

· `fillcolor` または `hatch` が指定されたとき、円弧を書くと両端が中心と結ばれ、扇形になる。

4. 使用例

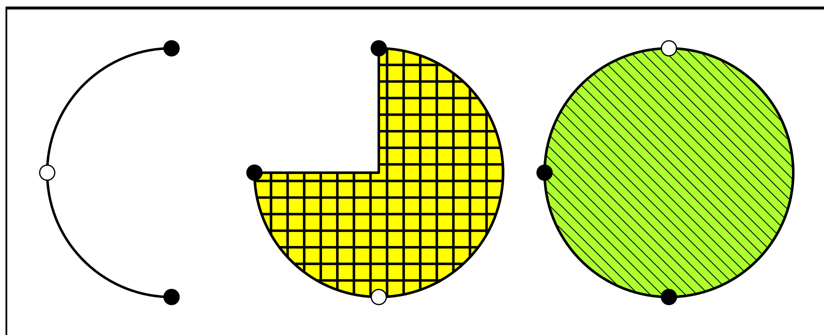


Fig.5-3 program tcircle_example

目次へ

■ 6 図形の描画 (2) 多角形 ■

□ plinec (多角形の x、y 座標による描画)

1. 呼び出し形式

(a) `call plinec(x_array,y_array[,n][,penmode][,pencolor][,penwidth][,fillcolor][,hatch])`

または、

(b) `call plinec(xy_array[,n][,penmode][,pencolor][,penwidth][,fillcolor][,hatch])`

2. 引数の説明

○ `x_array` (real,array) · · · x 座標の 1 次元配列。

○ `y_array` (real,array) · · · y 座標の 1 次元配列。

○ `xy_array` (structure,array) · · · 構造体 `coord_xy` の 1 次元配列。

○ `n` (integer,optional → [min(size(x_array),size(y_array))]) · · · 使用する座標の組の個数。

○ `penmode` (integer,optional → unchanged) · · · 線種の指定 (User's manual 参照)。

○ `pencolor` (integer,optional→ unchanged) · · · 線の色 (User's manual 参照)。

○ `penwidth` (integer,optional → unchanged) · · · 線の幅の指定 (User's manual 参照)。

○ `fillcolor` (integer,optional) · · · 塗りつぶし色 (User's manual 参照)。

○ `hatch` (integer,optional) · · · テクスチャ・パターン番号 (同上)。

3. 使用法

(a) · 座標 (`x_array(1)`,`y_array(1)`)、(`x_array(2)`,`y_array(2)`)、(`x_array(3)`,`y_array(3)`) · · · (`x_array(n)`,`y_array(n)`) を頂点とする多角形を描く。

· `n` が指定されていない場合、配列 `x_array`、`y_array` の次元の小さい方が `n` となる。

(b) · 座標 `xy_array(1)`、`xy_array(2)`、`xy_array(3)` · · · `xy_array(n)` を頂点とする多角形を描く。

· 構造体 `coord_xy` は以下のように、座標 (x,y) を要素に持つ構造体である。

```
type, public :: coord_xy
  real        :: x, y
end type coord_xy
```

· `n` が指定されていない場合、配列 `xy_array` の次元が `n` となる。

(共通) · その間端点処理や破線パターンの継続が行われる。

4. 使用例

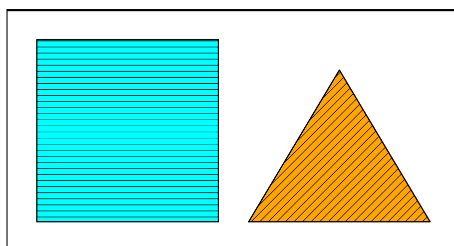


Fig.6-1 program plinec_example

目次へ

□ prect (長方形の描画)

1. 呼び出し形式

```
call prect(xo,yo,width,height[,ang][,fillcolor][,hatch])
```

2. 引数の説明

○ `xo` (real) ··· 長方形の左下隅の x 座標。

○ `yo` (real) ··· 長方形の左下隅の y 座標。

○ `width` (real) ··· 長方形の幅。

○ `height` (real) ··· 長方形の高さ。

○ `ang` (real,optional → 0.0) ··· 長方形の下辺の x 方向の軸の傾き、x 軸から左回りに計る。

○ `fillcolor` (integer,optional) ··· 塗りつぶし色を、基本整数型で指定。`pencolor` 参照。

○ `hatch` (integer,optional) ··· 斜線パターン番号。Use's Manual 参照。

3. 使用法

· Calcomp でよく使われるサブルーチンとは、縦横の順番が異なっている。

4. 使用例

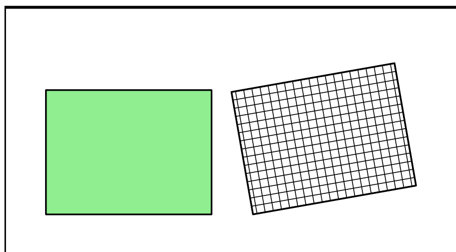


Fig.6-2 program prect_example

目次へ

□ ptriangle ((二等辺三角形の描画)

1. 呼び出し形式

```
call ptriangle(xo,yo,width,height[,ang][,fillcolor][,hatch])
```

2. 引数の説明

○ `xo` (real) · · · 二等辺三角形の左下隅の x 座標。

○ `yo` (real) · · · 二等辺三角形の左下隅の y 座標。

○ `width` (real) · · · 二等辺三角形の下辺の長さ。

○ `height` (real) · · · 二等辺三角形の高さ。

○ `ang` (real,optional → 0.0) · · · 二等辺三角形の底辺の x 方向の軸の傾き、x 軸から左回りに計る。

○ `fillcolor` (integer,optional) · · · 塗りつぶし色を、基本整数型で指定。`pencolor` 参照。

○ `hatch` (integer,optional) · · · 斜線パターン番号。Use's Manual 参照。

3. 使用法

· △型の二等辺三角形を描く。`height` を負にすると、▽型の二等辺三角形になる。

4. 使用例

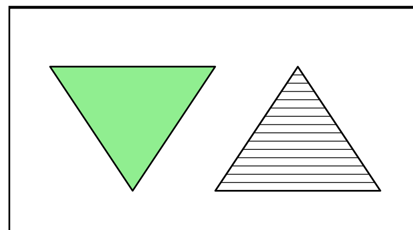


Fig.6-3 program ptriangle_example

目次へ

□ polygon (多角形の相対座標による描画)

1. 呼び出し形式

`call polygon(xo,yo,x_array,y_array[,n][,ang][,location][,fillcolor][,hatch])` ··· (a)

または、

`call polygon(xo,yo,xy_array[,n][,ang][,location][,fillcolor][,hatch])` ··· (b)

2. 引数の説明

○ `x_array` (real,array) ··· x 座標の 1 次元配列。

○ `y_array` (real,array) ··· y 座標の 1 次元配列。

○ `xy_array` (structure,array) ··· 構造体 `coord_xy` の 1 次元配列。

○ `n` (integer,optional → [min(size(x_array),size(y_array))]) ··· 使用する座標の組の個数。

○ `ang` (real,optional → 0.0) ··· 図形の `x` 方向の軸の傾き、`x` 軸から左回りに計る。

○ `location` (integer,optional) ··· `(xo,yo)` と図形との位置関係 (User's manual 参照)。

○ `fillcolor` (integer,optional) ··· 塗りつぶし色 (User's manual 参照)。

○ `hatch` (integer,optional) ··· テクスチャ・パターン番号 (同上)。

3. 使用法

(a) · 座標 `(x_array(1),y_array(1))`、`(x_array(2),y_array(2))`、`(x_array(3),y_array(3))` ··· `(x_array(n),y_array(n))` を形状とする多角形を描く。描く位置は `(xo,yo)` と `location` で決まる。

· `n` が指定されていない場合、配列 `x_array`、`y_array` の次元の小さい方が `n` となる。

(b) · 座標 `xy_array(1)`、`xy_array(2)`、`xy_array(3)` ··· `xy_array(n)` を形状とする多角形を描く。描く位置は `(xo,yo)` と `location` で決まる。

· 構造体 `coord_xy` は以下のように、座標 (x,y) を要素に持つ構造体である。

```
type, public :: coord_xy
  real       :: x, y
end type coord_xy
```

· `n` が指定されていない場合、配列 `xy_array` の次元が `n` となる。

(共通) `location` を負にすると、図形は左右反転して描かれる。

4. 使用例

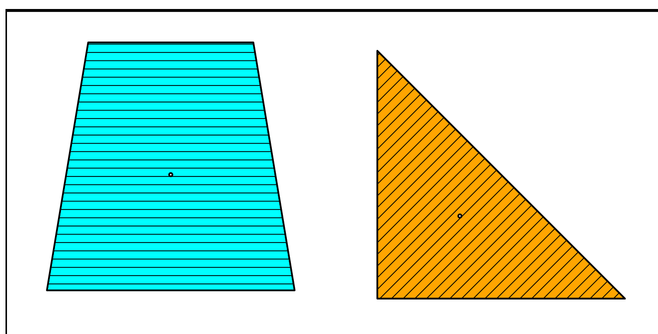


Fig.6-4 program polygon_example

目次へ

■ 7　図形の描画 (3) 矢印　■

□ arrow (始点と方向を指定した矢印の描画)

1. 呼び出し形式

```
call arrow(xo,yo,alen[,ang][,hlen][,breadth][,isort][,fillcolor])
```

2. 引数の説明

○ `xo` (real) ··· 矢印の始点の x 座標。

○ `yo` (real) ··· 矢印の始点の y 座標。

○ `alen` (real) ··· 矢印の長さ。

○ `ang` (real,optional → `0.0`) ··· 矢印の方向、`x` 軸から左回りに計る。

○ `hlen` (real,optional → `breadth` があれば、`breadth`/2、なければ、`alen`/10) ··· 鏃（やじり）の長さ。

○ `breadth` (real,optional → `hlen`/2) ··· 鏃の幅。

○ `isort` (integer,optional → 1) ··· 鏃の形状と、始点に対する矢印の位置を表すパラメータ (使用例参照)。

○ `fillcolor` (integer,optional) ··· 鏃の色 (User's manual 参照)。

3. 使用法

· `isort` は全部で 12 種類あり、一の位が矢印の形状、十の位が矢印の位置を表す (黒点が `(xo,yo)`)。

· `isort=-1,-2,-3` とすると、それぞれ `isort=21,22,23` と同じ意味になる。

4. 使用例

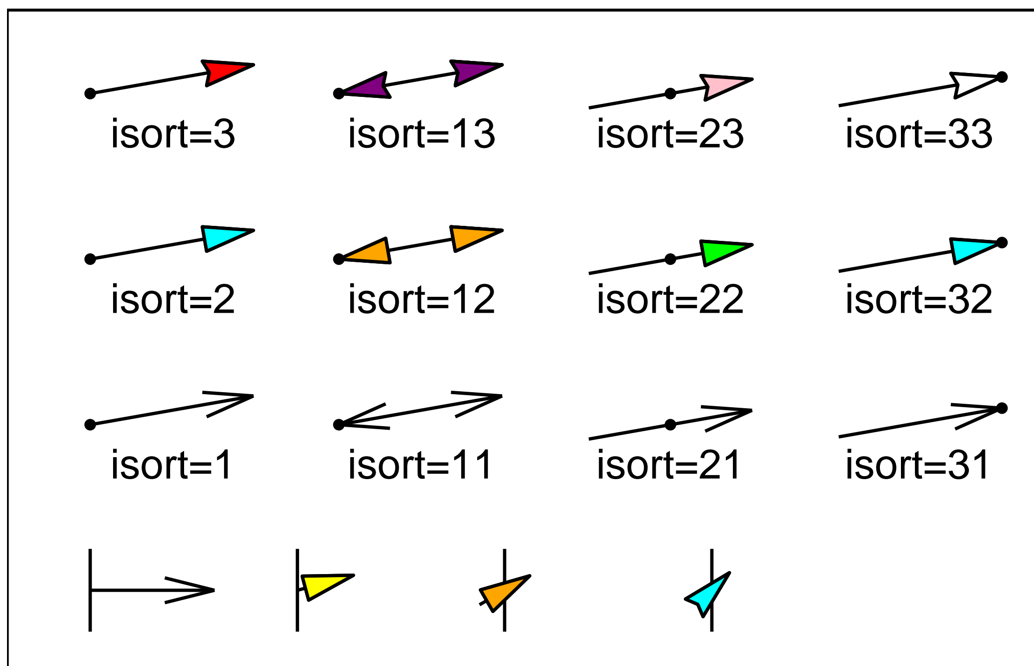


Fig.7-1 program arrow_example

目次へ

□ pvector (始点と終点を指定した矢印の描画)

1. 呼び出し形式

```
call pvector(x1,y1,x2,y2[,hlen][,breadth][,isort][,fillcolor])
```

2. 引数の説明

○ x1 (real) · · · 矢印の始点の x 座標。

○ y1 (real) · · · 矢印の始点の y 座標。

○ x2 (real) · · · 矢印の終点の x 座標。

○ y2 (real) · · · 矢印の終点の y 座標。

○ hlen (real,optional → breadth があれば、breadth/2、なければ、alen/10) · · · 鏃（やじり) の長さ。

○ breadth (real,optional → hlen/2) · · · 鏃の幅。

○ isort (integer,optional → 1) · · · 鏃の形状と、始点に対する矢印の位置を表すパラメータ (使用例参照)。

○ fillcolor (integer,optional) · · · 鏃の色 (User's manual 参照)。

3. 使用法

· isort は全部で 9 種類あり、一の位が矢印の形状、十の位が矢印の位置を表す (● が (x1,y1)、○ が (x2,y2))。

· isort=-1,-2,-3 とすると、それぞれ isort=11,12,13 と同じ意味になる (arrow との相違に注意)。

· (x1,y1)=(x2,y2) の場合、warning が出力され、長さのない右向きの矢印が出力される (図の右下隅)。

4. 使用例

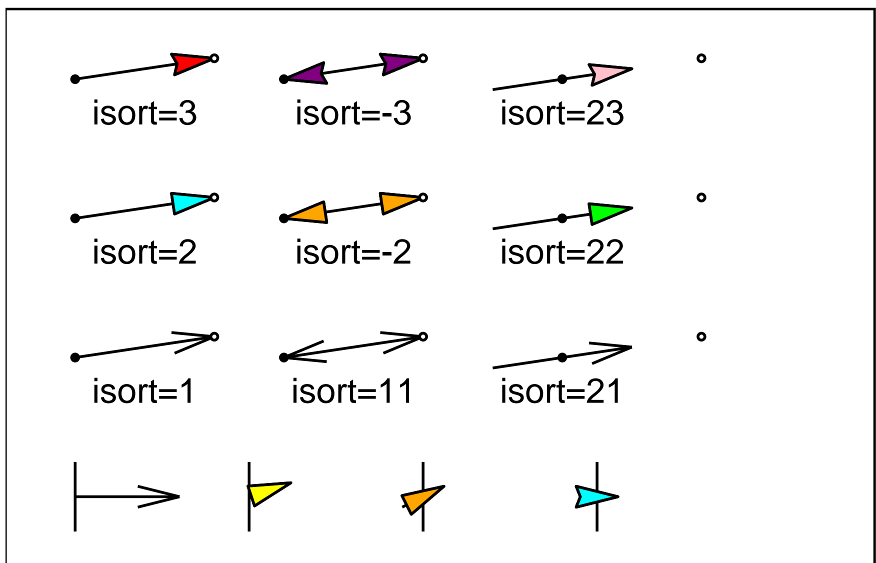


Fig.7-2 program pvector_example

目次へ

■ 8 二次元グラフと座標 (1) 標準 ■

□ paxis (任意方向の軸の設定-通常)

1. 呼び出し形式

```
call paxis (xo,yo,axlen,v_min,v_max,v_int &
     & [,title][,ang][,idir][,intsub][,cs][,bl][,bl2][,code])
```

2. 引数の説明

○ `xo` (real) · · · 軸の始点の x 座標。

○ `yo` (real) · · · 軸の始点の y 座標。

○ `axlen` (real) · · · 軸の長さ。

○ `v_min` (real) · · · 軸の始点での対象となる量の値。

○ `v_max` (real) · · · 軸の終点での対象となる量の値。

○ `v_int` (real) · · · 軸の数値を書く間隔となる量の値。

○ `title` (character,optional) · · · 軸の名称。

○ `ang` (real,optional → 0.0) · · · 始点から軸を引く角度、右方向から左回りに計る。

○ `idir` (integer,optional → 1) · · · 軸に対して、数値や目盛線を記入する位置 (使用例参照)。

○ `intsub` (integer,optional → 2) · · · 数値と数値の間を、補助目盛線で分割するときの分割数。

○ `cs` (real,optional → 0.3) · · · 目盛線の文字の高さ。

○ `bl` (real,optional → 0.2) · · · 目盛線の長さ。

○ `bl2` (real,optional → bl*2.0/3.0) · · · 補助目盛線の長さ。

○ `code` (character,optional → ’ascii’) · · · title の文字コード。

3. 使用法

· 任意の方向の軸を引くときに使う。x-y 平面グラフを作成するときは、pxaxis、pyaxis の方を使う。

4. 使用例

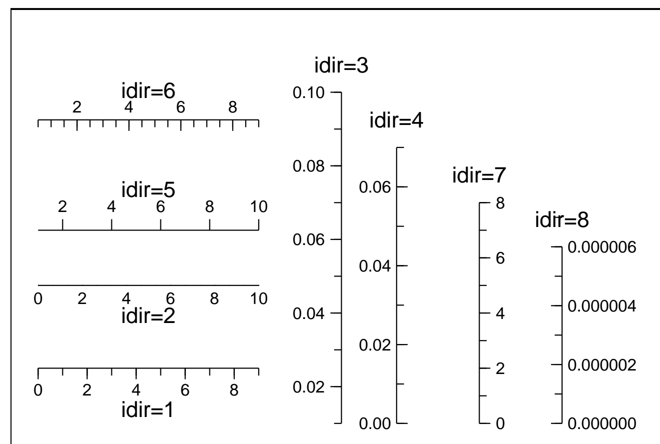


Fig.8-1 program paxis_example

目次へ

□ pxaxis (x 軸の設定-通常)

1. 呼び出し形式

```
call pxaxis (xo,yo,axlen,v_min,v_max,v_int &
         & [,title][,idir][,intsub][,cs][,bl][,bl2][,code])
```

2. 引数の説明

○ `xo` (real) ··· x 軸の始点の x 座標。

○ `yo` (real) ··· x 軸の始点の y 座標。

○ `axlen` (real) ··· x 軸の長さ。

○ `v_min` (real) ··· x 軸の始点での対象となる量の値。

○ `v_max` (real) ··· x 軸の終点での対象となる量の値。

○ `v_int` (real) ··· x 軸の数値を書く間隔となる量の値。

○ `title` (character,optional) ··· x 軸の名称。

○ `idir` (integer,optional → 1) ··· x 軸に対して、数値や目盛線を記入する位置 (使用例参照)。

○ `intsub` (integer,optional → 2) ··· 数値と数値の間を、補助目盛線で分割するときの分割数。

○ `cs` (real,optional → 0.3) ··· 目盛線の文字の高さ。

○ `bl` (real,optional → 0.2) ··· 目盛線の長さ。

○ `bl2` (real,optional → bl*2.0/3.0) ··· 補助目盛線の長さ。

○ `code` (character,optional → 'ascii') ··· title の文字コード。

3. 使用法

· x 軸を引くときに使う。これにより x 軸のパラメータがセットされるので、`xyline` を呼ぶ前に `pxaxis`(`lxaxis`)、および `pyaxis`(`lyaxis`) を呼んでおかなければならない。

4. 使用例

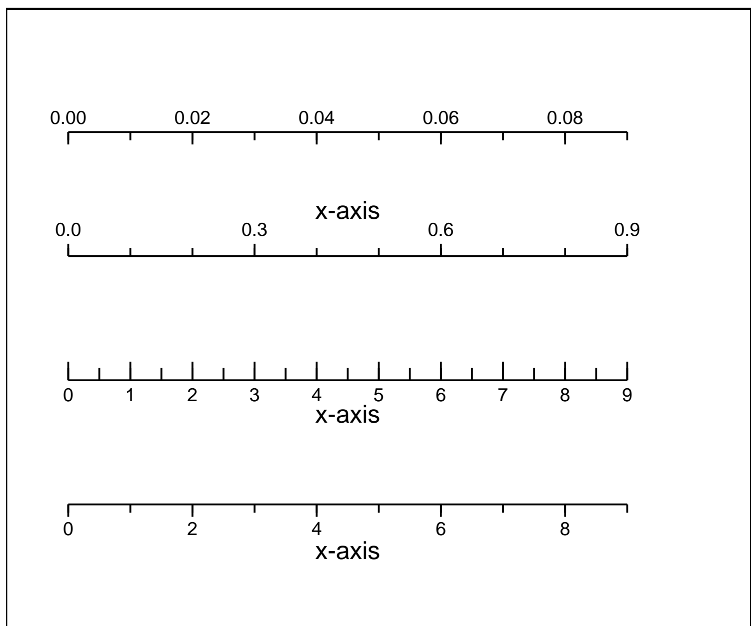


Fig.8-2 program pxaxis_example

目次へ

□ pyaxis (*y* 軸の設定-通常)

1. 呼び出し形式

```
call pyaxis (xo,yo,aylen,v_min,v_max,v_int &
     & [,title][,idir][,intsub][,cs][,bl][,bl2][,code])
```

2. 引数の説明

○ `xo` (real) · · · *y* 軸の始点の x 座標。

○ `yo` (real) · · · *y* 軸の始点の y 座標。

○ `aylen` (real) · · · *y* 軸の長さ。

○ `v_min` (real) · · · *y* 軸の始点での対象となる量の値。

○ `v_max` (real) · · · *y* 軸の終点での対象となる量の値。

○ `v_int` (real) · · · *y* 軸の数値を書く間隔となる量の値。

○ `title` (character,optional) · · · *y* 軸の名称。

○ `idir` (integer,optional → 3) · · · *y* 軸に対して、数値や目盛線を記入する位置 (使用例参照)。

○ `intsub`、`cs`、`bl`、`bl2`、`code` · · · `pxaxis` と同じ。

3. 使用法

· *y* 軸を引くときに使う。これにより *y* 軸のパラメータがセットされるので、`xyline` を呼ぶ前に `pxaxis(lxaxis)`、および `pyaxis(lyaxis)` を呼んでおかなければならない。

4. 使用例

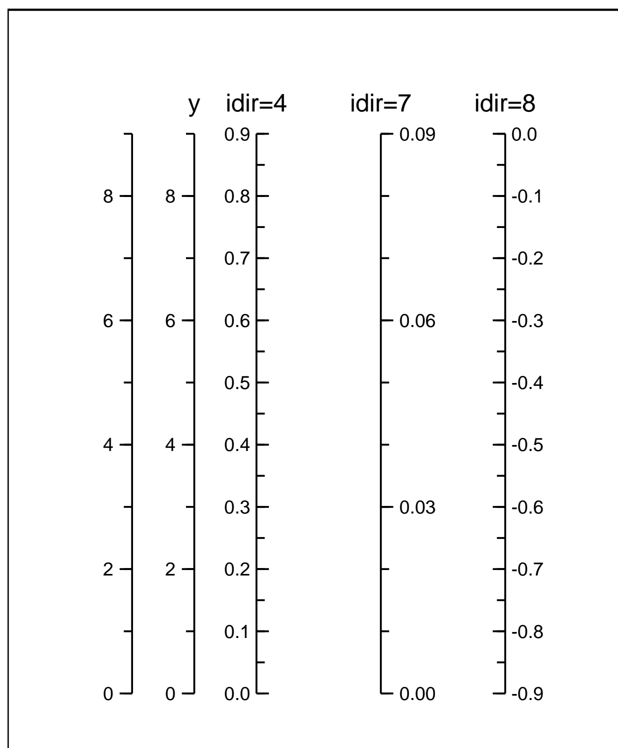


Fig.8-3 program pyaxis_example

目次へ

□ xygrid (x 軸、y 軸に平行な格子線を描く)

1. 呼び出し形式

```
call xygrid(x_int,y_int[,x_start][,y_start][,penmode][,pencolor][,penwidth])
```

x 軸に垂直な格子のみ描くとき、

```
call pxgrid(x_int[,x_start][,penmode][,pencolor][,penwidth])
```

y 軸に垂直な格子のみ描くとき、

```
call pygrid(y_int[,y_start][,penmode][,pencolor][,penwidth])
```

2. 引数の説明

○ `x_int` (real) · · · x 軸に垂直な格子の間隔 (軸の数値の単位)。

○ `y_int` (real) · · · y 軸に垂直な格子の間隔 (軸の数値の単位)。

○ `x_start` (real,optional → (図の左端) · · · x 軸に垂直な格子を書き始める最小値 (軸の数値の単位)。

○ `y_start` (real,optional → (図の下端) · · · y 軸に垂直な格子を書き始める最小値 (軸の数値の単位)。

○ `penmode` (integer,optional) · · · 格子線を描くペンの `penmode`。

○ `pencolor` (integer,optional) · · · 格子線を描くペンの `pencolor`。

○ `penwidth` (integer,optional) · · · 格子線を描くペンの `penwidth`。

3. 使用法

· pxaxis(lxaxis) と pyaxis(lyaxis) を呼んで軸を書いた後、xyline を呼ぶ前に、xygrid で格子線を描く。

· 縦、横どちらかの格子線のみ描きたいときには、pxgrid または pygrid を call する。

· xyline を呼んだ後に xygrid を呼ぶと、格子線によって上書きされてしまう。

4. 使用例

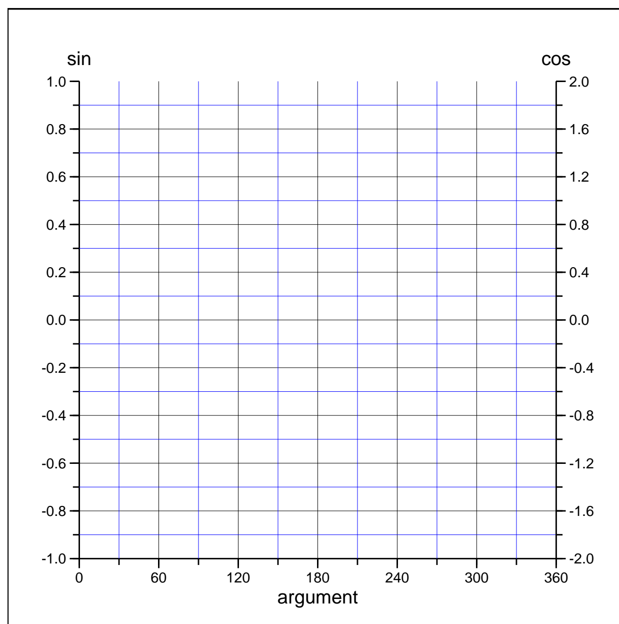


Fig.8-4 program xygrid_example

目次へ

□ xyline (x 軸、y 軸による二次元グラフ)

1. 呼び出し形式

```
call xyline(x_array,y_array[,n][,mask][,mode][,cs][,nin][,clip][,comment][,code])
```

2. 引数の説明

○ `x_array` (real,array) ··· データの x 値の配列。

○ `y_array` (real,array) ··· データの y 値の配列。

○ `n` (integer,optional → min(size(x_array),size(y_array))) ··· 記入するデータ数。

○ `mask` (logical,array,optional → .true.) ··· `.false.` のデータは描画しない。

○ `mode` (integer,optional → 0) ··· 0 <`mode`≦ 20 のとき、`mode` 番目のセンターシンボルを標本の (x,y) 座標に記入し、その間をカレントの線種で結ぶ。−20 <`mode`≦ 0 のとき、|`mode`| 番目のセンターシンボルを標本の (x,y) 座標に記入するが、線は引かない。`mode=0` または省略時 → 線のみを引く。

○ `cs` (real,optional → 0.2) ··· 記号の高さ。

○ `nin` (integer,optional → 1) ··· 記号を `nin` ごとに書くが、線は 1 つずつ結ぶ。`nin<0` ⇒ 記号を `nin` ごとに書き、線はその記号同士を結ぶ。

○ `clip` (charcter,optional → 'no') ··· `clip='yes'` ならば、x 軸と y 軸で囲まれた長方形以外をカット。

○ `comment` (character,optional) ··· reserved

○ `code` (character,optional → 'ascii') reserved

3. 使用法

· pxaxis(lxaxis) と pyaxis(lyaxis) を呼んで軸を書いた後、xyline で二次元グラフを書く。

· 軸が常用対数の場合も xyplot で描ける。

· pxaxis,pyaxis をもう一度呼ぶ前ならば、データを変えて同じ軸で別のグラフが書ける。

4. 使用例

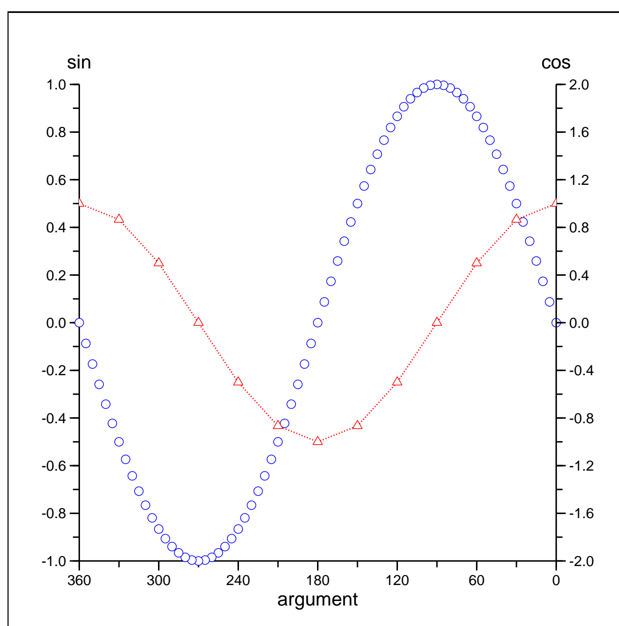


Fig.8-5 program xyline_example

目次へ

■ 9 二次元グラフと座標 (2) 常用対数軸 ■

□ laxis (任意方向の軸の設定-常用対数)

1. 呼び出し形式

```
call laxis (xo,yo,axlen,l_min,l_max[,title][,ang][,idir][,cs][,bl][,bl2][,code])
```

2. 引数の説明

○ `xo` (real) ··· 軸の始点の x 座標。

○ `yo` (real) ··· 軸の始点の y 座標。

○ `axlen` (real) ··· 軸の長さ。

○ `l_min` (integer) ··· 軸の始点での 10 のベキ乗のベキの数。

○ `l_max` (integer) ··· 軸の終点での 10 のベキ乗のベキの数

○ `title` (character,optional) ··· 軸の名称。

○ `ang` (real,optional → 0.0) ··· 始点から軸を引く角度、右方向から左回りに計る。

○ `idir` (integer,optional → 1) ··· 軸に対して、数値や目盛線を記入する位置 (使用例参照)。

○ `cs` (real,optional → 0.3) ··· 目盛線の文字の高さ。

○ `bl` (real,optional → 0.2) ··· 目盛線の長さ。

○ `bl2` (real,optional → bl*2.0/3.0) ··· 補助目盛線の長さ。

○ `code` (character,optional → 'ascii') ··· title の文字コード。

3. 使用法

· 任意の方向の軸を引くときに使う。x-y 平面グラフを作成するときは、lxaxis、lyaxis の方を使う。

4. 使用例

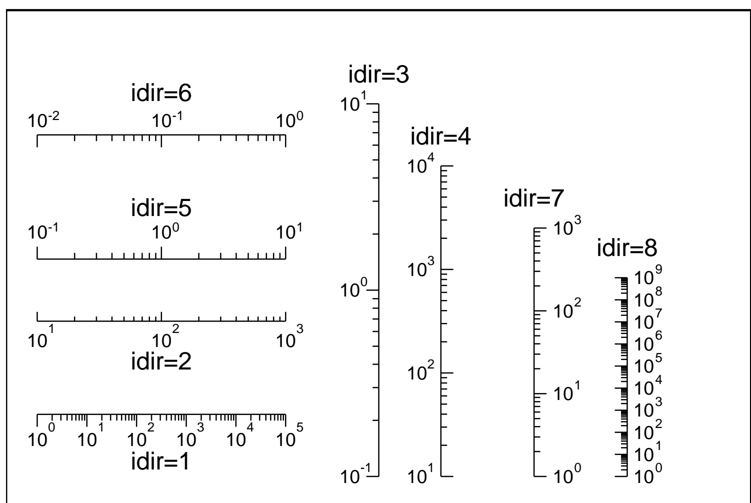


Fig.9-1 program laxis_example

目次へ

□ lxaxis (x 軸の設定-常用対数)

1. 呼び出し形式

```
call lxaxis (xo,yo,axlen,l_min,l_max[,title][,ang][,idir][,cs][,bl][,bl2][,code])
```

2. 引数の説明

○ `xo` (real) ··· 軸の始点の x 座標。

○ `yo` (real) ··· 軸の始点の y 座標。

○ `axlen` (real) ··· 軸の長さ。

○ `l_min` (integer) ··· 軸の始点での 10 のベキ乗のベキの数。

○ `l_max` (integer) ··· 軸の終点での 10 のベキ乗のベキの数

○ `title` (character,optional) ··· 軸の名称。

○ `ang` (real,optional → 0.0) ··· 始点から軸を引く角度、右方向から左回りに計る。

○ `idir` (integer,optional → 1) ··· 軸に対して、数値や目盛線を記入する位置 (使用例参照)。

○ `cs` (real,optional → 0.3) ··· 目盛線の文字の高さ。

○ `bl` (real,optional → 0.2) ··· 目盛線の長さ。

○ `bl2` (real,optional → bl*2.0/3.0) ··· 補助目盛線の長さ。

○ `code` (character,optional → 'ascii') ··· title の文字コード。

3. 使用法

· x 軸を引くときに使う。これにより x 軸のパラメータがセットされるので、`xyline` を呼ぶ前に `lxaxis`(`lxaxis`)、および `pyaxis`(`lyaxis`) を呼んでおかなければならない。

4. 使用例

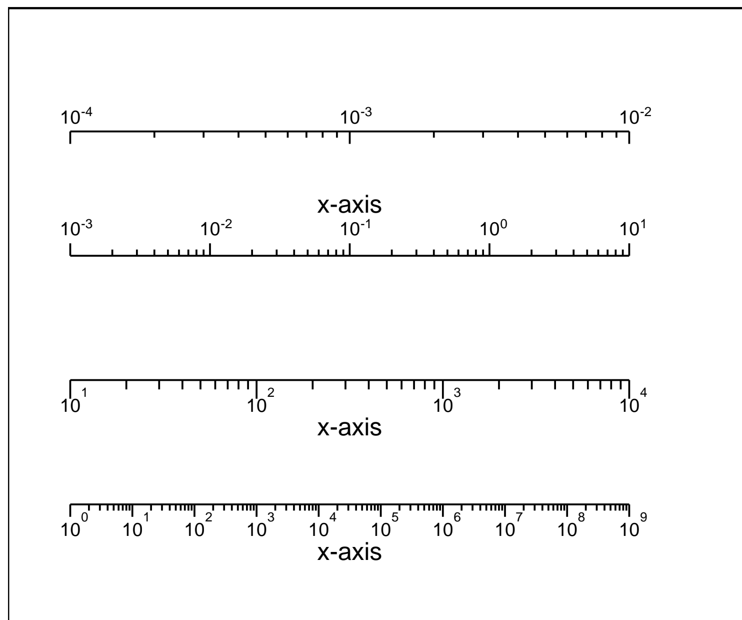


Fig.9-2 program lxaxis_example

目次へ

□ lyaxis (y 軸の設定-常用対数)

1. 呼び出し形式

```
call lyaxis (xo,yo,aylen,l_min,l_max[,title][,ang][,idir][,cs][,bl][,bl2][,code])
```

2. 引数の説明

○ `xo` (real) · · · 軸の始点の x 座標。

○ `yo` (real) · · · 軸の始点の y 座標。

○ `aylen` (real) · · · 軸の長さ。

○ `l_min` (integer) · · · 軸の始点での 10 のベキ乗のベキの数。

○ `l_max` (integer) · · · 軸の終点での 10 のベキ乗のベキの数

○ `title` (character,optional) · · · 軸の名称。

○ `ang` (real,optional → 0.0) · · · 始点から軸を引く角度、右方向から左回りに計る。

○ `idir` (integer,optional → 1) · · · 軸に対して、数値や目盛線を記入する位置 (使用例参照)。

○ `cs` (real,optional → 0.3) · · · 目盛線の文字の高さ。

○ `bl` (real,optional → 0.2) · · · 目盛線の長さ。

○ `bl` (real,optional → bl*2.0/3.0) · · · 補助目盛線の長さ。

○ `code` (character,optional → 'ascii') · · · title の文字コード。

3. 使用法

· y 軸を引くときに使う。これにより y 軸のパラメータがセットされるので、`xyline` を呼ぶ前に `lxaxis(lxaxis)`、および `pyaxis(lyaxis)` を呼んでおかなければならない。

4. 使用例

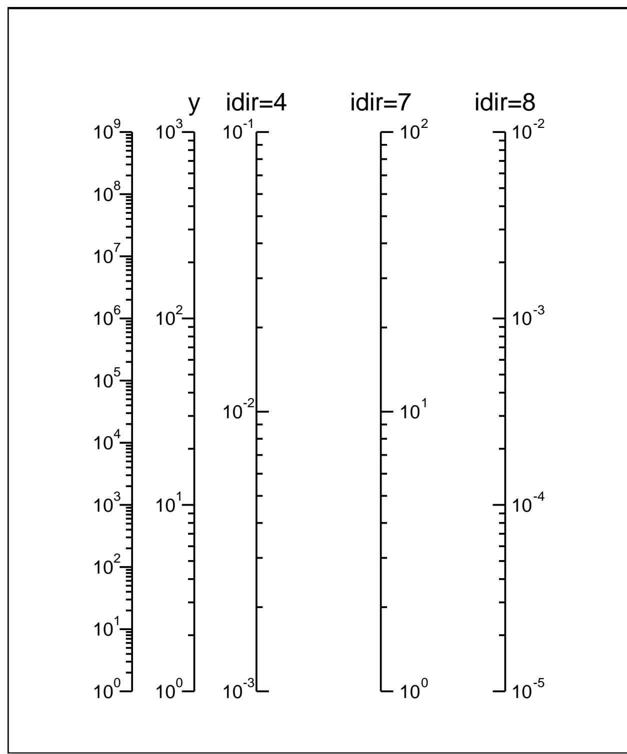


Fig.9-3 program lyaxis_example

目次へ

○片対数、両対数グラフの使用例

片対数や両対数のグラフは、対数軸を設定した後に、xygrid や xyline を呼び出せば描くことができる。ただし、以下の違いがある。

· xygrid で、x_int あるいは y_int が 10.0 のときには 10^n ごと、それ以外の時には $m \times 10^n$ ごとに、格子線が描かれる (m,n は整数)。このとき、前者の格子線は指定された線幅で、後者の格子線はその半分の線幅で描かれる。

· xygrid では、x_start、y_start は指定してあっても無視される。

· xyline で、対数軸側のデータに非正値があった場合には、描画はとまる。

使用例

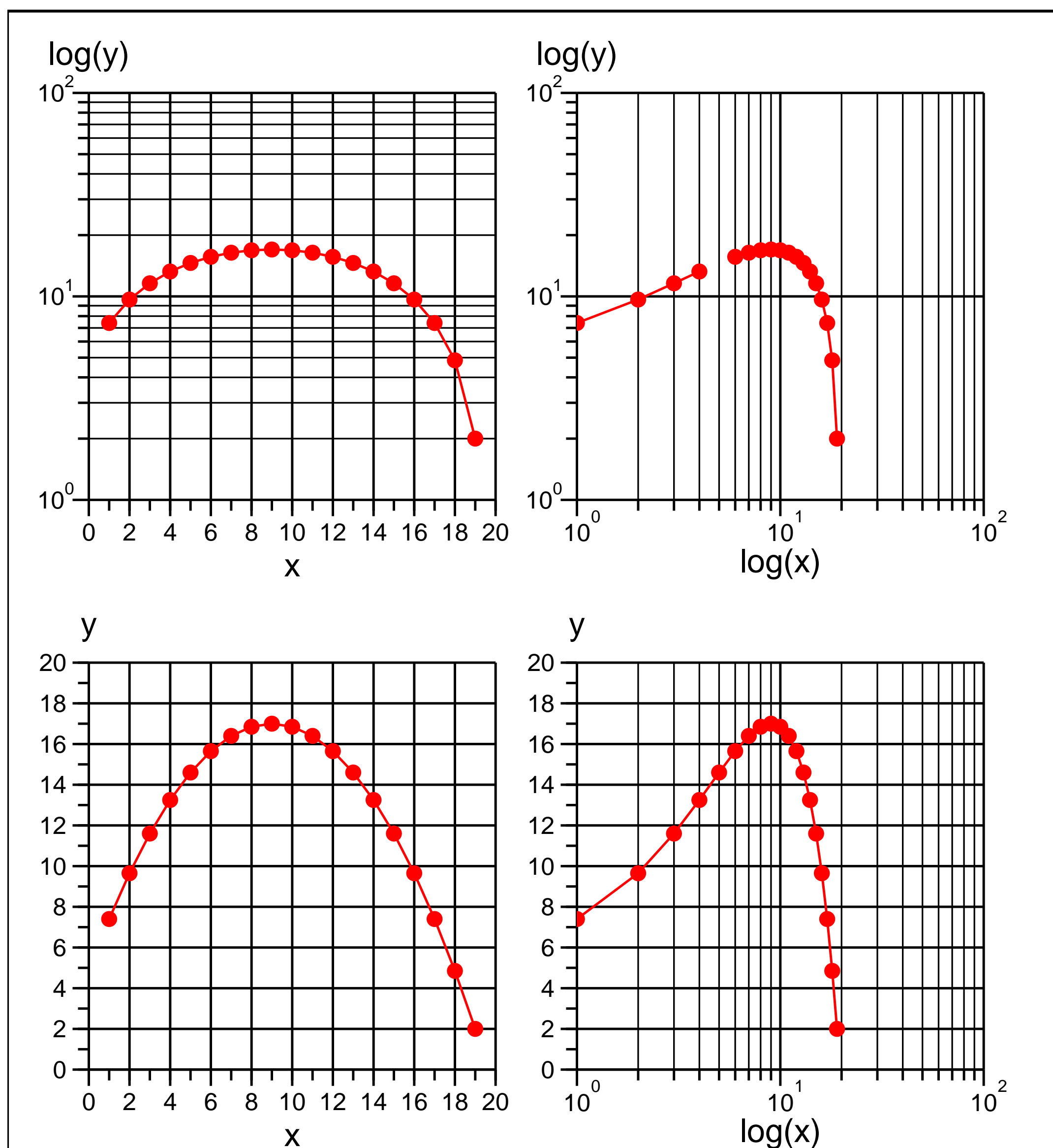


Fig.9-4 program xygrid2_example

目次へ

■ 10 二次元グラフと座標 (3) 時間軸 ■

□ txaxis (時間軸-横軸-の設定)

1. 呼び出し形式

```
call txaxis(xo,yo,axlen,d_min,d_max,d_int &
& [,title][,ang][,idir][,d_sub][,cs][,bl][,code])
```

または (処理系が長整数-8byte-をサポートしている場合のみ)、

```
call txaxis(xo,yo,axlen,i_min,i_max,i_int &
& [,title][,ang][,idir][,i_sub][,cs][,bl][,code])
```

2. 引数の説明

○ `xo`、`yo`、`axlen` ··· `paxis` と同じ。

○ `d_min` (integer,array(6))、`i_min` (integer(i_8)) ··· 軸の始点での時刻。

○ `d_max` (integer,array(6))、`i_max` (integer(i_8)) ··· 軸の終点での時刻。

○ `d_int` (integer,array(6))、`i_int` (integer(r_8)) ··· 軸の数値を書く間隔。

○ `title`、`ang`、`cs`、`bl`、`code` ··· ··· `paxis` と同じ。

○ `d_sub` (integer,array(6),optional)、`i_sub` (integer(r_8),optional) ··· 軸の数値を書く間隔。

○ `idir` (integer,optional → 1) ··· 軸に対して、数値や目盛線を記入する位置 (`paxis` 参照)。

3. 使用法

· 時間軸を横軸とするときに使う。時刻、時間の指定は、`d_xxx` の形式では、西暦年、月、日、時、分、秒をそれぞれ成分として持つ、大きさ 6 の基本整数型配列、`i_xxx` の形式では、年 (4 ケタ)、月、日、時、分、秒 (2 ケタ) を連続して並べた 14 ケタの長整数型、のどちらかで行う。ただし混在はできない。

4. 使用例

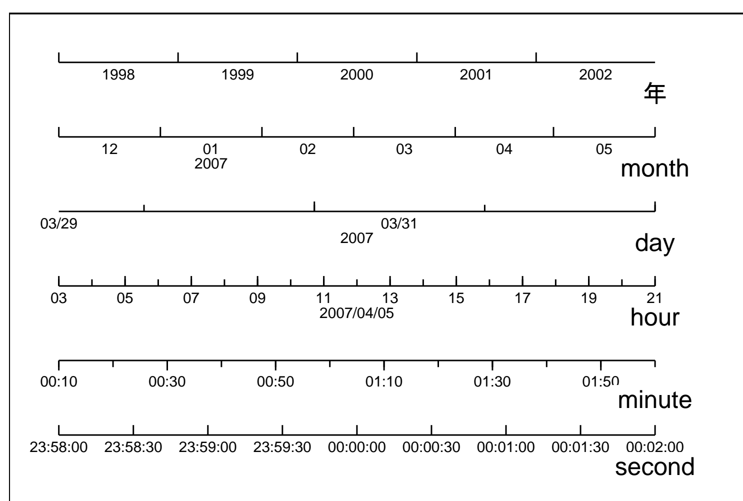


Fig.10-1 program txaxis_example

目次へ

□ tygrid (時間軸、y 軸に平行な格子線を描く)

1. 呼び出し形式

```
call tygrid(d_int,y_int[,d_start][,y_start][,penmode][,pencolor][,penwidth])
```

または、

```
call tygrid(i_int,y_int[,i_start][,y_start][,penmode][,pencolor][,penwidth])
```

・時間軸に垂直な格子のみ描くとき、

```
call ptgrid(d_int[,d_start][,penmode][,pencolor][,penwidth])
```

または、

```
call ptgrid(i_int[,i_start][,penmode][,pencolor][,penwidth])
```

2. 引数の説明

○ `d_int` (integer,array(6))、`i_int` (integer(i_8)) · · · 時間軸に垂直な格子の時間間隔。

○ `y_int` (real) · · · y 軸に垂直な格子の間隔 (軸の数値の単位)。

○ `d_start` (integer,array(6))、`i_start` (integer(i_8)) · · · 時間軸に垂直な格子を書き始める時刻。

○ `y_start` (real,optional → (図の下端) · · · y 軸に垂直な格子を書き始める最小値 (軸の数値の単位)。

○ `penmode` (integer,optional) · · · 格子線を描くペンの `penmode`。

○ `pencolor` (integer,optional) · · · 格子線を描くペンの `pencolor`。

○ `penwidth` (integer,optional) · · · 格子線を描くペンの `penwidth`。

3. 使用法

· ptaxis と pyaxis(lyaxis) を呼んで軸を書いた後、tyline を呼ぶ前に、tygrid で格子線を描く。

· 時間軸に垂直な格子線のみ描きたいときには、ptgrid を call する。

· tyline を呼んだ後に tygrid を呼ぶと、格子線によって上書きされてしまう。

4. 使用例

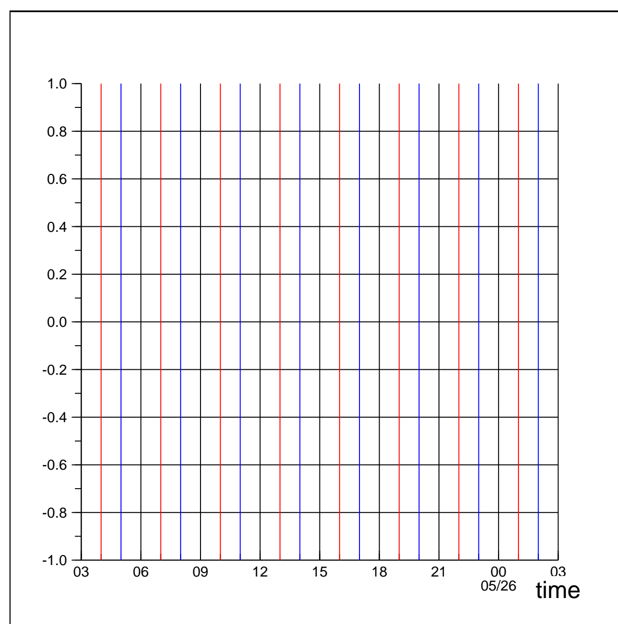


Fig.10-2 program tygrid_example

目次へ

□ tyline (*t* 軸、*y* 軸による二次元グラフ)

1. 呼び出し形式

```
call tyline(x_array,y_array[,n][,mask][,mode][,cs][,nin][,clip][,comment][,code])
```

2. 引数の説明

○ `x_array` (real(r_8),array) ··· 時刻データの配列 (Windows Date type)。

○ `y_array`、`n`、`mask`、`mode`、`cs`、`nin`、`clip`、`comment`、`code` ··· `xyline` と同じ。

3. 使用法

· `xyline` との違いは、時刻を格納する配列、x_array が基本実数型でなく、倍精度実数型でなければならないことである。x_array に格納する時刻の形式は、Excel データの処理の頻度が高いことから、Windows の Date 型とした。他の形式で時刻を表しているときは、後述の暦関係のユーティリティを利用して、Windows Date 型にあらかじめ変換しておかなければならない。

4. 使用例

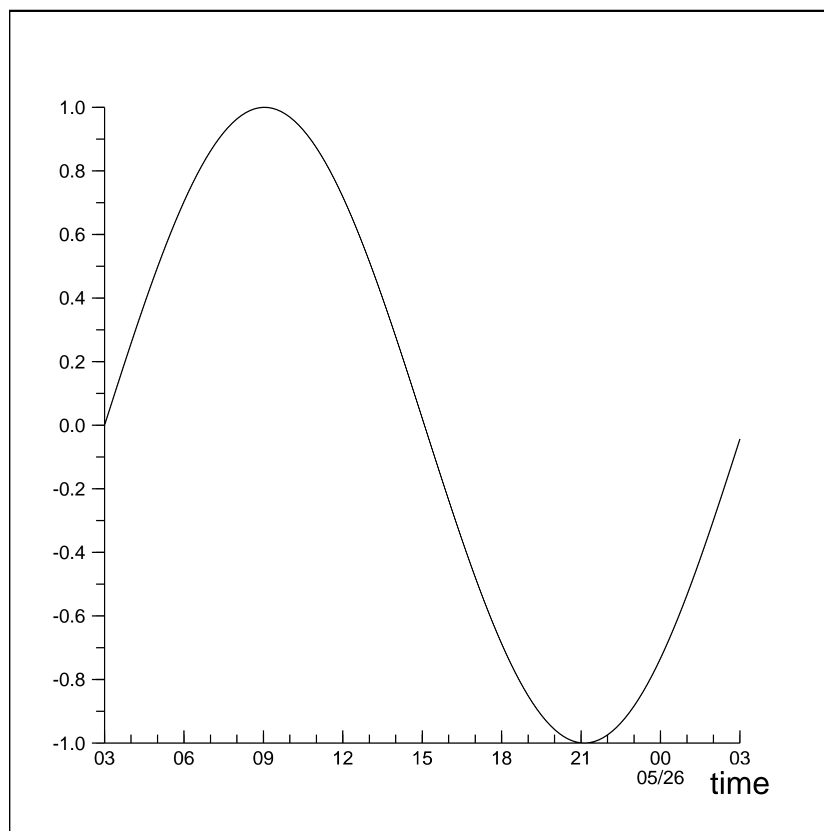


Fig.10-3 program tyline_example

目次へ

■ 11 その他 (一時的な座標移動) ■

□ subframe_begin (座標移動の開始)

1. 呼び出し形式

```
call subframe_begin([xo,yo][,ang][,fact])
```

2. 引数の説明

○ `xo` (real,optional → 0.0) · · · 一時的に座標を移動する先の x 座標。`yo` とペアで指定する。

○ `yo` (real,optional → 0.0) · · · 一時的に座標を移動する先の y 座標。`xo` とペアで指定する。

○ `ang` (real,optional → 0.0) · · · 一時的に座標を (`xo,yo`) に移動後、右方向から左回りに `ang` 回転させる。

○ `fact` (real,optional → 1.0) · · · 一時的に座標を (`xo,yo`) に移動後、`fact` 倍する。

3. 使用法

· 次の `subframe_end` とペアで使い、その間の座標軸を一時的に変換する。

· あくまで一時的であり、その間にペンの属性変更等を行った場合、戻ったときの結果は保証されない。

· 10 回までネストできる。

□ subframe_end (座標移動の終了)

1. 呼び出し形式

```
call subframe_end
```

2. 使用例

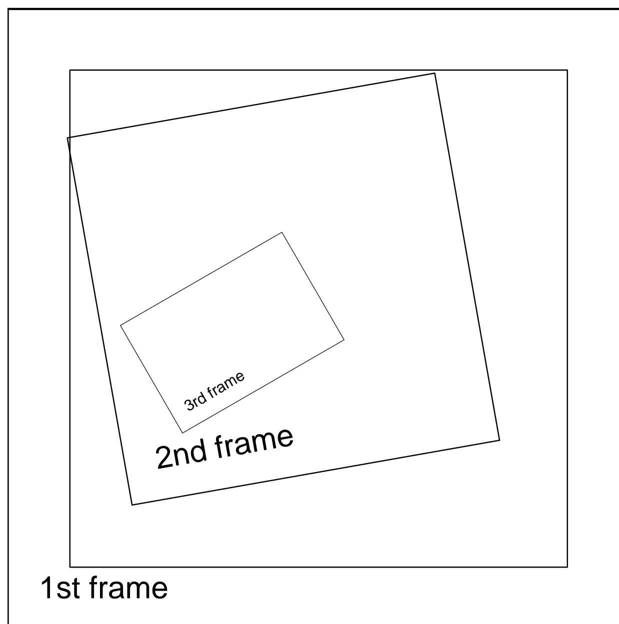


Fig.11-1 program subframe_example

目次へ

□ clip_begin (多角形領域へのクリッピング開始)

1. 呼び出し形式

(a) `call clip_begin(x_array,y_array[,n])`

または、

(b) `call clip_begin(xy_array[,n])`

2. 引数の説明

○ `x_array` (real,array) · · · x 座標の 1 次元配列。

○ `y_array` (real,array) · · · y 座標の 1 次元配列。

○ `xy_array` (structure,array) · · · 構造体 `coord_xy` の 1 次元配列。

○ `n` (integer,optional → [min(size(x_array),size(y_array))]) · · · 使用する座標の組の個数。

3. 使用法

· 次の `clip_end` とペアで使う。その間、引数で指定された多角形のクリッピング領域の、内部のみ出力する。

· あくまで一時的であり、その間にペンの属性変更等を行った場合、戻ったときの結果は保証されない。

· 10 回までネストできる。

□ clip_end (多角形領域へのクリッピング終了)

1. 呼び出し形式

```
call clip_end
```

2. 使用例

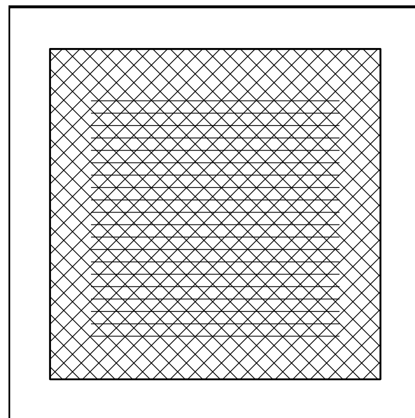


Fig.11-2 program clip_example

目次へ

■ 12 暦関係の副プログラム ■

□ grtowd (年月日時分秒から、Date 型への変換関数)

1. 呼び出し形式

```
WD=grtowd(idate)
```

2. 引数の説明

○ `idate` (integer,array(6)) · · · 大きさ 6 の整数型配列であり、順に、西暦年、暦月、日、時、分、秒。

3. 戻り値の説明

○ `WD` (real(r_8)) · · · 対応する windows Date 型での時刻。

4. 使用法

· 西暦年 (4 ケタ)、暦月、日、時、分、秒を格納した基本整数型の配列から、Windows Date 型への変換を行う。

目次へ

□ wdtogr (Date 型から、年月日時分秒への変換関数)

1. 呼び出し形式

```
idate=wdtogr(WD)
```

2. 引数の説明

○ `WD` (real(r_8)) · · · windows Date 型での時刻。

3. 戻り値の説明

○ `idate` (integer,array(6)) · · · 大きさ 6 の整数型配列であり、順に、西暦年、暦月、日、時、分、秒が返る。

4. 使用法

· `grtowd` と逆に、Windows Date 型から大きさ 6 の整数型配列への変換を行う。

目次へ

□ ldtogr (長整数型から、年月日時分秒への変換関数)

1. 呼び出し形式

```
idate=ldtogr(itime)
```

2. 引数の説明

○ itime (integer(i_8)) ··· 年 (4 ケタ)、月、日、時、分、秒 (2 ケタ) を連続して並べた 14 ケタの長整数。

3. 戻り値の説明

○ idate (integer,array(6)) ··· 大きさ 6 の整数型配列であり、順に、西暦年、暦月、日、時、分、秒が返る。

4. 使用法

· 長整数型から、大きさ 6 の整数型配列への変換を行う。

目次へ

□ add_time (年月日時分秒型時刻データの加算)

1. 呼び出し形式

```
kdate=add_time(idate,jdate)
```

2. 引数の説明

○ idate (integer,array(6)) ··· 大きさ 6 の整数型配列であり、順に、西暦年、暦月、日、時、分、秒。

○ jdate (integer,array(6)) ··· 同上。

3. 戻り値の説明

○ kdate (integer,array(6)) ··· 同上。

4. 使用法

· idate(時刻) に、jdate(時間) を加えた時刻が、kdate(時刻) として返される。

目次へ

□ day_month (指定された年、月の日数)

1. 呼び出し形式

```
md=day_month(year,month)
```

2. 引数の説明

○ year (integer) ··· 西暦年 (4 ケタ)。

○ month (integer) ··· 暦月。

3. 戻り値の説明

○ md (integer) ··· その年、月の日数。

目次へ

■ 13　例図のプログラム　■

fig. 1-1 plots

```
1   program plots_example
2       use eps_plot
3       implicit none
4
5       call plots(-8.0,3.2,file='../plots.eps')
6       call symbol(4.0,2.0,0.8,'eps_plot',location=2)
7       call symbol(4.0,0.5,0.8,'subroutine package',location=2)
8
9       call plote
10      stop
11  end program plots_example
```

説明に戻る

fig. 1-2 factor

```
1   program factor_example
2       use eps_plot
3       implicit none
4
5       call plots(-5.0,4.0,file='../factor.eps')
6
7       call symbol(2.5,3.0,0.8,'abcde',location=5)
8
9       call plot(2.5,2.0,-3)
10      call factor(0.5,1.0)
11      call symbol(0.0,0.0,0.8,'fghij',location=5)
12      call factor(2.0,1.0)
13
14      call plot(0.0,-1.0,-3)
15      call factor(2.0,1.0)
16      call symbol(0.0,0.0,0.8,'klmno',location=5)
17
18      call plote
19      stop
20  end program factor_example
```

説明に戻る

fig. 1-3 xviewp

```
1   program xviewp_example
2       use eps_plot
3       implicit none
4
```

```
5       call plots(-12.0,4.0,file='../xviewp.eps')
6       call newpen(4)
7       call pcircle(2.0,2.0,0.8)
8       call symbol(2.0,2.0,0.5,'raw',location=5)
9
10      call plot(6.0,2.0,-3)
11      call factor(2.0)
12      call pcircle(0.0,0.0,0.8)
13      call symbol(0.0,0.0,0.25,'factor=2.0',location=5)
14      call plot(-3.0,-1.0,-3)
15      call factor(0.5)
16
17      call xviewp(-4.0, 0.0, 2.0, 2.0)
18      call pcircle(1.0,1.0,0.8)
19      call symbol(1.0,1.0,0.25,'xviewp',location=5)
20
21      call plote
22      stop
23  end program xviewp_example
```

説明に戻る

fig. 2-1 plot

```
1   program plot_example
2       use eps_plot
3       implicit none
4       real :: d_array(4)=(/0.3,0.1,0.1,0.1/)
5
6       call plots(-8.0,4.0,file='../plot.eps')
7
8       call newpen(50)
9       call plot(0.5,3.0,3)
10      call plot(0.5,0.5,2)
11      call plot(3.0,0.5,2)
12
13      call plot(1.0,3.5,3)
14      call plot(3.5,3.5,2)
15      call pencolor(RGB_red)
16      call plot(3.5,1.0,2)
17
18      call pencolor(RGB_black)
19      call dashs(d_array,penmode=12)
20      call newpen(12,12)
21      call plot(4.5,3.0,3)
22      call plot(4.5,0.5,2)
23      call plot(7.0,0.5,2)
24
25      call plot(5.0,3.5,3)
26      call plot(7.5,3.5,2)
27      call pencolor(RGB_red)
```

```
28      call plot(7.5,1.0,2)
29
30      call plote
31      stop
32  end program plot_example
```

説明に戻る

fig. 2-2 pline(1)

```
1   program pline1_example
2       use eps_plot
3       implicit none
4       real :: d_array(4)=(/0.3,0.1,0.1,0.1/)
5
6       call plots(-8.0,4.0,file='../pline1.eps')
7
8       call newpen(50)
9       call pline(0.5,3.0,0.5,0.5)
10      call pline(0.5,0.5,3.0,0.5)
11
12      call pline(1.0,3.5,3.5,3.5)
13      call pencolor(RGB_red)
14      call pline(3.5,3.5,3.5,1.0)
15
16      call pencolor(RGB_black)
17      call dashs(d_array,penmode=12)
18      call newpen(12,12)
19      call pline(4.5,3.0,4.5,0.5)
20      call pline(4.5,0.5,7.0,0.5)
21
22      call pline(5.0,3.5,7.5,3.5)
23      call pencolor(RGB_red)
24      call pline(7.5,3.5,7.5,1.0)
25
26      call plote
27      stop
28  end program pline1_example
```

説明に戻る

fig. 2-3 pline(2)

```
1   program pline2_example
2       use eps_plot
3       implicit none
4
5       real :: d_array(4)=(/0.3,0.1,0.1,0.1/)
6       real :: x1(4)=(/0.5,0.5,3.3,3.3/), y1(4)=(/3.3,0.5,0.5,3.3/)
7       type(coord_xy) :: d_array2(3)= &
8       & (/coord_xy(4.0,0.5),coord_xy(5.5,3.0),coord_xy(7.0,0.5)/)
9
```

```
10      call plots(-7.5,4.0,file='../pline2.eps')
11      call dashs(d_array,penmode=12)
12      call newpen(12,12)
13      call pline(x1, y1)
14
15      call newpen(2,1)
16      call pline(d_array2,fillcolor=RGB_GreenYellow)
17
18      call plote
19      stop
20  end program pline2_example
```

説明に戻る

fig. 2-4 pbezier

```
1   program pbezier_example
2       use eps_plot
3       implicit none
4
5       call plots(-11.0,6.0,file='../pbezier.eps')
6
7       call center(1.0,1.0,0.1,3)
8       call center(5.0,3.0,0.1,3)
9       call center(1.0,5.0,0.1,3)
10      call newpen(1,2)
11      call plot(1.0,1.0,3)
12      call plot(5.0,3.0,2)
13      call plot(1.0,5.0,2)
14      call symbol(1.0,0.8,0.3,'(x1,y1)',location=8)
15      call symbol(4.7,3.0,0.3,'(x2,y2)',location=6)
16      call symbol(1.0,5.2,0.3,'(x3,y3)',location=2)
17
18      call newpen(2,1)
19      call pbezier(1.0,1.0,5.0,3.0,1.0,5.0)
20
21      call center(6.0,1.0,0.1,3)
22      call center(6.0,5.0,0.1,3)
23      call center(10.0,1.0,0.1,3)
24      call center(10.0,5.0,0.1,3)
25      call newpen(1,2)
26      call plot(6.0,1.0,3)
27      call plot(10.0,1.0,2)
28      call plot(10.0,5.0,2)
29      call plot(6.0,5.0,2)
30      call symbol(6.0,0.8,0.3,'(x1,y1)',location=8)
31      call symbol(10.0,0.8,0.3,'(x2,y2)',location=8)
32      call symbol(10.0,5.2,0.3,'(x3,y3)',location=2)
33      call symbol(6.0,5.2,0.3,'(x4,y4)',location=2)
34
35      call newpen(2,1)
```

```
36        call pbezier(6.0,1.0,10.0,1.0,10.0,5.0,6.0,5.0)
37
38        call plote
39
40        stop
41    end program pbezier_example
```

説明に戻る

fig. 3-1 newpen

```
1     program newpen_example
2         use eps_plot
3         implicit none
4
5         call plots(-8.0,3.0,file='../newpen.eps')
6
7         call newpen(10)
8         call penmode(2)
9         call pencolor(RGB_Orange)
10        call pline(0.5,0.5,7.5,0.5)
11
12        call newpen(41,penmode=5,pencolor=RGB_cyan,penwidth=11)
13        call pline(0.5,1.5,7.5,1.5)
14
15        call newpen(pencolor=RGB_Green)
16        call pline(0.5,2.5,7.5,2.5)
17
18        call plote
19        stop
20    end program newpen_example
```

説明に戻る

fig. 3-2 penmode

```
1     program penmode_example
2         use eps_plot
3         implicit none
4         integer :: i
5         character(10) :: str
6
7         call plots(-7.0,7.5,file='../penmode.eps')
8
9         str='penmode=  '
10        call newpen(3)
11        do i=1,10
12          call penmode(i)
13          write(str(9:10),'(i2)') i
14          call symbol(3.2,real(i)*0.7-0.2,0.5,str,location=6)
15          call plot(3.5,real(i)*0.7-0.2,3)
16          call plot(6.5,real(i)*0.7-0.2,2)
```

```
17        enddo
18
19        call plote
20        stop
21    end program penmode_example
```

説明に戻る

fig. 3-3 penmodeset

```
1     program penmodeset_example
2         use eps_plot
3         implicit none
4
5         real :: d_array1(8)=(/0.2,-0.1,-0.3,0.1,-0.2,0.1,-0.3,0.1/) ! Calcomp form
6         real :: d_array2(8)=(/-0.1,0.2,-0.1,0.1,-0.2,0.1,-0.3,0.2/) ! same as above
7         real :: d_array3(5)=(/0.2,0.1,0.3,0.1,0.1/)                 ! postscript form
8
9         call plots(-6.0,4.0,file='../penmodeset.eps')
10
11        call newpen(3)
12        call symbol(1.3,3.2,0.5,'(1)',location=6)
13        call penmodeset(d_array1,penmode=11)
14        call plot(1.5,3.2,3) ; call plot(5.5,3.2,2)
15
16        call symbol(1.3,2.4,0.5,'(2)',location=6)
17        call dashs(d_array2,4)
18        call plot(1.5,2.4,3) ; call plot(5.5,2.4,2)
19
20        call symbol(1.3,1.6,0.5,"(3)",location=6)
21        call penmodeset(d_array3,4)
22        call plot(1.5,1.6,3) ; call plot(5.5,1.6,2)
23
24        call symbol(1.3,0.8,0.5,"(1')",location=6)
25        call newpen(penmode=11)
26        call plot(1.5,0.8,3) ; call plot(5.5,0.8,2)
27
28        call plote
29        stop
30    end program penmodeset_example
```

説明に戻る

fig. 3-4 pencolor1

```
1     program pencolor1_example
2         use eps_plot
3         implicit none
4
5         call plots(-8.0,4.0,file='../pencolor1.eps')
6
7         call pencolor(z'ff0000')
```

```
8       call pcircle(2.0,2.0,1.5,hatch=5)
9
10      call newpen(5,pencolor=RGB_Green)
11      call prect(4.0,0.5,3.0,3.0,fillcolor=RGB_GreenYellow)
12
13      call plote
14      stop
15  end program pencolor1_example
```

説明に戻る

fig. 3-5 pencolor2

```
1   program pencolor2_example
2       use eps_plot
3       implicit none
4
5       call plots(-8.0,4.0,file='../pencolor2.eps')
6
7       call pencolor(0.6,0.8,0.7,usercolor=-4)
8       call prect(0.5,0.5,3.0,3.0)
9
10      call pencolor(-4)
11      call prect(4.5,0.5,3.0,3.0,fillcolor=-1)
12
13      call plote
14      stop
15  end program pencolor2_example
```

説明に戻る

fig. 3-6 pencolor3

```
1   program pencolor3_example
2       use eps_plot
3       implicit none
4
5       call plots(-8.0,4.0,file='../pencolor3.eps')
6
7       call newpen(5)
8       call pencolor(0, 255, 0, usercolor=-3)
9       call prect(0.5,0.5,3.0,3.0)
10
11      call newpen(10)
12      call pencolor(0,0,255)
13      call pellipse(6.0,2.0,1.5,asp=0.5,ang=10.0,fillcolor=-3)
14
15      call plote
16      stop
17  end program pencolor3_example
```

説明に戻る

fig. 3-7 hatchset

```
1  program hatchset_example
2      use eps_plot
3      implicit none
4
5      real :: x(4)=(/0.0,4.0,4.0,0.0/)
6      real :: y(4)=(/0.0,0.0,4.0,4.0/)
7
8      call plots(-9.5,5.0,file='../hatchset.eps')
9
10     call plot(0.5,0.5,-3)
11     call pline(x,y,4,hatch=2)
12     call plot(4.5,0.0,-3)
13     call hatchset(2,penmode=2,pencolor=RGB_red,penwidth=2,interval=0.2,angle=135.0,angle2=4
14     call pline(x,y,4,hatch=2)
15
16     call plote
17     stop
18 end program hatchset_example
```

説明に戻る

fig. 3-8 dotset

```
1  program dotset_example
2      use eps_plot
3      implicit none
4
5      real :: x(4)=(/0.0,4.0,4.0,0.0/)
6      real :: y(4)=(/0.0,0.0,4.0,4.0/)
7
8      call plots(-9.5,5.0,file='../dotset.eps')
9      call plot(0.5,0.5,-3)
10     call pline(x,y,4,hatch=-3)
11     call plot(4.5,0.0,-3)
12     call dotset(3,pencolor=RGB_Blue,interval=0.07,size=0.02,angle=45.0)
13     call pline(x,y,4,hatch=-3)
14
15     call plote
16     stop
17 end program dotset_example
```

説明に戻る

fig. 4-1 symbol

```
1  program symbol_example
2      use eps_plot
3      implicit none
4
5      call plots(-8.0,6.0,file='../symbol.eps')
```

```
6
7       call symbol(1.5,1.0,1.0,'漢字 abcde',code='utf8')
8       call symbol(1.5,2.5,1.0,'abcde 漢字',code='utf8')
9
10      call symbol(1.5,4.0,1.0,'abcdefg')
11      call pencolor(RGB_red)
12      call symbol(3.5,4.4,-0.5,'overwrite',location=0)
13
14      call plote
15      stop
16  end program symbol_example
```

説明に戻る

fig. 4-2 kanji

```
1   program kanji_example
2       use eps_plot
3       implicit none
4
5       call plots(-8.0,6.0,file='../kanji.eps')
6
7       call kanji(1.0,2.0,-0.8,'愛媛大学 農学部',30.0,code='sjis')
8       call kanji(2.0,0.5,0.8,'生物環境保全学 ',30.0,0,code='sjis')
9
10      call plote
11      stop
12  end program kanji_example
```

説明に戻る

fig. 4-3 number

```
1   program number_example
2       use eps_plot
3       implicit none
4
5       call plots(-8.0,4.6,file='../number.eps')
6
7       call number(0.5,3.5,0.8,-1.23456,0.0,3)
8       call number(0.5,2.5,0.8,-1.23456,0.0,-1)
9       call number(0.5,1.5,0.8,-1.23456,0.0,-3)
10      call number(0.5,0.5,0.8,-1.23456,0.0)
11      call number(4.0,3.5,0.8,-1.23456)
12      call number(4.0,2.5,0.8,-1.234)
13      call number(4.0,1.5,0.8,-1.)
14      call number(4.0,0.5,0.8,1.,location=4)
15
16      call plote
17      stop
18  end program number_example
```

説明に戻る

fig. 4-4 pfont

```
1   program pfont_example
2       use eps_plot
3       implicit none
4
5       call plots(6.5,-5.5,file='../pfont.eps')
6       call symbol(0.5,0.5,0.7,'Helvetica(default)')
7       call symbol(0.5,1.5,0.7,'Helvetica-Oblique',font=6)
8       call symbol(0.5,2.5,0.7,'Helvetica')
9       call pfont(15)
10      call symbol(0.5,3.5,0.7,'Helvetica-Narrow')
11      call symbol(0.5,4.5,0.7,'unchanged')
12
13      call plote
14      stop
15  end program pfont_example
```

説明に戻る

fig. 4-5 center

```
1   program center_example
2       use eps_plot
3       implicit none
4
5       integer :: i, j
6       real :: x, y
7
8       call plots(-10.0,8.0,file='../center.eps')
9
10      do j=1,4
11        call plot(0.5, real(j*2-1), 3) ; call plot(9.5, real(j*2-1), 2)
12      enddo
13      do i=1,5
14        call plot(real(i*2-1), 0.5, 3) ; call plot(real(i*2-1), 7.5, 2)
15      enddo
16
17      do j=1,4
18        do i=1,5
19          call center(real(i*2-1), real(j*2-1), 0.7, (j-1)*5+i)
20          call number(real(i*2-1)+0.4, real(j*2-1)+0.1, 0.4, real((j-1)*5+i))
21        enddo
22      enddo
23
24      call plote
25      stop
26  end program center_example
```

説明に戻る

fig. 5-1 pcircle

```
program pcircle_example
    use eps_plot
    implicit none

    call plots(-10.0,5.0,file='../pcircle.eps')

    call newpen(3)
    call pcircle(2.5,2.5,1.0,fillcolor=RGB_Orange)
    call pcircle(2.5,2.5,-2.0,10.,150.,fillcolor=-2,hatch=1)
    call pcircle(2.5,2.5,2.0,180.,330.)

    call pcircle(7.5,2.5,2.0,fillcolor=-2)
    call pcircle(7.5,2.5,1.5,asp=0.7,ang=10.0,fillcolor=RGB_GreenYellow)

    call plote
    stop
end program pcircle_example
```

説明に戻る

fig. 5-2 pellipse

```
program pellipse_example
    use eps_plot
    implicit none

    call plots(-8.0,4.0,file='../pellipse.eps')
    call newpen(10)
    call hatchset(51,penwidth=10,interval=0.5,angle=60.0)
    call pcircle(2.0, 2.0,1.7,asp=0.5,ang=30.,fillcolor=rgb_yellow,hatch=51)
    call pellipse(6.0,2.0,1.7,asp=0.5,ang=30.,fillcolor=rgb_yellow,hatch=51)

    call plote
    stop
end program pellipse_example
```

説明に戻る

fig. 5-3 tcircle

```
program tcircle_example
    use eps_plot
    implicit none

    call plots(-10.0,4.0,file='../tcircle.eps')

    call newpen(3)
    call tcircle(2.0,3.5,0.5,2.0,2.0,0.5)
    call center(2.0,3.5,0.2,3)
    call center(0.5,2.0,0.2,2)
```

```
11        call center(2.0,0.5,0.2,3)
12
13        call tcircle(4.5,3.5,4.5,0.5,3.0,2.0,fillcolor=RGB_yellow,hatch=30)
14        call center(4.5,3.5,0.2,3)
15        call center(4.5,0.5,0.2,2)
16        call center(3.0,2.0,0.2,3)
17
18        call tcircle(6.5,2.0,8.0,3.5,8.0,0.5,fillcolor=RGB_GreenYellow,hatch=3,full=.true.)
19        call center(6.5,2.0,0.2,3)
20        call center(8.0,3.5,0.2,2)
21        call center(8.0,0.5,0.2,3)
22
23        call plote
24        stop
25    end program tcircle_example
```

説明に戻る

fig. 6-1 plinec

```
1    program plinec_example
2        use eps_plot
3        implicit none
4
5        real :: x1(4)=(/0.5,0.5,3.5,3.5/), y1(4)=(/3.5,0.5,0.5,3.5/)
6        type(coord_xy) :: xy(3)= &
7        & (/coord_xy(4.0,0.5),coord_xy(5.5,3.0),coord_xy(7.0,0.5)/)
8
9        call plots(-7.5,4.0,file='../plinec.eps')
10
11       call pline(x1, y1, fillcolor=RGB_cyan, hatch=2)
12       call pline(xy,     fillcolor=RGB_Orange,hatch=1)
13
14       call plote
15       stop
16   end program plinec_example
```

説明に戻る

fig. 6-2 prect

```
1    program prect_example
2        use eps_plot
3        implicit none
4
5        call plots(-5.5,3.0,file='../prect.eps')
6
7        call newpen(2)
8        call prect(0.5,0.5,2.0,1.5,fillcolor=RGB_lightgreen)
9        call prect(3.0,0.5,2.0,1.5,ang=10.0,hatch=6)
10
11       call plote
```

```
12      stop
13  end program prect_example
```

説明に戻る

fig. 6-3 ptriangle

```
1   program ptriangle_example
2       use eps_plot
3       implicit none
4
5       call plots(-5.0,2.7,file='../ptriangle.eps')
6
7       call ptriangle(0.5,2.0,2.0,-1.5,fillcolor=RGB_lightgreen)
8       call ptriangle(2.5,0.5,2.0,1.5,hatch=2)
9
10      call plote
11      stop
12  end program ptriangle_example
```

説明に戻る

fig. 6-4 polygon

```
1   program polygon_example
2       use eps_plot
3       implicit none
4
5       real :: x1(4)=(/0.5,0.0,3.0,2.5/), y1(4)=(/3.0,0.0,0.0,3.0/)
6       type(coord_xy) :: xy(3)= &
7       & (/coord_xy(0.0,0.0),coord_xy(3.0,0.0),coord_xy(3.0,3.0)/)
8
9       call plots(-8.0,4.0,file='../polygon.eps')
10
11      call polygon(2.0, 2.0, x1, y1, fillcolor=RGB_cyan, hatch=2)
12      call pcircle(2.0, 2.0, 0.02, fillcolor=-2)
13      call polygon(5.5, 1.5, xy, fillcolor=RGB_Orange, hatch=1 ,location=0)
14      call pcircle(5.5, 1.5, 0.02, fillcolor=-2)
15
16      call plote
17      stop
18  end program polygon_example
```

説明に戻る

fig. 7-1 arrow

```
1   program arrow_example
2       use eps_plot
3       implicit none
4       integer :: i
5
6       call plots(-12.5,8.0,file='../arrow.eps')
```

```
7        call newpen(4)
8
9        call pcircle(1.0, 3.0, 0.05, fillcolor=0)
10       call arrow(1.0, 3.0, 2.0, 10.0, breadth=0.3, isort=1)
11       call symbol(2.0, 2.3, 0.5, 'isort=1', location=2)
12
13       call pcircle(4.0, 3.0, 0.05, fillcolor=0)
14       call arrow(4.0, 3.0, 2.0, 10.0, hlen=0.6, isort=11)
15       call symbol(5.0, 2.3, 0.5, 'isort=11',location=2)
16
17       call pcircle(8.0, 3.0, 0.05, fillcolor=0)
18       call arrow(8.0, 3.0, 2.0, 10.0, hlen=0.6, isort=21, fillcolor=RGB_cyan)
19       call symbol(8.0, 2.3, 0.5, 'isort=21', location=2)
20
21       call pcircle(12.0, 3.2, 0.05, fillcolor=0)
22       call arrow(12.0, 3.2, 2.0, 10.0, hlen=0.6, isort=31)
23       call symbol(11.0, 2.3, 0.5, 'isort=31', location=2)
24
25       call pcircle(1.0, 5.0, 0.05, fillcolor=0)
26       call arrow(1.0, 5.0, 2.0, 10.0, breadth=0.3, isort=2, fillcolor=RGB_cyan)
27       call symbol(2.0, 4.3, 0.5, 'isort=2', location=2)
28
29       call pcircle(4.0, 5.0, 0.05, fillcolor=0)
30       call arrow(4.0, 5.0, 2.0, 10.0, hlen=0.6, isort=12, fillcolor=RGB_orange)
31       call symbol(5.0, 4.3, 0.5, 'isort=12',location=2)
32
33       call pcircle(8.0, 5.0, 0.05, fillcolor=0)
34       call arrow(8.0, 5.0, 2.0, 10.0, hlen=0.6, isort=22, fillcolor=RGB_lime)
35       call symbol(8.0, 4.3, 0.5, 'isort=22', location=2)
36
37       call pcircle(12.0, 5.2, 0.05, fillcolor=0)
38       call arrow(12.0, 5.2, 2.0, 10.0, hlen=0.6, isort=32, fillcolor=RGB_cyan)
39       call symbol(11.0, 4.3, 0.5, 'isort=32', location=2)
40
41       call pcircle(1.0, 7.0, 0.05, fillcolor=0)
42       call arrow(1.0, 7.0, 2.0, 10.0, breadth=0.3, isort=3, fillcolor=RGB_red)
43       call symbol(2.0, 6.3, 0.5, 'isort=3', location=2)
44
45       call pcircle(4.0, 7.0, 0.05, fillcolor=0)
46       call arrow(4.0, 7.0, 2.0, 10.0, hlen=0.6, isort=13, fillcolor=RGB_purple)
47       call symbol(5.0, 6.3, 0.5, 'isort=13',location=2)
48
49       call pcircle(8.0, 7.0, 0.05, fillcolor=0)
50       call arrow(8.0, 7.0, 2.0, 10.0, hlen=0.6, isort=-3, fillcolor=RGB_pink)
51       call symbol(8.0, 6.3, 0.5, 'isort=23', location=2)
52
53       call pcircle(12.0, 7.2, 0.05, fillcolor=0)
54       call arrow(12.0, 7.2, 2.0, 10.0, hlen=0.6, isort=33,fillcolor=RGB_White)
55       call symbol(11.0, 6.3, 0.5, 'isort=33', location=2)
56
```

```
57      do i=1,4
58        call plot(1.0+real(i-1)*2.5,0.5,3) ; call plot(1.0+real(i-1)*2.5,1.5,2)
59      enddo
60
61      call arrow(1.0, 1.0, 1.5, 0.0, hlen=0.6, isort=1)
62      call arrow(3.5, 1.0, 0.7, 15.0, breadth=0.3, isort=2, fillcolor=RGB_yellow)
63      call arrow(6.0, 1.0, 0.7, 30.0, hlen=0.6, isort=22,  fillcolor=RGB_orange)
64      call arrow(8.5, 1.0, 0.0, 45.0, hlen=0.6, isort=23, fillcolor=RGB_cyan)
65
66      call plote
67      stop
68  end program arrow_example
```

説明に戻る

fig. 7-2 pvector

```
1   program pvector_example
2       use eps_plot
3       implicit none
4       integer :: i
5
6       call plots(-12.5,8.0,file='../pvector.eps')
7       call newpen(4)
8
9       call pvector(1.0, 3.0, 3.0, 3.3, breadth=0.3, isort=1)
10      call symbol(2.0, 2.3, 0.5, 'isort=1', location=2)
11      call pcircle(1.0, 3.0, 0.05, fillcolor=-1)
12      call pcircle(3.0, 3.3, 0.05, fillcolor=-2)
13
14      call pvector(4.0, 3.0, 6.0, 3.3, hlen=0.6, isort=11)
15      call symbol(5.0, 2.3, 0.5, 'isort=11',location=2)
16      call pcircle(4.0, 3.0, 0.05, fillcolor=-1)
17      call pcircle(6.0, 3.3, 0.05, fillcolor=-2)
18
19      call pvector(8.0, 3.0, 10.0, 3.3, hlen=0.6, isort=21, fillcolor=RGB_cyan)
20      call symbol(8.0, 2.3, 0.5, 'isort=21', location=2)
21      call pcircle(8.0, 3.0, 0.05, fillcolor=-1)
22      call pcircle(10.0, 3.3, 0.05, fillcolor=-2)
23
24      call pvector(1.0, 5.0, 3.0, 5.3, breadth=0.3, isort=2, fillcolor=RGB_cyan)
25      call symbol(2.0, 4.3, 0.5, 'isort=2', location=2)
26      call pcircle(1.0, 5.0, 0.05, fillcolor=-1)
27      call pcircle(3.0, 5.3, 0.05, fillcolor=-2)
28
29      call pvector(4.0, 5.0, 6.0, 5.3, hlen=0.6, isort=-2, fillcolor=RGB_orange)
30      call symbol(5.0, 4.3, 0.5, 'isort=-2',location=2)
31      call pcircle(4.0, 5.0, 0.05, fillcolor=-1)
32      call pcircle(6.0, 5.3, 0.05, fillcolor=-2)
33
34      call pvector(8.0, 5.0, 10.0, 5.3, hlen=0.6, isort=22, fillcolor=RGB_lime)
```

```
35        call symbol(8.0, 4.3, 0.5, 'isort=22', location=2)
36        call pcircle(8.0, 5.0, 0.05, fillcolor=-1)
37        call pcircle(10.0, 5.3, 0.05, fillcolor=-2)
38
39        call pvector(1.0, 7.0, 3.0, 7.3, breadth=0.3, isort=3, fillcolor=RGB_red)
40        call symbol(2.0, 6.3, 0.5, 'isort=3', location=2)
41        call pcircle(1.0, 7.0, 0.05, fillcolor=-1)
42        call pcircle(3.0, 7.3, 0.05, fillcolor=-2)
43
44        call pvector(4.0, 7.0, 6.0, 7.3, hlen=0.6, isort=-3, fillcolor=RGB_purple)
45        call symbol(5.0, 6.3, 0.5, 'isort=-3',location=2)
46        call pcircle(4.0, 7.0, 0.05, fillcolor=-1)
47        call pcircle(6.0, 7.3, 0.05, fillcolor=-2)
48
49        call pvector(8.0, 7.0, 10.0, 7.3, hlen=0.6, isort=23, fillcolor=RGB_pink)
50        call symbol(8.0, 6.3, 0.5, 'isort=23', location=2)
51        call pcircle(8.0, 7.0, 0.05, fillcolor=-1)
52        call pcircle(10.0, 7.3, 0.05, fillcolor=-2)
53
54        do i=1,4
55          call plot(1.0+real(i-1)*2.5,0.5,3) ; call plot(1.0+real(i-1)*2.5,1.5,2)
56        enddo
57
58        call pvector(1.0, 1.0, 2.5, 1.0, hlen=0.6, isort=1)
59        call pvector(3.5, 1.0, 4.1, 1.2, breadth=0.3, isort=2, fillcolor=RGB_yellow)
60        call pvector(6.0, 1.0, 6.6, 1.3, hlen=0.6, isort=22,  fillcolor=RGB_orange)
61        call pvector(8.5, 1.0, 8.5, 1.0, hlen=0.6, isort=23, fillcolor=RGB_cyan)
62
63        call plote
64        stop
65    end program pvector_example
```

説明に戻る

fig. 8-1 paxis

```
1     program paxis_example
2         use eps_plot
3         implicit none
4
5         call plots(-12.0,8.0,file='../paxis.eps')
6
7         call paxis(0.5, 1.5, 4.0, 0.0, 9.0, 2.0, 'idir=1', 0.0, 1, 2)!, bl=0.2)
8         call paxis(0.5, 3.0, 4.0, 0.0, 10.0, 2.0, 'idir=2', 0.0, 2, 3, bl=0.0)
9         call paxis(0.5, 4.0, 4.0, 1.0, 10.0, 2.0, 'idir=5', 0.0, 5, bl=0.2)
10        call paxis(0.5, 6.0, 4.0, 0.5, 9.0, 2.0, 'idir=6', 0.0, 6, 4, bl=0.2)
11        call paxis(6.0, 0.5, 6.0, 0.01, 0.1, 0.02, 'idir=3', 90.0, 3, 2, bl=0.2)
12        call paxis(7.0, 0.5, 5.0, 0.0, 0.07, 0.02, 'idir=4', 90.0, 4, 2, bl=0.2)
13        call paxis(8.5, 0.5, 4.0, 0.0, 8.0, 2.0, 'idir=7', 90.0, 7, 2, bl=0.2)
14        call paxis(10.0 ,0.5, 3.2, 0.0, 0.000006, 0.000002, 'idir=8', 90.0, 8, 2, bl=0.2)
15
```

```
16       call plote
17       stop
18   end program paxis_example
```

説明に戻る

fig. 8-2 pxaxis

```
1    program pxaxis_example
2        use eps_plot
3        implicit none
4
5        call plots(-12.0,10.0,file='../pxaxis.eps')
6        call newpen(3)
7        call pxaxis(1.0, 2.0, 9.0, 0.0, 9.0, 2.0, 'x-axis')
8        call pxaxis(1.0, 4.0, 9.0, 0.0, 9.0, 1.0, 'x-axis', 2, cs=0.3, bl=0.3)
9        call pxaxis(1.0, 6.0, 9.0, 0.0, 0.9, 0.3, 'x-axis', 5, intsub=3)
10       call pxaxis(1.0, 8.0, 9.0, 0.0, 0.09, 0.02, idir=6)
11
12       call plote
13       stop
14   end program pxaxis_example
```

説明に戻る

fig. 8-3 pyaxis

```
1    program pyaxis_example
2        use eps_plot
3        implicit none
4
5        call plots(-10.0,12.0,file='../pyaxis.eps')
6        call newpen(3)
7        call pyaxis(2.0, 1.0, 9.0, 0.0, 9.0, 2.0)
8        call pyaxis(3.0, 1.0, 9.0, 0.0, 9.0, 2.0,'y')
9        call pyaxis(4.0, 1.0, 9.0, 0.0, 0.9, 0.1, 'idir=4', 4)
10       call pyaxis(6.0, 1.0, 9.0, 0.0, 0.09, 0.03, 'idir=7', 7, intsub=3)
11       call pyaxis(8.0, 1.0, 9.0, -0.9, 0.0, 0.1, 'idir=8', 8)
12
13       call plote
14       stop
15   end program pyaxis_example
```

説明に戻る

fig. 8-4 xygrid

```
1    program xygrid_example
2        use eps_plot
3        implicit none
4
5        call plots(-13.0,13.0,file='../xygrid.eps')
6        call newpen(3,2)
```

```
7        call pxaxis(1.5, 1.5, 10.0, 0.0, 360.0, 60.0, 'argument',&
8             & idir=1,intsub=2,bl=0.2)
9        call pyaxis(1.5, 1.5, 10.0, -1.0, 1.0, 0.2, 'sin',idir=3,intsub=2,bl=0.2)
10       call pxgrid(60.0,penmode=1,penwidth=1)
11       call pygrid(0.2,penmode=1,penwidth=1)
12       call xygrid(60.0,0.2,30.0,-0.9,1,RGB_blue,1)
13       call pyaxis(11.5, 1.5, 10.0, -2.0, 2.0, 0.4, 'cos',idir=7,intsub=2,bl=0.2)
14
15       call plote
16       stop
17   end program xygrid_example
```

説明に戻る

fig. 8-5 xyline

```
1    program xyline_example
2        use eps_plot
3        implicit none
4        integer, parameter :: n=361
5        real :: x(0:n-1), y(0:n-1), z(0:n-1)
6        logical :: mask(0:n-1)
7        integer:: i
8
9        do i = 0, n-1
10         x(i)=i
11         y(i)=sin(real(i)/180.0*3.1415927)
12         z(i)=cos(real(i)/180.0*3.1415927)
13       enddo
14       mask=.true.
15
16       call plots(-13.0,13.0,file='../xyline.eps')
17       call newpen(3,2)
18       call pxaxis(1.5, 1.5, 10.0, 360.0, 0.0, 60.0, 'argument',&
19            & idir=1,intsub=2,bl=0.2)
20       call pyaxis(1.5, 1.5, 10.0, -1.0, 1.0, 0.2, 'sin',idir=3,intsub=2,bl=0.2)
21       call pencolor(RGB_blue)
22       call xyline(x, y, mode=-2, mask=mask, nin=5,clip='no')
23       call pencolor(RGB_black)
24       call pyaxis(11.5, 1.5, 10.0, -2.0, 2.0, 0.4, 'cos',idir=7,intsub=2,bl=0.2)
25       call pencolor(RGB_red)
26       call xyline(x, z, mode=11, mask=mask, nin=-30, clip='yes')
27
28       call plote
29       stop
30   end program xyline_example
```

説明に戻る

fig. 9-1 laxis

```
1    program laxis_example
```

```
2       use eps_plot
3       implicit none
4
5       call plots(-12.0,8.0,file='../laxis.eps')
6
7       call laxis(0.5, 1.5, 4.0, 0, 5, 'idir=1', 0.0, 1, bl=0.2)
8       call laxis(0.5, 3.0, 4.0, 1, 3, 'idir=2', 0.0, 2, bl=0.15)
9       call laxis(0.5, 4.0, 4.0, -1, 1, 'idir=5', 0.0, 5, bl=0.2)
10      call laxis(0.5, 6.0, 4.0, -2, 0, 'idir=6', 0.0, 6, bl=0.2)
11      call laxis(6.0, 0.5, 6.0, -1, 1, 'idir=3', 90.0, 3, bl=0.2)
12      call laxis(7.0, 0.5, 5.0, 1, 4,  'idir=4', 90.0, 4, bl=0.2)
13      call laxis(8.5, 0.5, 4.0, 0, 3, 'idir=7', 90.0, 7, bl=0.2)
14      call laxis(10.0 ,0.5, 3.2, 0, 9, 'idir=8', 90.0, 8, bl=0.2)
15
16      call plote
17      stop
18  end program laxis_example
```

説明に戻る

fig. 9-2 lxaxis

```
1   program lxaxis_example
2       use eps_plot
3       implicit none
4
5       call plots(-12.0,10.0,file='../lxaxis.eps')
6       call newpen(3)
7       call lxaxis(1.0, 2.0, 9.0, 0, 9, 'x-axis')
8       call lxaxis(1.0, 4.0, 9.0, 1, 4, 'x-axis', cs=0.3, bl=0.3)
9       call lxaxis(1.0, 6.0, 9.0, -3, 1, 'x-axis', 5)
10      call lxaxis(1.0, 8.0, 9.0, -4, -2, idir=6)
11
12      call plote
13      stop
14  end program lxaxis_example
```

説明に戻る

fig. 9-3 lyaxis

```
1   program lyaxis_example
2       use eps_plot
3       implicit none
4
5       call plots(-10.0,12.0,file='../lyaxis.eps')
6       call newpen(3)
7       call lyaxis(2.0, 1.0, 9.0, 0, 9)
8       call lyaxis(3.0, 1.0, 9.0, 0, 3,'y')
9       call lyaxis(4.0, 1.0, 9.0, -3, -1,'idir=4', 4)
10      call lyaxis(6.0, 1.0, 9.0, 0, 2, 'idir=7', 7)
11      call lyaxis(8.0, 1.0, 9.0, -5, -2,'idir=8', 8)
```

```
12
13        call plote
14        stop
15    end program lyaxis_example
```

説明に戻る

fig. 9-4 xygrid2

```
1     program xygrid2_example
2         use eps_plot
3         implicit none
4         integer, parameter :: n=19
5         real :: x(n), y(n)
6         logical :: mask(n)
7         integer:: i
8
9         do i = 1, n
10          x(i)=i
11          y(i)=-real(i-9)**2*0.15 +17.0
12        enddo
13        mask=.true.
14
15        call plots(-13.0,14.0,file='xygrid2.eps')
16
17        call newpen(3)
18        call pencolor(RGB_black)
19        call pxaxis(1.0, 1.0, 5.0, 0.0, 20.0, 2.0, 'x',intsub=2)
20        call pyaxis(1.0, 1.0, 5.0, 0.0, 20.0, 2.0, 'y',intsub=2)
21        call xygrid(2.0,2.0)
22        call pencolor(RGB_red)
23        call xyline(x, y, mode=3, mask=mask, clip='yes')
24
25        call newpen(3)
26        call pencolor(RGB_black)
27        call lxaxis(7.0, 1.0, 5.0,   0,    2, 'log(x)')
28        call pyaxis(7.0, 1.0, 5.0, 0.0, 20.0, 2.0, 'y',intsub=2)
29        call xygrid(1.0,2.0)
30        call pencolor(RGB_red)
31        call xyline(x, y, mode=3, mask=mask, clip='yes')
32
33        call newpen(3)
34        call pencolor(RGB_black)
35        call pxaxis(1.0, 8.0, 5.0, 0.0, 20.0, 2.0, 'x',intsub=2)
36        call lyaxis(1.0, 8.0, 5.0,   0,    2, 'log(y)')
37        call xygrid(2.0,1.0)
38        call pencolor(RGB_red)
39        call xyline(x, y, mode=3, mask=mask, clip='yes')
40
41        call newpen(3)
42        call pencolor(RGB_black)
```

```
43        call lxaxis(7.0, 8.0, 5.0,   0,    2, 'log(x)')
44        call lyaxis(7.0, 8.0, 5.0,   0,    2, 'log(y)')
45        call xygrid(2.0,10.0)
46        call pencolor(RGB_red)
47
48        x(5)=-1.0
49        mask(5)=.false.
50
51        call xyline(x, y, mode=3, mask=mask, clip='yes')
52
53        call plote
54        stop
55    end program xygrid2_example
```

説明に戻る

fig. 10-1 txaxis

```
1     program txaxis_example
2         use eps_plot
3         implicit none
4         integer :: is(6),ie(6),it(6),id(6)
5
6         call plots(-15.0,10.0,file='../txaxis.eps',code='sjis')
7         call newpen(3)
8
9         is=(/2007,04,04,23,58,00/)
10        ie=(/2007,04,05,00,02,00/)
11        it=(/0000,00,00,00,00,30/)
12        call txaxis(1.0, 1.5, 12.0, is, ie, it, 'second',bl=0.2)
13
14        call txaxis(1.0, 3.0, 12.0, 20070405001000_i_8, &
15        &  20070405020000_i_8, 2000_i_8, 'minute', idir=1, &
16        & i_sub=1000_i_8, bl=0.2)
17
18        is=(/2007,04,05,03,00,00/)
19        ie=(/2007,04,05,21,00,00/)
20        it=(/0000,00,00,02,00,00/)
21        id=(/0000,00,00,01,00,00/)
22        call txaxis(1.0, 4.5, 12.0, is, ie, it, 'hour',d_sub=id,bl=0.2)
23
24        is=(/2007,03,29,12,00,00/)
25        ie=(/2007,04,02,00,00,00/)
26        it=(/0000,00,02,00,00,00/)
27        id=(/0000,00,01,00,00,00/)
28        call txaxis(1.0, 6.0, 12.0, is, ie, it, 'day',d_sub=id,bl=0.2)
29
30        is=(/2006,12,01,00,00,00/)
31        ie=(/2007,06,01,00,00,00/)
32        it=(/0000,01,00,00,00,00/)
33        call txaxis(1.0, 7.5, 12.0, is, ie, it, 'month',bl=0.2)
```

```
34
35      is=(/1998,01,01,00,00,00/)
36      ie=(/2002,12,31,00,00,00/)
37      it=(/0001,00,00,00,00,00/)
38      call txaxis(1.0, 9.0, 12.0, 19980101000000_i_8, 20021231000000_i_8, &
39      &  10000000000_i_8, '年',bl=0.2,code='sjis')
40
41      call plote
42      stop
43  end program txaxis_example
```

説明に戻る

fig. 10-2 tygrid

```
1   program tygrid_example
2       use eps_plot
3       implicit none
4       integer :: is(6),ie(6),im(6),id(6),iss(6)
5       integer(i_8) :: jm, jss
6
7       is=(/2007,05,25,03,00,00/)
8       iss=(/2007,05,25,04,00,00/)
9       ie=(/2007,05,26,03,00,00/)
10      im=(/0000,00,00,03,00,00/)
11      id=(/0000,00,00,01,00,00/)
12      jm=030000_i_8
13      jss=20070525050000_i_8
14
15      call plots(-13.0,13.0,file='tygrid.eps')
16      call newpen(2)
17      call txaxis(1.5, 1.5, 10.0, is, ie, im, 'time', d_sub=id, bl=0.2)
18      call pyaxis(1.5, 1.5, 10.0, -1.0, 1.0, 0.2,idir=3,intsub=2,bl=0.2)
19      call tygrid(jm,0.2,penmode=1,penwidth=2,pencolor=RGB_Black)
20      call ptgrid(im,iss,penmode=1,penwidth=2,pencolor=RGB_Red)
21      call ptgrid(jm,jss,penmode=1,penwidth=2,pencolor=RGB_Blue)
22
23      call plote
24      stop
25  end program tygrid_example
```

説明に戻る

fig. 10-3 tyline

```
1   program tyline_example
2       use eps_plot
3       implicit none
4
5       integer :: is(6),ie(6),im(6),id(6),in(6),it(6)
6       integer, parameter :: n=145
7       real(r_8) :: t(0:n-1)
```

```
 8       real :: y(0:n-1)
 9       logical :: mask(0:n-1)=.true.
10       integer:: i
11
12       is=(/2007,05,25,03,00,00/)
13       ie=(/2007,05,26,03,00,00/)
14       im=(/0000,00,00,03,00,00/)
15       id=(/0000,00,00,01,00,00/)
16       in=(/0000,00,00,00,10,00/)
17
18       do i = 0, n-1
19         if( i== 0 ) then
20           it=is
21         else
22           it=add_time(it,in)
23         endif
24         t(i)=grtowd(it)
25         y(i)=sin(real(i)/real(n)*3.1415927*2)
26       enddo
27
28       call plots(-13.0,13.0,file='../tyline.eps')
29       call newpen(2)
30       call txaxis(1.5, 1.5, 10.0, is, ie, im, 'time', d_sub=id, bl=0.2)
31       call pyaxis(1.5, 1.5, 10.0, -1.0, 1.0, 0.2,idir=3,intsub=2,bl=0.2)
32       call tyline(t, y, mask=mask, clip='no')
33
34       call plote
35       stop
36   end program tyline_example
```

説明に戻る

fig. 11-1 subframe

```
 1   program subframe_example
 2       use eps_plot
 3       implicit none
 4
 5       call plots(10.0,-10.0,file='../subframe.eps')
 6       call prect(1.0,1.0,8.0,8.0)
 7       call subframe_begin(2.0,2.0,10.0)
 8         call prect(0.0,0.0,6.0,6.0)
 9         call subframe_begin(1.0,1.0,20.0,0.5)
10           call prect(0.0,0.0,6.0,4.0)
11           call symbol(0.5,0.5,0.5,'3rd frame')
12         call subframe_end
13         call symbol(0.5,0.5,0.5,'2nd frame')
14       call subframe_end
15       call symbol(0.5,0.5,0.5,'1st frame')
16
17       call plote
```

```
18        stop
19    end program subframe_example
```

説明に戻る

fig. 11-2 clip

```
1     program clip_example
2         use eps_plot
3         implicit none
4         real    :: x_array(4)=(/1.0,4.0,4.0,1.0/), y_array(4)=(/1.0,1.0,4.0,4.0/)
5
6         call plots(-5.0,5.0,file='clip.eps')
7         call prect(0.5,0.5,4.0,4.0,hatch=13)
8         call clip_begin(x_array, y_array)
9         call prect(0.5,0.5,4.0,4.0,hatch=14)
10        call clip_end
11        call prect(0.5,0.5,4.0,4.0,hatch=15)
12        call plote
13
14        stop
15    end program clip_example
```

説明に戻る