Professional Programmer's Guide to Fortran77

Clive G. Page, University of Leicester, UK

7th June 2005

Last update/bug fix: 2005 June 5.

This file contains the text of *Professional Programmer's Guide to Fortran77* published by Pitman in 1988. The book is now long out of print, so it seemed sensible to make the text freely available over the Internet. The ISO Standard for Fortran77 is, of course, now obsolete, since Fortran90 and Fortran95 have replaced it. I strongly recommend using Fortran95 as a multitude of features have been added to Fortran which make programming easier and programs more reliable.

One of the attractions of Fortran77 is that a good free compiler exists in the form of GNU Fortran, g77. At present I don't know of any free compilers for full Fortran95, but you can download a compiler for a subset language called F, which seems an excellent way to learn modern Fortran. Unfortunately this book will not be much help with F or Fortran95. Perhaps some day I may get time to revise it completely.

For more information on Fortran (and F) see these web-sites, which have links to many others:

`http://www.star.le.ac.uk/~cgp`	My home page
`http://www.fortran.com/`	Fortran Market and F home page
`http://www.ifremer.fr/ditigo/molagnon/fortran90/engfaq.html`	Excellent FAQ
`http://dsm.dsm.fordham.edu/~ftnchek/`	FTNCHEK static analyzer

Whether you write your own programs in Fortran77, or merely use code written by others, I strongly urge you to use FTNCHEK syntax checker to find mistakes. You can download versions for many platforms from the web-site listed above.

I wrote the book originally using WordPerfect, but later translated it into LaTeX to make it easier to produce on-line versions in HTML and Postscript. The text here is very similar to the published version but I took the opportunity to correct a few mistakes and make some very minor updates. If you find more errors, please let me know (email to `c` (at-sign) `page.demon.co.uk`).

The book was intentionally kept as short as possible so it could be sold at a modest price, but I managed to cover the entire Fortran77 language as defined in the ANSI and ISO Standards, including several topics which are often omitted from much larger textbooks because they are deemed to be too "advanced".

In order to encourage the writing of clear, reliable, portable, robust, and well structured code, short sections appear throughout the book offering specific guidance on the the practical use of Fortran.

Obsolete or superfluous features of the language, mainly those which have been retained for compatibility with earlier versions of Fortran, are omitted from the main text but are covered in the section 13. This is provided solely for the assistance of those who have to cope with existing poorly-written programs or ones which pre-date the Fortran77 standard.

Contents

1 Fortran とは

Fortran は数値計算の分野で最も広く用いられているプログラミング言語 (programing language) です．これは，他の言語よりも早く開発されたことや，特に科学者や技術者などのユーザが使いやすいような特長を発展させてきた経緯によるものです．以前のバージョンの規格に従って作成されたプログラムの互換性を確保するために，Fortran では古い機能を削除するのではなく，新しい機能を追加していく方法で改良されてきています．もちろん現在の標準からみれば多少時代遅れの部分もあります．これらの問題のうち一部は簡単に回避することができますが，一部は欠点として残されたままになっています．

この章では、Fortran の開発史と将来の展望を述べるとともに，その長所と欠点を簡単に述べます．

1.1 開発史

Fortran は 1950 年代前半に IBM のプログラマグループによって開発されました．John Backus に率いられたこのグループは 1957 年に IBM 704 用の Fortran コンパイラを発表しました．彼らの主要な目的の一つが “formula translation” であったため Fortran と名付けられました．しかし，Fortran は入出力 (input/output) に関して完全な制御構造 (control structure) と機能 (facility) を備えており，実際には高級言語の一つに数えられています．Fortran は急速に広まり，IBM の他のコンピュータにも開発されました．また，すぐに，他のコンピュータメーカも自社機種用の Fortran コンパイラを開発する必要に迫られるようになりました．1963 年までには主要なコンピュータメーカのほとんどが参入し，多数の異なった Fortran が存在する状況になっており，これらの Fortran のほとんどは独自の拡張が加えられていました．

このことは互換性のない方言が多数存在する混乱した状態を引き起こしてしまいました．1966 年に Fortran の規格 (American national standard・Fortran66）が作られ，多少秩序が回復されました．この規格はプログラミング言語の規格の最初のものです．この規格化は非常に意味のあるものではありましたが，言語の発展に関することについては何の規定も設けられませんでした．Fortran66 規格はきわめて慎重に制定されており，現在満たすべき言語仕様が指定されるだけで他の機能を加えることは禁止されていませんでした．このため，時がたつにつれて再び独自の拡張が多くなされるようになり，新たな規格の必要性が明白になりました．そして，現在の規格である Fortran77 の制定につながりました．

1.2 標準化

Fortran プログラムの最も重要な特徴のうちの 1 つは，それらの可搬性 (portability・あるコンピュータ・システムから別のコンピュータ・システムに容易に移植できること) です．今日ハードウェアは数年で世代交代するのに対して，ほとんどの高品位のソフトウェアはそれよりも長く使われるため，ソフトウェアの可搬性は重要です．コンピュータ・ネットワークの発展も可搬性の向上を進める一因となっています．

可搬性を実現するための第一歩は，標準的な形式で書かれたプログラムがどのシステムでも処理できるようにすることです．この必要性はよく認識されていて，主要なプログラミング言語の開発に反映されています．しかし実際には新しい規格は無視されることも多く，BASIC や Pascal などの言語の規格に合致した（standard-conforming）なシステムはいまだまれなものです．

幸いにも Fortran は恵まれた状態にあります．現行のほとんどの Fortran システムは Fortran77 と呼ばれる規格に合致するように作られています．この規格は 1977 年に American National Standards Institute (ANSI) に作られ，その後 International Standards Organisation (ISO) にも採用されています．これらは ANSI X3.9-1978，ISO 1539-1980 として出版されています．この本で言う標準 Fortran とは，

この規格によって規定された Fortran77 を意味します.

Fortran は現在，市場に出回っているほとんど全てのコンピュータで利用可能なコンパイラ (compiler) が存在する最も多く使われるプログラミング言語の一つです．Fortran77 は数値だけでなく文字列の操作も行いやすく，さらに，ファイル操作や入出力の仕組みも強力であるため，それ以前の規格よりもより広範囲の種類のプログラミングができるようになっています.

1.2.1 フル規格とサブセット規格の Fortran

ANSI 標準では 2 つのレベルの Fortran77 を定義しています．簡略化されたサブセット規格の Fortran は，言語の全ての規格を実装するのが困難な非力なコンピュータで使うために作られたものです．しかし，現在ではパーソナルコンピュータでさえもフル規格の Fortran を扱うのに十分な能力を持っており，サブセット規格はもはや使われることはありません．従ってこの本ではフル規格の Fortran だけを扱います.

1.2.2 Fortran90

公式的には Fortran77 の ISO 規格は Fortran90 の ISO 規格に置き換えられています．後者は，プログラミングを容易に，また，より可搬的かつロバスト（robust) なプログラムを生成できるように，他の高級言語ですでに取り入れられているさまざまな機能を取り入れたものです．Fortran77 はすべて Fortran90 に正しいサブセットとして含まれているため，Fortran77 規格に合致した（standard-conforming）既存のプログラムは自動的に Fortran90 の規格に対しても合致することになります．ただし，実績のある Fortran90 コンパイラが利用可能な現在でも，多くのユーザが Fortran77 を使っていますが，これは多少なりとも Fortran77 の拡張機能を使っているためと思われます.

1.3 長所と欠点

Fortran は様々な特長の独自の組み合わせによって広く使われるようになりました．Fortran は論理計算や文字列の操作については他の言語とほぼ同程度，数値計算と入出力の機能では他の言語の追随を許さない機能を持っています．Fortran はプログラム単位ごとの分割コンパイルなど大規模なプログラム開発に必要な機能を持っている一方で，非常に単純な言語で，コンピュータの専門家でなくても非常に簡単に修得できます．また，Fortran で書かれたプログラムは他の主要言語と比べて可搬的 (portable) です．言語が直接的でコンパイラの Fortran の文の取り扱いが非常に洗練されたレベルに達しているため，生成される実行プログラムの効率が非常に高くなっています．さらに，既存の手続き (procedure) を新しいプログラムに取り込むことが容易であるために，旧いプログラムを元にして新しいプログラムを開発することが非常に簡単になっています.

一方で Fortran には多くの弱点や欠点があるのも否定できません．これらの弱点や欠点は，例えば，英字名が 6 文字までに制限される点やプログラムのレイアウトが固定されていること，文番号を使わなければならない点など，Fortran が開発された時点から存在し，とっくに解消されていなければならなかった問題です.

Fortran には寛容な規則と多くのディフォルト値が設定されています．これはプログラミングの手間を省くものですが，一方でシステムがプログラマのミスを検出するのを難しくしてしまいます．例えば，他のプログラミング言語ではすべての変数の型があらかじめ宣言されていなければならないのが普通です．Fortran ではこのような規則はありませんが，一方で，変数名をミスタイプすると，同じ変数のつもりがコンパイラは別の変数として処理してしまいます．このような間違いは致命的ですが，システムがこの誤りを検出することは簡単なことではありません.

Fortran にはプログラミングを容易にする現代的な制御構造やデータ構造を欠くという問題もありますが，この問題は，他の制約とともに Fortran90 で解消されています.

1.4 注意事項

1.4.1 拡張機能と可搬性

コンピュータメーカには「より良い」Fortran システムを提供することで他との競争を有利に進めようとする傾向があります．普通，これは言語の拡張機能を付加することで行われます．これは規格に合致したプログラムを正しく処理するため，Fortran 規格には反していません．また，長期的にはこのような拡張機能を吸収する形で言語が進化しているのが実際のところです．しかし，短期的にはプログラムの可搬性を妨げることになるため，このような拡張機能を使用することには問題があります.

1977 年に制定された規格に対して，過去の経緯による無意味な制約にこだわることへの失望感がありました．米国国防省はメーカが実装すべき拡張のリストを公開しています．その主なものは次のようなものです.

- `END DO` 文
- `DO WHILE` ループ
- `INCLUDE` 文
- `IMPLICIT NONE` 機能
- 整数型への bit 演算の組込関数 (intrinsic function)

多くの Fortran システム，特に米国で開発されたシステムでは，これらの機能はサポートされていますが，全てのシステムでサポートされているわけではないので，可搬性が必要な場合にはこれらの機能を使うべきではありません.

Fortran77 で最も面倒な制約は英字名 (symbolic name) が 6 文字以内に限られることでしょう．このため，プログラマは英字名としてわかりやすい名前を使うことができず，短縮，省略，頭文字を用いるなどの工夫をする必要があります．この制約をゆるめたシステムを使用している場合にはこの規則を無視したくなりますが，重大な影響を及ぼしうることを忘れてはいけません．`TEMPERATURE` と `TEMPERED` いう二つの変数を使うプログラムを考えてください．多くのコンパイラでは全く問題を生じませんし，二三のコンパイラでは変数の長さのためエラーとなるでしょう．しかし，7 文字目以降は単に無視するコンパイラも存在し，この場合にはどちらの変数も `TEMPER` という一つの変数に扱われてしまいます．このような動作は困ったものですが，規格に従ったプログラムが正しく処理されることを要求する Fortran の規格には完全に合致しています.

このような問題を避けるには一切の拡張を使うことをやめて，標準の Fortran 規格に従ってプログラムを作成するしかありません．大抵のコンパイラには非標準の文に警告を出すスイッチあるいはオプションがあります．この本では標準の Fortran77 について説明し，非標準の事項については注意書きを付します.

1.4.2 ガイドライン

コンピュータプログラムが正しく動作するためには，十分な注意と正確さを持ってプログラミングする必要があります．上で説明したように Fortran の寛容な規則のためにシステムが間違いを見つけることが難しく，他の言語よりもこのことを強く意識する必要があります．正しくプログラムするた

めには単に規格に従ったプログラムを書くだけでなく，落とし穴から身を守るようにすることが重要なのです．

NASA の打ち上げた惑星探査機の失敗がわかりやすい例でしょう．結局この失敗の原因は次のような一文にあることが突き止められました．

```
DO 15 I = 1.100
```

正しくは

```
DO 15 I = 1,100
```

とすべきで，カンマ “,” がドット “.” になってしまっています．Fortran は空白文字を無視して処理するため，コンパイラにはこの文は

```
DO15I=1.100
```

のように見えるため，これは変数 DO15I に代入する完全に正しい代入文として扱われてしまい，決して意図したようには扱ってはくれません．

Fortran77 では DO 文の後にカンマを入れることを許しているので，この文は次のように書くことができます．

```
DO 15, I = 1,100
```

この書き方には上記の過ち (single-point failure) がもはや生じる余地がないという大きな利点があります．

Fortran にはこの種のリスクが数多くありますが，長年のプログラミングから得られたノウハウを取り入れていくことでこのようなミスを犯す危険を小さくすることができます．この本では全編を通して様々な推奨例やガイドラインを示します．あまりに時代遅れの機能や不十分な機能は本文では説明せず，13 章で触れることにします．

この本ではプログラム設計の技法やソフトウェア工学に踏み込むだけのスペースはありません．この本で推奨した事柄は可能な限り modular design と構造化プログラミング (structured programming) の技法に従っていますが，大規模なプログラミングに着手するときには，あらかじめこれらのトピックについて詳しく勉強してください．

2 Fortran の基礎

この章ではプログラム例を使って Fortran の基本的な考え方を示します．なるべくプログラムを簡単にするために，示したプログラムは電卓の計算を超えるものではありませんし，通常の商用プログラムなら普通に見られるような洗練されたものでもありません．しかし，これらのプログラムは実行可能なものであり，皆さんが Fortran を利用可能ならばすぐに試してみることができます．また，そうでなくても Fortran のそれぞれの命令がどのように全体のプログラムを構成していくかを見ることができるので，プログラムを読んでみることは有意義なことです．

2.1 文

次のような最も簡単なプログラムから始めます．

```
PROGRAM TINY
WRITE(UNIT=*, FMT=*) 'Hello, world'
END
```

多分すぐにわかると思いますが，このプログラムのすることは，例の Hello，world というメッセージをコンピュータの画面に表示するだけです．しかし，プログラムの書き方 (layout) や構造については説明が必要でしょう．

このプログラムは 3 行からなり，それぞれの行は一つの Fortran の文 (statement) を含んでいます．Fortran の文はそれぞれ 1 行ずつ（必要ならばそれ以上）に書かなければいけません．ただし，最初の 6 カラムは文番号 (statement label) と継続行マーク (continuation marker) に使われます．この例ではどちらも必要ないので，どの行も左の 6 カラムは空白になっています．

PROGRAM 文はプログラム単位 (program unit) に名前を与え，また，これが主プログラム単位 (main program unit) であることを宣言します．主プログラム単位以外のプログラム単位については後述します．プログラム名は初めの文字がアルファベットで（残念なことですが）全部で 6 文字以内という Fortran の規則の許す範囲で自由に決めることができます．プログラム名には，プログラムを打ち込んだファイル (source code) のファイル名と同じ名前を用いるのが良いでしょう．

WRITE 文は出力を行います．括弧の中に，出力の位置と形式（書式）を指定した制御情報 (control) を記します．この例では，UNIT=*は出力先を標準出力 (standard output) にします．標準出力とは普通はコンピュータの画面です．また，FMT=*は書式をディフォルト (default)[1]の出力形式（専門的には並びによる書式 (list-directed format) と呼ぶ）にします．ここで使われているアスタリスク (*) は Fortran では頻繁に用いられ，ディフォルトあるいは標準的な選択を行うことを意味します．このプログラムは WRITE 文の短縮形を用いて，多少短くすることができます．

```
WRITE(*,*) 'Hello, world'
```

このように UNIT=や FMT=などの語は書かなくても良いのですが，書いておいた方がプログラムが読みやすくなります．制御情報並び (control list) の各要素は，Fortran で用いられる他の並びと同様にカンマで区切る必要があります．制御情報の後には出力するデータのリストを記します．この例では，アポストロフィー（シングルクオテーション）で囲んだ文字定数 (character constant) の一つだけです．

また，全てのプログラム単位の最後には END 文が必要です．END 文はプログラムがコンパイルされる（機械語 (machine code) に翻訳される）ときにコンパイラにプログラム単位の終端であることを伝え，プログラムの実行時 (run-time) にはプログラムを停止し，制御をオペレーティングシステム (operating system, OS) に戻します．

標準の Fortran の文字セット (character set) には小文字が含まれないため，一般論としては Fortran の文は全て大文字で書く必要があります．しかし，Fortran のコンパイラはコンピュータがサポートする文字ならば処理することができます．また，先ほどのプログラム内のメッセージのような文字定数ではこの規則は適用されません．

2.2 式と代入

次の例は多少実用的な例で，住宅ローンのような固定期間のローンの返済金額を求める計算です．返済は利子と元金の分からなり，年間返済額は次の公式で計算できます．

$$\text{payment} = \frac{\text{rate} \times \text{amount}}{1 - (1 + \text{rate})^{-\text{nyears}}}$$

この公式では rate は年利のことで小数値ですが，通常はパーセントで表すのが普通のため，この変換もプログラムで行います．

```
PROGRAM LOAN
WRITE(UNIT=*, FMT=*)' Enter amount, % rate, years'
```

[1]訳注：陽に示さないときに選択されるもの（こと）をディフォルトという．

```
      READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
      RATE = PCRATE / 100.0
      REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
      WRITE(UNIT=*, FMT=*)' Annual repayments are ' , REPAY
      END
```

このプログラムでは，READ 文と代入文 (assignment statement) の 2 つが新たに使われています．どちらも変数に数値を代入するためのものです．

READ 文は WRITE 文と似た形式で書かれます．この例では，プロンプトに従ってコンピュータターミナルから入力された 3 つの数値を読み込んで，3 つの変数 AMOUNT，PCRATE，NYEARS に順に代入します．WRITE 文の場合と同様に，FMT=*は並びによる書式 (list-directed) あるいは自由形式 (free-format) の入力を表し，カンマやスペースで区切られた数値や 1 行に 1 数値ずつ入力された場合についても読み込むことができます．

4 番目の文は代入文で，PCRATE を 100 で割り，その結果を RATE という別の変数に代入します．その次の代入文は公式に従って返済額を計算し，結果を変数 REPAY に代入します．

このプログラムではたくさんの算術演算子 (arithmetic operators) が使われています．他の言語と同様に / は除算 (division) を，* は乗算 (multiplication) を表します．また，Fortran では ** はべき乗 (exponentiation)，すなわち a**b は a の b 乗を表します．表現が曖昧になるため，二つの演算子を続けて記すことはできません．例えば **-N ではなく **(-N) としなければいけません．

もう一点プログラムの書き方に関する特徴として，Fortran の文ではスペースが無視されることが挙げられます．このため，適宜スペースを入れることでプログラムの可読性を向上することができます．

先ほどのプログラムを実行すると次のような表示が得られます．

```
Enter amount, % rate, years
20000, 9.5, 15
Annual repayments are 2554.873
```

なお，並びによる書式を用いた出力では計算の精度によって出力される桁が自動的に決められますが，コンピュータによって計算の精度が異なるため，若干上記の出力と異なることがあります．

2.3 整数型と実数型

先ほどの LOAN プログラムでは暗黙の型の規則を用いているため，多少簡略化されたプログラムになっています．これについては多少説明が必要でしょう．

コンピュータは様々な方法で数値を記憶しますが，数値の最も一般的な型 (data type) は整数 (integer) と実数 (real) です．整数型の変数 (integer variable) はものを数えるのに使われ，正確にその値を記憶することができます．これ以外の目的によく使われるのは，浮動小数表現 (floating-point representation) を用いて数値を記憶する実数型の変数 (real variable) です．浮動小数表現とは整数部分と小数部分を記憶するものです．

実数型の欠点は浮動小数表現では数値を正確には記憶できない点で，正しい桁は高々最初の 6 乃至 7 桁です．このため，適切な変数型を選択することが大切です．先ほどのプログラムでは返済年数 NYEARS が整数型，それ以外の変数は実数型になっています．

定数 (constant) の型は書かれた形式から判定できます．例えば，文字定数は引用符で囲まれていますし，数値データでは小数点があれば，整数型ではなく実数型であるとわかります．例えば，1 ではなく 1.0 と表された「1」は実数型になります．

変数の型を宣言する方法は複数あります．その一つはプログラムの初めに陽に型宣言 (explicit type statement) を行う方法です．先ほどのプログラムで型宣言をすると次のようになります．

```
      PROGRAM LOAN
      INTEGER NYEARS
      REAL AMOUNT, PCRATE, RATE, REPAY
```

他の多くのプログラミング言語ではこのような型宣言が必要ですが，Fortran では必要というわけではありません．プログラマの立場からは，プログラミングが簡単になる，あるいは，Fortran プログラマは楽をできるということになります[2]．Fortran でこれらの型宣言が省略できるのは，型宣言がない場合には名前の最初の一文字目で型が決められるためです．この決まりは次のようなものです．

初めの文字 I-N	整数型 (integer type)
初めの文字 A-H あるいは O-Z	実数型 (real type)

先ほどのプログラムで返済年数が単に YEARS でなく NYEARS としたのはこれを整数型にするためで，他の変数は実数型になります．

2.4 DO ループ

住宅ローンの返済額は普通定額ですが，ローン残高は毎年減少していきます．次のプログラムは DO ループ (DO-loop) を用いて，ローン残高をシミュレートします．

```
      PROGRAM REDUCE
      WRITE(UNIT=*, FMT=*)' Enter amount, % rate, years'
      READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
      RATE = PCRATE / 100.0
      REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
      WRITE(UNIT=*, FMT=*)' Annual repayments are ', REPAY
      WRITE(UNIT=*, FMT=*)' End of Year Balance'
      DO 15,IYEAR = 1,NYEARS
          AMOUNT = AMOUNT + (AMOUNT * RATE) - REPAY
          WRITE(UNIT=*, FMT=*) IYEAR, AMOUNT
 15   CONTINUE
      END
```

プログラムの初めの部分は先ほどのものと同じで，その後に二つの数値を出力する別の WRITE 文を加えたものです．

DO 文はループ (繰り返し・loop) の開始を示します．ループの中の文はループ変数 (loop control variable) IYEAR を 1 から NYEAR まで順に変化させながら，繰り返し実行されます．ループ内の最初の文は，AMOUNT の値に年間の利息を加え，返済額を引いてローンの残額を更新しています．すなわち，年の最終日のローン残額が AMOUNT に記憶されています．その次の文は，年数と AMOUNT の値を出力しています．その次の行に CONTINUE 文が使われています．これは実際に何かをするわけではなく，位置を示すために使われるものです．ループの開始を示す DO 文の番号と同じ 15 の文番号 (statement label) がこの CONTINUE 文に振られているので，ループの終端はこの行になります．

ループの中の文は外部の文と比べて少し字下げされています．これは必ず必要というわけではありませんが，プログラムの構造を見やすくするため Fortran プログラマは非常によく用います．

このプログラム REDUCE は，次のような，数学的には正しいものの大変読みにくい表を出力します．

```
 Enter amount, % rate, years
```

[2]訳注：暗黙の型宣言は致命的なバグの原因となることが多いため，IMPLICIT NONE 文あるいはコンパイラオプションを用いてこの機能を禁止して，全ての変数を型宣言することをすすめます．14.1.1 節参照．

```
2000, 9.5, 5
  Annual repayments are   520.87262
  End of Year Balance
 1  1669.12732
 2  1306.82178
 3  910.097229
 4  475.683838
 5  0.00118229422
```

2.5 書式付き出力

先ほどの表は，小数点の位置を揃え，小数点以下 2 桁に統一すると読みやすくなります．表の最後の値は 1/3000 ポンド以下で，コンピュータの誤差範囲内の実質的にはゼロの値です[3]．より読みやすい出力は，今まで使ってきた並びによる書式に変えて，陽に書式指定 (format specification) を行うことで簡単に得ることができます．書式指定を行うには最後の `WRITE` 文を，例えば次のように書き直します．

```
WRITE(UNIT=*, FMT=' (1X,I9,F11.2)' ) IYEAR, AMOUNT
```

この修正によって次のように整った数表が得られます．

```
  Enter amount, % rate, years
2000, 9.5, 5
  Annual repayments are   520.87262
  End of Year Balance
        1    1669.13
        2    1306.82
        3     910.10
        4     475.68
        5       0.00
```

書式指定は括弧で囲み，さらにこれは文字列定数であるため，シングルクォテーション（引用符）で囲む必要があります．書式指定子の並びの最初の `1X` は行送りの制御 (carriage-control convention) のためのものですが，この機能は後の Fortran では削除されており，この場合には各行の先頭に余計な空白を出力するだけです．ここでは単に，非常に古い Fortran コンパイラとの互換性だけのために使っています．残りの書式指定子はそれぞれの値の形式を指定しています．まず，`I9` は最初の整数値が 9 桁を使って出力することを，また，同様に `F11.2` はその次の実数値（浮動小数値）を 11 桁，小数点以下 2 桁で出力することを指定しています．

数値は指定された出力欄にすべて右寄せで出力されます．このプログラム例では見出しに合わせて出力されるように数値の出力桁数を決めています．

2.6 関数

Fortran には様々な計算をする上で有用な組込関数 (intrinsic function) が備えられています．種々の型変換 (data type conversion) に加え，平方根，最大値や最小値，sin，cos などです．また，プログラマは独自の関数を定義することができます．次の例は三角形の面積を求めるプログラムで，上記二通りの関数が使われています．

[3]訳注：これは Gnu Fortran77+WindowsXP を用いて得た出力で，原著の最後の値は 2.980016E-04 になっています．

三辺の長さがそれぞれ a，b，c である三角形の面積は次の公式で与えられます.

$$\begin{aligned} s &= (a+b+c)/2 \\ \text{area} &= \sqrt{s(s-a)(s-b)(s-c)} \end{aligned}$$

```
      PROGRAM TRIANG
      WRITE(UNIT=*,FMT=*)' Enter lengths of three sides:'
      READ(UNIT=*,FMT=*) SIDEA, SIDEB, SIDEC
      WRITE(UNIT=*,FMT=*)' Area is ' , AREA3(SIDEA,SIDEB,SIDEC)
      END

      FUNCTION AREA3(A, B, C)
*Computes the area of a triangle from lengths of sides
*三角形の三辺の長さから面積を求める
      S = (A + B + C)/2.0
      AREA3 = SQRT(S * (S-A) * (S-B) * (S-C))
      END
```

このプログラムは二つのプログラム単位 (program unit) からできています．最初のプログラム単位は主プログラム (main program) で今まで見てきた例と同じ形式です．唯一新しい点は，WRITE 文に関数 AREA3 の呼び出し (call) を含んでいることです．この関数は三角形の面積を計算するもので，関数副プログラム (function subprogram) と呼ばれる独立したプログラム単位で定義された外部関数 (external function) です.

外部関数は関数名と仮引数 (dummy argument) のリストを記した FUNCTION 文で始まります．この関数では A，B，C の 3 つの仮引数が使われています．関数が呼ばれるときに，実引数 (actual argument) である SIDEA，SIDEB，SIDEC の値がそれぞれに対応した仮引数に渡されます．外部関数内の変数名は主プログラム内の変数名とは全く無関係で，実引数と仮引数は引数並びの中の位置だけで関連付けられます．すなわち，SIDEA の値は A に，SIDEB の値は B のように渡されていきます．関数名はその関数副プログラムの中で変数として使うことができます．この変数の値は呼び出し側のプログラムにとっては関数の戻り値になるので，この関数が終了して呼び出し側に制御を戻す前に，この変数に値を代入しておかなければなりません.

関数の中では仮引数も変数として使うこともできます[4]．プログラム例の最初の代入文では，まず和を求め，2 で割った後に局所変数 (local variable) S に代入しています．次の代入文では，引数として与えられた値の平方根を計算する組込関数 SQRT を用い，この結果を関数名と同じ名前の変数に代入することで，関数の戻り値として呼び出し側のプログラムに返しています．このプログラム単位の END 文は計算を終了させるのではなく，呼び出し側のプログラムに制御を戻すためのものです.

このプログラム例のもう一つの新しい点は，関数の動作を説明したコメント行 (comment line) です．コメント行は 1 カラム目のアスタリスク*で始まります[5].

この二つのプログラム単位はそれぞれ別のソースファイルに書くこともできますし，別々にコンパイルすることもできます．その場合，別々にコンパイルされたオブジェクトモジュール (object module) から実行プログラム (executable program) を作成するには，リンク (linking) と呼ばれる操作が必要になります．これは，このような単純なプログラムでは無駄な作業が増えるだけですが，次章で説明するように，大きなプログラムを作成する場合には利点があります.

このような非常に単純なプログラムでは，その他の入出力部分と計算部分を分離する必要は全くありませんが，複雑なプログラムでは有効なやり方です．それは，同じ計算を他の必要な箇所でも関数

[4]訳注：仮引数の値を書き換えると，呼び出し側の実引数の値も変わってしまいますので注意が必要です.
[5]訳注：C でもよい．4.2.2 節参照.

を呼び出すだけで済むようになるためと，複雑な作業を取り扱いが容易な小さな独立した単位に分割することで，プログラムの複雑さを低減することができるためです.

2.7 IF ブロック

もう一つ重要な Fortran の重要な制御機構 (control structure) に IF 文があります．一群の文（ブロック）をある条件のもとで実行させたり，異なった動作を選択させるためのものです.

先ほどの関数 AREA3 には不適切な入力を防ぐ機能がないという明らかな欠点があります．例えば 1.0, 2.0, 7.0 のように，三角形を作れない 3 つの実数の組み合わせはいくらでもあります．簡単な計算によって，そのようなあり得ない組み合わせでは平方根関数の引数が負になることがわかりますが，これは数学的に許されません．Fortran システムではこのようなエラーは実行時 (run-time) に検出されますが，システムによってその動作は異なります．またたとえ検出される場合でも，ユーザに何に問題があるのか伝えるためには，“negative argument for square-root” のようなメッセージでは不十分です．次の AREA3 の改訂版は多少ユーザに使いやすくなっています．残念ながら，WRITE 文の中で用いられている関数の中で WRITE 文を使うことができないので，エラーメッセージは STOP 文で出力する必要があります.

```
      REAL FUNCTION AREA3(A, B, C)
*Computes the area of a triangle from lengths of its sides.
*If arguments are invalid issues error message and returns zero.
*三角形の三辺の長さから面積を求める
*あり得ない値に対してはエラーメッセージを出力し 0 を返す
      REAL A, B, C
      S = (A + B + C)/2.0
      FACTOR = S * (S-A) * (S-B) * (S-C)
      IF(FACTOR .LE. 0.0) THEN
          AREA3 = 0.0    ! The translator added this line.
          STOP  'Impossible triangle '
      ELSE
          AREA3 = SQRT(FACTOR)
      END IF
      END
```

IF 文は ELSE と END IF で二つのブロックを囲んで使います．初めのブロックは IF 文中の条件が真 (true) のとき，また，次のブロックは条件が偽 (false) のときに実行されます．これらの IF ブロック中の文は見やすいように字下げされますが，これはループの場合と同様にあくまでもプログラミングの慣習です.

この改訂版では，変数 FACTOR の値をチェックし，値が負あるいは 0 の場合にはエラーメッセージを出力します．さらに，エラーが生じたことを示すため，AREA3 をあり得ない値である 0 にしています．Fortran の文字セットには「<」がないため,「以下もしくは等しい (less-than-or-equal)」を示すのに .LE. と記されることに注意してください．FACTOR が正の場合には先ほどと同じように計算が行われます.

2.8 配列

Fortran には配列 (array) を扱う十分な機能が備わっています．配列は 7 次元 (dimension) まで可能です．次のプログラム例 STATS では，データファイルから実数値を一次元配列に読み込み，平均値 (mean)

と標準偏差 (standard deviation) を計算します．$x_1, x_2, x_3, \ldots, x_N$ の値が与えられたとき，平均 M と標準偏差 S は次式で求められます．

$$M = \frac{\sum x_i}{N}$$
$$S^2 = \frac{\sum (x_i - M)^2}{N-1}$$

プログラム例では簡単にするために，ファイルに含まれるデータ数をプログラムに教える整数値がデータファイルの先頭に書かれているものとしています．

```
      PROGRAM STATS
      CHARACTER FNAME*50
      REAL X(1000)
      WRITE(UNIT=*, FMT=*) ' Enter data file name:'
      READ(UNIT=*, FMT='(A)') FNAME
      OPEN(UNIT=1, FILE=FNAME, STATUS='OLD')
*Read number of data points NPTS
*データ数 NPTS を読み込む
      READ(UNIT=1, FMT=*) NPTS
      WRITE(UNIT=*, FMT=*) NPTS, ' data points'
      IF(NPTS .GT. 1000) STOP 'Too many data points'
      READ(UNIT=1, FMT=*) (X(I), I = 1,NPTS)
      CALL MEANSD(X, NPTS, 1000, AVG, SD)
      WRITE(UNIT=*, FMT=*) 'Mean =', AVG, ' Std Deviation =', SD
      END

      SUBROUTINE MEANSD(X, NPTS, MPTS, AVG, SD)
      INTEGER NPTS, MPTS
      REAL X(MPTS), AVG, SD
      SUM = 0.0
      DO 15, I = 1,NPTS
          SUM = SUM + X(I)
   15 CONTINUE
      AVG = SUM / NPTS
      SUMSQ = 0.0
      DO 25, I = 1,NPTS
          SUMSQ = SUMSQ + (X(I) - AVG)**2
   25 CONTINUE
      SD = SQRT(SUMSQ /(NPTS-1))
      END
```

このプログラム例では新しい文がいくつか使われています．初めの CHARACTER 文では FNAME が 50 文字まで記憶できる変数であることを宣言しています．50 文字という値はどのオペレーティングシステムでもファイル名を表すのには十分でしょう．次の REAL 文で X(1) から X(1000) までの 1000 要素 (element) の実数変数 X を宣言しています．READ 文では，文字列 (character string) を読み込むための書式指定子 A が使われています．もともと，A は英数字 (alpha-numeric) を意味していました．

次の OPEN 文では入出力装置番号 (I/O unit number) 1 (このような小さい整数値がよく使われます)

をデータファイルに割り当てています．装置番号はこの後の入出力 (input/output) 文で使用されます．指定子の一つ，STATUS='OLD' はファイルが既存のものであることを指定しています．

次の IF 文は IF ブロックの文が一つだけであるときに用いられる特別な形式です．この例では，データに対して配列の大きさが不足しているときにプログラムを停止させるために使っています．

その次の READ 文では DO 並び (implied DO-loop) と呼ばれる形式を用いています．これは一つの文ですべてのデータを配列 X に順に読み込むものです．

関数の呼び出しが FUNCTION 文で定義された関数を呼び出すように，CALL 文は SUBROUTINE 文で定義されたサブルーチン副プログラムを呼び出します．先ほどの関数の例とこのサブルーチンの例の違いは，こちらの例では引数のうち X と NPTS はサブルーチンに関数の例と同じように値を引き渡していますが，引数 AVG と SD はサブルーチンからの戻り値を受け取っている点です．データの受け渡しの方向は，サブルーチンの中での仮引数 (dummy argument) の使い方だけで決めることができます．すなわち，引数はどちらの方向にも，あるいは，両方向にデータを受け渡すことができるのです．

INTEGER 文は前にも述べたように必須というわけではありませんが，全ての手続き (procedure) [6]の引数の型をはっきりと宣言しておくのはよいことです．

REAL 文は X が実数配列であることを宣言していますが，ここでは，別の引数 MPTS を配列の大きさの指定に用いて，整合配列 (adjustable array) として使う方法を用いています[7]．これは，配列が仮引数の場合にだけに許されるものです．一般に Fortran の配列の上下限 (bound) は定数で与えておかなければいけませんが，手続きの内部では，引き渡された配列は呼び出し側のプログラム単位ですでに記憶領域を確保してあるため，配列の大きさは仮引数を使って宣言することができます．

サブルーチンの残りの部分では，ループを使って要素の和を変数 SUM に，また要素の二乗和を SUMSQ に入れた後，公式に従って，平均と標準偏差を計算し，この値を出力する呼び出し側に値を返しています．

3 Fortran の実際

この章ではテキストとして書かれた Fortran プログラムを実行形式に変換する手順について説明します．主要な作業は Fortran のソースコード (source code) を適切な機械語 (machine code) に翻訳することです．ほとんどの Fortran システムではこの作業を二段階に分けて行います．この章では，この作業について詳しく説明します．

この章の説明は，この本のほかの部分の説明と 2 つの点で異なっています．1 つ目の点は，Fortran システムを使うためには，必ずしも Fortran システムがどのようにこの作業を行うかを知っている必要はないという点です．このことは，車を運転するのにエンジンがどのように動作しているかを知る必要がないのと同じことです．しかし，どちらの場合でも機械の動作について基本的な知識がある方がより簡単によりよい結果を得ることができます．

2 つ目の点は，この章の説明がほかの章と比べてより強くシステムに依存することです．Fortran 規格では実行されたときに Fortran のプログラムがどのように動作するのかが規定されているだけで，翻訳の手順については何も規定されていません．しかしながら，実際にはほとんどすべての Fortran システムが非常によく似た方法で動作しており，お使いのシステムとここで説明する「標準的な」システムとの違いはほとんどないはずです．ただし，残念ながら，それぞれのシステムの製造元が用いる名称の差違によって本質的な類似性が見えにくくなってしまっているのも事実です．

[6]訳注：手続きは関数副プログラムとサブルーチン副プログラムのこと．

[7]訳注：原著では実際のデータ数 NPTS を整合寸法に用いていますが，実際のデータ数を多次元配列の右端以外の添字の整合寸法に用いると致命的なバグになりえます．このためプログラム例では，別の仮引数 MPTS を加えた修正を行いました．

3.1 Fortran システム

プログラムを機械語に翻訳する方法は，インタプリタ (interpreter) を用いる方式とコンパイラ (compiler) を用いる方式の二つに大別することができます．インタプリタはプログラムの実行をすべて管理しています．インタプリタはソースコードを一行ずつ機械語に翻訳し，直ちにその行を実行します．そして次の行の翻訳に取りかかるということを繰り返していきます．もしエラーが起きた場合には，間違いを修正して，プログラムの停止した点から実行を再開することができます．これによってプログラムの開発を迅速に行うことができます．

この方式の問題点は，それほど簡単ではないすべてのプログラムがループや手続きの呼び出しなどの繰り返しの形式を含んでいることです．このような場合，ソースコードの同じ行を何度も繰り返し翻訳することになります．高度なインタプリタでは繰り返しの無駄な作業をすべて行うことは避けるようになってはいますが，無駄なオーバーヘッドを完全に回避するということはできません．

一方，コンパイラは全く異なった動作をします．コンパイラとはソースコードを機械語に一度に翻訳する独立したプログラムです．普通，機械語はファイルに保存されます．このファイルは実行ファイル (executable image) と呼ばれ，必要なときにはいつでもこのファイルだけで実行することができます．それぞれの文はただ一度だけ翻訳され，かつ，必要なだけ何度でも実行することができるので，翻訳に要する時間はあまり問題になりません．多くのシステムで，高度な処理を行って効率的な機械語を生成する最適化コンパイラと呼ばれるコンパイラを使うことができます．最適化されたコードは高速な内部のレジスタを可能な限り使用するようになっており，コンパイラはソースプログラムを一行ずつ処理するのではなく，ブロックごとに解析します．このようなことから，コンパイルされたプログラムはインタプリタで実行するよりも格段に高速に実行することができます．コンパイル方式の最大の問題はプログラムが正しく動作しないときに，ソースコードを修正し，プログラム全体を再コンパイルした後に，最初から再実行する必要があることです．また，ソースプログラムに含まれている英字名や行番号がコンパイル作業によって失われてしまうため，コンパイルされたプログラムの出すエラーメッセージがインタプリタのそれよりも情報量で劣る点もコンパイラ方式の欠点として挙げられます．

インタプリタは，より「ユーザフレンドリー」であり，高度にインタラクティブな利用や初心者が小規模なプログラムを作成・実行するような場合に特に適しています．このため，APL，Basic，Logo などの言語ではインタプリタ方式が用いられるのが普通です．一方，Fortran は，例えば天気の予測（この場合，計算時間がかかりすぎるとその結果に意味がなくなってしまいます）などの非常に長い計算時間が必要なジョブに多く用いられます．このため，コンパイラ方式の実行速度に関する利点は非常に重要な要素であり，ほとんどすべての Fortran システムがコンパイラ方式で翻訳するようになっています．

3.1.1 分割コンパイル

コンパイラ方式の最大の欠点は，どんなに小さな修正であろうとも修正後には必ずプログラム全体を再コンパイルする必要があることです．Fortran ではプログラム単位ごとに別々にコンパイルすることができるようにすることで，この問題をある程度軽減しています．このように別々にコンパイルされたプログラム単位あるいはモジュールは，最後に一つのファイルとしてリンクされます．

通常，コンパイラはソースコードをオブジェクトコードに変換します．オブジェクトコードとは機械語の命令で構成されてはいますが，記憶領域が絶対アドレスではなく相対アドレスで指定されているものです．すべてのプログラム単位を一緒にコンパイルすることも，それぞれ別にコンパイルすることもできます．どちらの方法によってもオブジェクトコードが生成されます．その次のステップはすべてのオブジェクトモジュールを一つのファイルにまとめ上げる作業です．この作業はリンク (linking) と呼ばれるのが普通ですが，loading，link-edit，task-building などと呼ばれることもあります．リン

カ (linker) の仕事はすべてのオブジェクトモジュールを一つにまとめ上げるとともに，それぞれのモジュールに絶対アドレスを割り当てて，完全な実行プログラム (executable program) を生成することです．実行プログラムは実行イメージ (executable image) とも呼ばれることがあります．

このように二段階にわけて処理することの利点は，一つのプログラム単位だけに修正を行った場合にはそのプログラム単位だけを再コンパイルすればよいということです．もちろん，最後に再リンクしてプログラムを生成し直す必要はあります．しかし，リンカのする仕事は比較的単純なものであるため，リンカの処理速度は高速であってしかるべきものです．残念ながらこのことはいつも当てはまるとは言えず，システムによっては小さなプログラムのリンクにコンパイルよりも長い処理時間を費やしてしまうこともあります．

3.2 ソースコードの作成

プログラムを書いた次のステップはコンピュータに入力することです．入力されたファイルをソースコードと呼びます．普通，Fortran システムにはそれ専用のエディタはついておらず，適当な使いやすいテキストエディタを用いて作成します[8]．

多くのテキストエディタには面倒な Fortran の文の入力の手間を軽減してくれるオプションが備わっています．Fortran の文用の欄 (statement field) の頭の 7 カラム目にキー一つで飛べるように定義できるものもあります（Fortran 規格に合致したソースファイルを作成するためには，タブ文字ではなく，適当な個数の空白文字を挿入するように定義しなければなりません）．さらに有用な機能は文用の欄 (statement field) の右端である 72 カラム目を超えた場合に警告を出す機能です．また，ほとんどのテキストエディタでは，両端が空白文字で区切られた単語を削除したり，挿入したりするのが簡単にできるようになっています．このため，Fortran の文の項目の両側に空白文字を入れておくと，後で編集するのが楽になります．また，このようにしておくとプログラムがより読みやすくなります．

ほとんどのプログラムは複数のプログラム単位から構成されています．プログラム単位をそれぞれ別のファイルに書いても，すべて一つのファイルにまとめて書いてもよいですし，任意の組み合わせで，分けて書いても構いません．はじめにプログラムを入力するときにはすべてを一つのファイルにまとめて書くようにした方が簡単に感じるかもしれませんが，プログラムの開発段階ではそれぞれのプログラム単位をそれぞれのファイルに書いておいた方が，単独で編集したり，コンパイルすることができるため作業がしやすくなります．それぞれのソースファイルの名前は，そのファイルに書かれた（最初の）プログラム単位の名前にしておくと，あまり混乱せずに済みます．

3.2.1 INCLUDE 文

多くのシステムで INCLUDE 文（システムによっては INSERT 文）と呼ばれる疑似命令 (pseudo-statement) を使用することができます．これは，INCLUDE 文で指定されたテキストファイルの内容を INCLUDE 文の書かれた位置に挿入するものです．この機能は，同じ内容の一群の文（宣言文の場合が多い）が複数のプログラム単位に現れる場合に特に便利です．このような例として，PARAMETER 文を用いて定数を定義する場合や，COMMON 文を用いて共有ブロックを定義する場合などが挙げられます．INCLUDE 文を用いることで，キー入力の手間とミスタイプの危険を低減することができます．INCLUDE 文は標準の文ではありませんが，コンパイラによって翻訳されるソースコード自体を変更するのでなく，単にソースファイルを操作するだけであるため，INCLUDE 文によって著しく可搬性が損なわれることはありません．

[8] Windows では「メモ帳」という簡易テキストエディタが付属しています．

3.3 コンパイル

Fortran コンパイラの機能は一つあるいは複数のソースファイルを読み込んで，対応するオブジェクトモジュールをオブジェクトファイルに書き出すことです．

ほとんどのコンパイラにはコンパイラの動作を指示したり，副次的な出力ファイルを生成するための多くのオプション (option) が備えられています．非常に有用で，かつ，多くのシステムで使用できるものについて次に説明します．

- ほとんどのコンパイラはリスティングファイル (listing file) を出力することができます．リスティングファイルとはソースコードをその行番号やエラーメッセージ，その他の有用な情報とともに出力したものです．多くの場合，それぞれのプログラム単位で用いられているすべての英字名と文番号のリストも出力されます．このリスト中の心当たりのないエントリはスペルミスの結果である可能性があるため，このリストをチェックしておく方がよいでしょう．

- さらに有用なリスティングは相互参照テーブル (cross-reference table) です．これは，それぞれの英字名が使っているメモリの位置を列挙したものです．良質のコンパイラは一度しか使われていない英字名があると警告を出しますが，これはプログラミングのミスの結果であることが多いためです．

- もう一つの広く使用可能なオプションは Fortran 規格に従っていない文を検出するもので，このオプションによって可搬性を維持することができます．

- 多くのシステムで最適化レベルを指定することができます．コンパイラの実行速度を向上させるための最適化は，プログラムの開発段階では最も低いレベルにしておくべきです．これによってエラーを検出する機能を向上させることができます．高度に最適化された実行ファイルは高速に実行することができますが，ソースコードの行が入れ替えられてしまうとエラーメッセージが役に立たなくなってしまいます．

- 多くのシステムで実行時エラーをチェックするコードを付加することができます．通常，配列や文字列の範囲を超えてアクセスした場合や桁あふれ (arithmetic over-flow) などを検出することができます．このようなエラーは頻繁に起きるので，このオプションは非常に便利です．プログラミングマニュアルには，この機能はプログラムの開発時のみに用い，実行速度が重要になるそれ以降では用いないことを勧めているものもあります．しかし，実行時のエラーチェックによって，実行速度がそれほど遅くなることはありません．

3.4 リンク

リンカ (linker) は，簡単に言うとコンパイラによって生成されたオブジェクトモジュール群を読み込んで，一つの実行ファイルにまとめ上げるものです．オブジェクトモジュール群中の一つのオブジェクトモジュールは必ず主プログラムを含むものでなくてはならず，そのほかのモジュールは手続きや初期値設定プログラム (block data subprogram) のものになります．

複数の異なったプログラムであっても，同じ計算が必要になることがよくあります．このような計算を手続き化しておき，それぞれのプログラムにリンクすることができれば，プログラミングの手間を大きく軽減することができます．このブロック化 (“building block”) は大きなプログラムに特に有用な手法です．多くの団体で，ソフトウェアの重要なリソースとなっている手続き (procedure) のコレクションを構築しています．すでに既存のアプリケーションで十分にテストされ，デバッグされているため，このようにして集められた手続きは，バグがなく信頼性が高くなっています．

3.4.1 オブジェクトライブラリ

様々なプログラムで共通に使われる手続きをオブジェクトモジュールとしてコンパイルされた形式で保存しておけば，コンパイルに要する時間を減らすことができます．ほとんどのオペレーティングシステムでは，オブジェクトモジュールをオブジェクトライブラリ（コンパイル済み (pre-compiled) コードライブラリ，再配置可能 (relocatable) コードライブラリと呼ばれることもあります）として保存しておくことができます．このファイルは，複数のオブジェクトモジュールを，簡単にオブジェクトモジュールを取り出せるようにインデックスとともに格納したものです．オブジェクトライブラリは効率が良いだけではなく，一つのファイル名をリンカに指定すればよいので，簡単に使用することができます．オブジェクトライブラリが指定されると，リンカは現在処理しているオブジェクトモジュールに含まれる CALL 文や関数の参照に相当するオブジェクトモジュールを探し出してリンクします．また，オブジェクトライブラリによって手続き群の管理が簡単になるのと同時に，必要なディスク容量を減らすこともできます．オブジェクトライブラリに含まれるオブジェクトモジュールのリストを出力したり，モジュールを削除したり，新しい版のモジュールと入れ替えたりする操作が簡単にできるようになっているのが普通です．

すべての Fortran システムは，SIN，COS，SQRT など，様々な組み込み関数のオブジェクトモジュールを格納したシステムライブラリとともに供給されています．このライブラリはリンカによって自動的に検索されるため，ユーザが指定する必要はありません．

また，Fortran のプログラムにリンクして使うことができるモジュールを格納したライブラリ形式の商用ソフトウェアも多く存在します．これらのライブラリによって統計学，信号処理 (signal processing)，グラフィックス，数値計算などの分野がカバーされています．

3.4.2 リンカオプション

主プログラムを最初に指定しなければならないシステムも中にはありますが，通常はオブジェクトモジュールをリンカに与える順序は任意です．しかし，ライブラリファイルを検索する順序は重要であるため，同時に複数の異なったライブラリを用いる場合には一定の注意を払う必要があります．

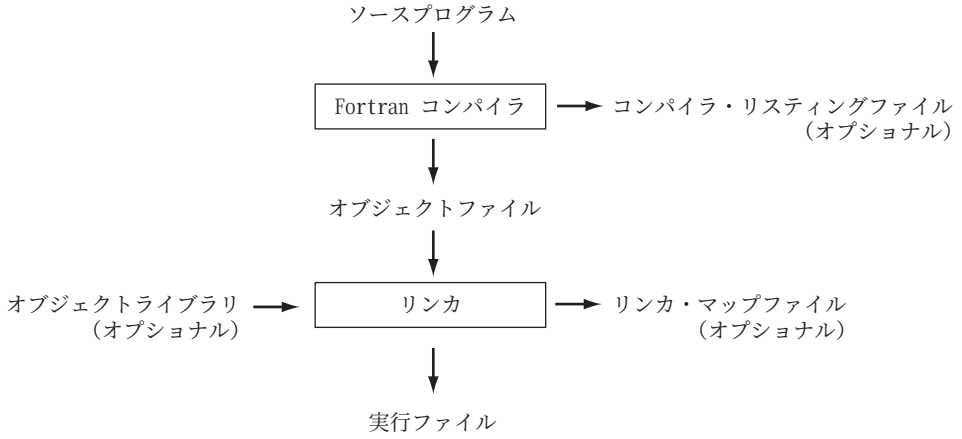
リンカの主要な出力は，普通，実行ファイル (executable image) と呼ばれる一つのファイルです．ほとんどのリンカはさまざまのモジュールのメモリ中の位置を示した，ストレージマップも出力することができます．この出力には，プログラムのデバッグに有用なシンボルテーブルなどの情報も含まれている場合があります．

3.5 プログラム開発

プログラム開発は次のような多くの段階を経て行われます．いくつかの段階は，最終的なプログラムが正しく動くようになるまで何度も繰り返し行われます．

1. プログラムの設計とソースコードの作成．
2. テキストファイルとして入力し，Fortran のソースファイル（群）を作成する．
3. ソースファイルをコンパイルし，オブジェクトモジュールを生成する．
4. オブジェクトモジュール（群）やオブジェクトライブラリ（群）をリンクして，実行ファイルを生成する．
5. 複数のテストデータを用いて実行ファイルを実行し，得られた結果をチェックする．

この過程の主要な部分を図に示すと次のようになります．



3.5.1 エラー処理

プログラム開発のほとんどすべての段階で，いろいろな原因で間違える可能性がありますが，そのほとんどはプログラマのミスによるものです．当然のことながら，プログラマの間違いのすべてを Fortran システムで検出することはできません．Fortran の文の文法間違いに対してはコンパイラがエラーを検出し，エラーの内容と可能ならばその位置の情報を含んだエラーメッセージを出力します．

他のエラーはリンクの段階で検出されます．例えば，サブルーチンや関数の名前をミスタイプすると，コンパイラはプログラム単位を 1 つずつ処理していくためこの間違いを検出することができず，リンカが「unsatisfied external reference （外部参照エラー）」のようなエラーメッセージを出力します．配列と関数の参照が同じ形式で書かれるため，配列名をミスタイプした時にもこの種のエラーメッセージが出力されることがあります．

実行時に生じるほとんどのエラーはプログラマのミスの結果，あるいは，少なくともエラーの生じる可能性を予見し損なったことによるものです．ゼロによる除算 (division by zero) や配列要素の宣言範囲を超えてアクセスするエラーなども，十分に注意してプログラミングすれば避けることができます．

しかし，いくら考えても回避できない種類の実行時エラーもあります．これらのエラーはほとんどすべて入出力に関連するものです．例えば，削除されたファイルをオープンしたり，データの壊れた入力ファイルから読み込もうとした場合などです．このため，ほとんどの入出力エラーは入出力文に `IOSTAT=`キーワードや `ERR=`キーワードを用いることで検出することができます．標準 Fortran では，そのほかの種類の文の実行時エラーを捕捉する方法は提供されていません．

コンパイル，リンク，実行時ともに明らかなエラーが生じなかった場合でも，すべてのバグが取り除かれたと考えるのは危険です．他にも，間違った結果を出すような，別の種類の間違いも存在します．プログラムが正しいと確認する唯一の方法は，できれば別の方法でも計算できるような，複数のテストデータを用いて試してみることです．プログラムが非常に複雑でその結果をあらかじめ知ることができない場合には，全体を部分に分けて，順にチェックを行っていくのがよいでしょう．

4 Fortran の文法と書き方

この章では Fortran プログラムの作成方法とその書き方のルールを説明します．Fortran のプログラムは複数のプログラム単位から成り立っています．また，それぞれのプログラム単位は Fortran の文 (statement) とコメント行から成り立っています．さらに，Fortran の文は Fortran のキーワードや英字名などで構成されています．またこれらの項目は文字 (character) で書かれています．

4.1 Fortran の文字セット

Fortran のプログラムを書く上で必要な，そして可搬的なプログラムを書くために使うべき文字は Fortran の文字セット (character set) として次のように定義されています.

英大文字 26 字　A B C ... X Y Z
数字 10 字　0 1 2 3 4 5 6 7 8 9
記号類 13 種

+	プラス (plus)	-	マイナス (minus)
*	アスタリスク (asterisk)	/	スラッシュ(slash)
	空白 (blank)	=	等号 (equals)
(	左括弧 (left parenthesis)	)	右括弧 (right parenthesis)
.	小数点 (decimal point)	,	カンマ (comma)
’	アポストフィー (apostrophe)	:	コロン (colon)
$	通貨記号 (currency symbol)		

この文字セットは多少制限されたものですが，広く使われているものであるため，可搬性の高いプログラムを書くことができるのです．一方で，例えば<, >などの演算子 (operator) を.LT. や.GT. と書かなければならないことや，アスタリスク (*) や括弧類は複数の目的で使われることなど，プログラムの可読性は犠牲になっています.

4.1.1 空白

空白文字（スペース）は，アポストフィーで囲まれた文字列定数 (character constant) 中以外の Fortran の文中では全て無視されます．Fortran の言語項目 (item) を空白で区切る必要はありませんが，適切に空白を入れておくと可読性が向上し，プログラムの編集が容易になります．ただし，一行が 72 カラムを超えてはいけないという制限があります.

4.1.2 通貨記号

通貨記号は決まった字形を持っていません．多くのシステムではドル記号$ですが，£ であるシステムもあります[9]．しかし，通貨記号は Fortran の文法では使われていないため，この違いはあまり問題にはなりません.

4.1.3 その他の文字

多くのコンピュータでは，英小文字，角括弧，アンパサンド (&)，パーセント記号など多くの印字可能な記号が含まれた文字セットを持っています．コンピュータで利用可能なこれらの文字は，コメント行や文字列定数の中で使うことができます[10].

Fortran の文字セットには，タブ (tab) や改行復帰 (carriage-return)，行送り (form-feed) などの印字制御 (carriage-control) 用の文字も含まれていませんが，書式付きの WRITE 文を使うことで，ページ番号を付けたり，表形式にした出力ファイルを作成することができます.

Fortran のプログラムではそのコンピュータでサポートされた文字データを処理することができます.

[9]訳注：半角の ¥は通貨記号には含まれておらず，特殊な意味があるため不用意に使うとエラーになります.

[10]訳注：g77+Windows では全角文字も同様に使用できます．ただし，2 バイト目が ¥と同じ文字コード (5C) の全角文字（「構十申貼能表暴予」など）では “Unknown escape sequence ‘¥’ followed by ...” というエラーになります．これを回避するには「構¥」のように該当する文字の直後に ¥を補ってください.

Fortran の規格は特定の文字コード (character code) の利用を前提とはしていませんが，American Standard Code for Information Interchange (ASCII)[11]の文字順を用いた文字比較関数 (character comparison function) を備えることが決められています．これについては 7.6 章で詳しく説明します．

4.2　文と行

文 (statement) は Fortran のプログラムの最小の単位で，他のプログラミング言語ではインストラクション (instruction) や命令 (command) と呼ばれるものに相当します．ほとんどの種類の文は，例えば READ，DO，ELSE IF，GO TO のように動作を意味する一語あるいは二語の英単語のキーワードで始まります．空白文字は無視されるため，二語のキーワードは例えば ELSEIF や ELSE IF のように一語でも二語でも書くことができます（ただし，後者の方が読みやすいでしょう）．

残念ながら，文の書き方の規則はパンチカードを使用していた頃の名残です．すべての文はそれぞれ行を改めて書いていかなければならず，行は次のように 3 つのフィールドに分かれています．

- 1〜5 カラム：文番号欄 (label field)
- 6 カラム：継続行マーク欄 (continuation marker field)
- 7〜72 カラム：文用の欄 (statement field)

文番号や継続行マークはそれほど多くの行で必要ではないため，始めの 6 カラム目の多くは空白のまま残ります．

73 カラム目以降は無視されます（以前 73〜80 カラムはパンチカードの番号欄として使われていたことがありました）．72 カラムの後の目に見えない境界は，これ以降の部分が切り捨てられても文法的には正しい可能性があり，この場合，コンパイラが間違いを検出できないため，致命的なバグの原因になるので注意が必要です．

4.2.1　継続行

文は一行に書かなければいけないわけではありません．複数行にわたって一つの文を書く場合，最初の行の 6 カラム目は空白でなければなりません．2 行目以降では 6 カラム目に空白（あるいは「0」）以外の文字を書いておきます．継続行は 19 行まで，すなわち全部で 20 行まで書くことができます．継続行のカラムの使い方の例を以下に示します．

```
columns
123456789...
      IF(REPLY .EQ.  'Y ' .OR. REPLY .EQ.  'y ' .OR.
     $ REPLY .EQ.  'T ' .OR. REPLY .EQ.  't ') THEN
```

ドル記号は継続行マークに適しています．それは，万一ミスで隣のカラムに書いてしまった場合にほとんどの場合コンパイル中にエラーとして検出されるためです．

END に関しては継続行の規則は適用されません．すなわち，END 文に継続行を続けることはできず，また，どの文も END だけから行を開始行に持つことはできないということです．どちらの規則も実用上問題にはなりません．

過度の継続行の使用によってプログラムが読みにくく，修正が難しくなるため，長い文はできるだけ短い文に分割するようにしましょう．

[11] 訳注：アスキーと読みます．

4.2.2 コメント行

コメント行はコンパイラに無視されるだけであるにも拘わらず，どのコンピュータプログラムにおいても重要なものです．その目的はプログラムを読んで，理解しなければならない人（例えば 6ヶ月後の，プログラムを書いた本人）の手助けをすることです．

Fortran のコメントは，1 カラム目にアスタリスク「*」を記した独立した行に書きます．例を挙げると，

```
*Calculate the atmospheric refraction at PRESS mbar.
*PRESS(mbar) における大気の屈折率の計算
      REF = PRESS * (0.1594 + 1.96E-2 * A + 2E-5 * A**2)
*Correct for the temperature T (Celsius)
*温度 T（摂氏）の補正
      TCOR = (273.0 + T) * (1.0 + 0.505 * A + 8.45E-2 * A**2)
```

コメントは END 文の後以外のプログラム単位のどの部分でも書くことができます．ただし，別のプログラム単位が続く場合は例外で，この場合，コメント行は後のプログラム単位の先頭の行になります．完全な空白行も可能で，これは空白のコメント行として扱われます．従って，プログラムの最後の END 文の後には空白行は許されないということになります．

連続するコメント行の数に制限はありません．また継続行の間にコメント行を入れることもできます．Fortran 規格に従うならば，コメント行は 72 カラムを超えて記すべきではありませんが，実際にこの規則が適用されるシステムはほとんどありません．

コメントには Fortran の文字セット以外の文字も用いることができます．(利用可能なシステムならば）小文字を主に用いることでコメント行をプログラムから目立たせることができます．また，文を説明するコメント行はその文の後よりも前に記した方がよいでしょう．

行末にコメントを許すシステムもあります．これは通常，感嘆符 (exclamation mark) をコメント部分の最初に記します．ただし，これは Fortran の規格で許されているものではありません．また，Fortran66 との互換性のため,「C」を 1 カラム目に記してコメント行とすることもできます．

4.2.3 文番号

文番号 (statement label) はどの文にも付けることができます．文番号を用いるには 3 つの理由があります．

- DO 文に用いた番号を文番号に用いて，その DO ループの終端を表す．
- FORMAT 文には必ず文番号を付けて，それを参照する READ や WRITE を示す．
- 任意の実行文 (executable statement) に文番号を付けて，例えば GO TO 文のような文によって，その行へ制御を移す（ジャンプする）ために用いる．

例えば，

```
*Read numbers from input file until it ends, add them up.
*入力ファイルから値を終わりまで読み込んで，すべてたし合わせる．
      SUM = 0.0
100   READ(UNIT=IN, FMT=200, END=9999) VALUE
200   FORMAT(F20.0)
      SUM = SUM + VALUE
      GO TO 100
9999  WRITE(UNIT=*, FMT=*)'SUM of values is ', SUM
```

文番号は 1 から 99999 までの符号なしの数字です．空白や，文番号に先行する 0 は無視されます．数値としての意味は全くなく，計算することもできません．文番号は文の最初の行の 1〜5 カラム目に書かなければならず，継続行では文番号欄は必ず空白になります．

一つのプログラム単位内では同じ文番号を用いることはできませんが，他のプログラム単位では完全に独立しています．どの文にも文番号を付けることができますが，FORMAT 文と実行文 (executable statement) に付けた文番号だけが意味を持ちます．そもそも，その他の文では文番号の使い道がありません．

ほとんどすべての文番号は，プログラム単位内の他点から制御を移す先を指示するために使われるため，文番号を多く用いるとプログラムを理解するのが困難になります．残念ながら現状では，Fortran で文番号を全く用いないことは不可能です．文番号を用いるときには，数字が増える順番で用い，10 あるいは 100 ずつ増やしていくのがよいでしょう．文番号は文番号欄に右詰にするべきではありません．

4.3 プログラム単位

完全な実行形式のプログラム (executable program) は一つあるいは複数のプログラム単位 (program unit) から成り立っています．この中には，PROGRAM 文で始まる主プログラム (main program unit) 常に一つ（そして一つだけ）あります．この他に，次の三種類の副プログラム単位 (subprogram unit) があります．

- サブルーチン副プログラム：これは SUBROUTINE 文で始まります．
- 関数副プログラム (function subprogram)，外部関数 (external function) とも呼ばれます：FUNCTION 文で始まります．
- 初期値設定副プログラム (block data subprogram)：BLOCK DATA 文で始まります．

サブルーチンと外部関数は外部手続き (external procedure) に分類されます．一方，初期値設定副プログラムは手続き (procedure) ではなく，名前付き共通ブロックの初期化だけに使われるものです．

すべてのプログラム単位は END 文で終端を示さなければなりません．

4.3.1 手続き

サブルーチンと外部関数は外部手続き (external procedure) に分類されます．これらについては 9 章で詳述します．手続き (procedure) とは，プログラムのどこからでも必要に応じて呼ぶ (call) ことができる，独立した一群の計算のことです．Fortran では SIN，COS，TAN，MIN，MAX などの多くの組込関数 (intrinsic function) が用意されています．これらの関数は必要なときには自分たちで準備することなく使用できる手続きです．引数がいくつあるかは場合によりますが，外部関数も関数名を媒介としてただ 1 つの値を返しますので，これらの関数と同じように使用することができます．

サブルーチンはもう少し一般的な形式の手続き (procedure) で，任意の個数の入力用の引数と出力用の引数を持つことができますが，その呼び出しは CALL 文で陽に行わなければなりません．手続きの中で別の手続きを呼び出すことは自由ですが，自分自身を呼び出すことは直接的，間接的を問わずできません．Fortran では再帰呼び出し (recursive call) はサポートされていないのです．

ほとんどの Fortran システムでは，一部の手続きを他の言語で書き，Fortran で書かれた他のモジュールとリンクして実行プログラムを生成することができます．もし用いる言語の手続きのインターフェースが Fortran のサブルーチンや関数のインターフェースと同じならば何の問題も生じません．

9 章で説明するように，プログラム単位間の情報の受け渡しは引数並びで行われるのが普通ですが，メモリを共有 (share) する共通ブロック (common block) を使うこともできます．これは多少手続きの独立性を損なう点があります．詳しくは 12 章で説明します．

4.4 文の種類と順序

Fortran の文は実行文と非実行文の二つに分類することができます．コンパイラは実行文 (executable statement) をそのまま機械語の命令 (machine code instruction) に翻訳します．一方，非実行文 (non-executable statement) は主にプログラムに関する情報をコンパイラに伝えるために用いられ，そのまま機械語に翻訳されるわけではありません．下表の END 文と右下段の枠内の全てが実行文，その他は非実行文です．

一つのプログラム単位の中での文の一般的な順序は次のようになります．

- プログラム単位のヘッダ (PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA 文)
- 宣言文 (specification statement)
- 実行文
- END 文

下表は文の順序に関するルールを示しています．それぞれの枠内の文は，同じ高さにある枠内の命令と順序を気にせずに書くことができます．例えば，PARAMETER 文は IMPLICIT 文と混ぜて使うことができます．一方，上下に分けられた枠内の文はそれぞれのプログラム単位にこの順で書いていかなければなりません（例えば，全ての文関数は実行文の前に書いておかなければいけません）．

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA</td></tr>
<tr><td rowspan="5">FORMAT</td><td rowspan="2">PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>型宣言文 (Type statements):
INTEGER, REAL, DOUBLE PRECISION,
COMPLEX, LOGICAL, CHARACTER
その他の宣言文 (Other specification statements):
COMMON, DIMENSION, EQUIVALENCE,
EXTERNAL, INTRINSIC, SAVE</td></tr>
<tr><td rowspan="2">DATA</td><td>文関数定義文 (Statement function statements)</td></tr>
<tr><td>実行文 (Executable statements):
BACKSPACE, CALL, CLOSE, CONTINUE, DO,
ELSE, ELSE IF, END IF, GO TO, IF,
INQUIRE, OPEN, READ, RETURN, REWIND,
STOP, WRITE, 代入文.</td></tr>
<tr><td colspan="3">END</td></tr>
</table>

4.4.1 実行順序

プログラムの実行は主プログラムの最初の実行文から開始されます．制御を別の行に移さない限り，プログラムは上から順に実行されていきます．これに対して，例えば IF 文や GO TO 文は同じプログラム単位の別の行へ，一方，CALL 文や関数の参照 (reference) では一時的に他の手続きへ制御を移します．

プログラムはいずれかのプログラム単位の STOP 文や主プログラムの END 文に行き着くまで，あるいは致命的な実行時エラー (fatal error) が発生するまで実行されていきます．プログラムが (STOP 文や END 文で）正常終了した場合にはオペレーティングシステムに制御を戻す前に Fortran システムは

オープンされたままのファイルをクローズします．しかし，プログラムの実行が途中で打ち切られてしまった場合にはファイル，特に出力ファイルに不完全あるいは壊れた記録が残ることがあります．

4.5 英字名

変数や配列，関数，サブルーチン，共通ブロックなどに与える名前が英字名 (symbolic name) です．全ての英字名は，名前の一文字目が英字，最大で 6 文字の英数字という単純な規則に合致していなければなりません．次は正しい英字名の例です．

```
I MATRIX VOLTS PIBY4 OLDCHI TWOX R2D2 OUTPUT
```

一方，次の例は規格に従っていない例です[12]．

`COMPLEX`	(長すぎる)
`MAX_EL`	(下線 (underscore) は許されていない)
`2PI`	(数字で始まっている)
`Height`	(小文字 (lower-case letter) は許されていない)

読み間違えることが多いので，英字名に数字を用いることは意味が明白な場合を除いて避けた方がよいでしょう．多くの装置の表示では，数字の `1` は英字の `I` と，同様に，`0` は英字の `O` と識別が困難なことが多いのです．

英字名の最大 6 文字の制限は最も不満の残る Fortran の規則の一つです．名前に暗号のような頭字語 (cryptic acronym) や省略語 (abbreviation) を用いると，プログラムを理解することがより難しくなってしまいますが，6 文字だけでは選択の余地はありません．多くのシステムでこの制限を強いることはありません（また，Fortran90 では 31 文字まで許されます）が，現状でソフトウェアの可搬性を維持するにはこの制限を厳密に守る必要があります．さらに，定数や変数，配列，関数の英字名には関連付けられた型 (data type) があることが問題になります．すなわち，型を陽に宣言しない限り，最初の文字で型が決められるため，さらに選択の余地がなくなってしまうのです．

4.5.1 英字名の有効範囲

共通ブロックと全ての種類のプログラム単位に付けられた英字名の有効範囲 (scope) は大域的（global）になります．すなわち，これらの名前は実行プログラム全体を通して固有の名前でなければなりません．その他のもの（変数，配列，文関数，組込関数，すべての種類の仮引数）に付けられた名前はそれが用いられているプログラム単位ごとの局所的 (local) なものになります．従って，他のプログラム単位では同じ名前を全く別のものとして使うことができます．

この規則の要点を理解するために簡単な例を挙げてみましょう．あるプログラムに `SUMMIT` という名前のサブルーチンが含まれているとします．これは大域名 (global name) ですので，同じ実行プログラムの中では別の外部手続きや共通ブロックなどの大域的要素 (global item) の名前（大域名）として用いることはできません．`SUMMIT` サブルーチンと，このサブルーチンをコールする他のプログラム単位の中では，変数や配列などの局所的要素 (local item) にこの名前を用いることはできません．しかしながら，このサブルーチンを間接的にコールするものを含む他のプログラム単位では，`SUMMIT` という名前を定数や変数，配列などの局所名として自由に用いることができます．

大域的要素に付けられる名前は，後から変更するのが大変であるため慎重に選ぶ必要があります．プログラム単位の名前を機械的に定めていくのでなければ，大きなプログラムを書いたり，手続きのライブラリを作成するときに名前が衝突してしまうのを避けることは難しいことなのです．手続きの名前に動詞のような名前を用いるのは理にかなっていると思います．理にかなった手続き名が浮かばない

[12] 訳注：g77 では `2PI` 以外は使用できます．

ときには，英語には例えば，DO，ASK，GET，PUT，TRY，EDIT，FORM，LIST，LOAD，SAVE，PLOT のように名前の素になる 3，4 字の動詞がたくさんあることを思い出してください．これらの単語に 1，2 文字加えれば，あらゆる手続き名を作り出すことができるのです．

4.5.2 予約語

ほとんどのコンピュータの言語では，システムによって予約されているために symbolic name として使うことのできない，多数の単語があるものです．例えば，Cobol 言語のプログラマは 500 近い予約語を覚えなければなりません．一方，Fortran には予約語がありません．例えば DATA，END，OPEN などは，長さも短く，完璧な英字名として使うことができます．ただし，このような使い方は規則に反するものではありませんが，多少混乱を引き起こすことになるでしょう．

組込関数の名前（例えば，SQRT，MIN，CHAR など）は厳密に言えば局所名 (local name) であり，ユーザは自由に定義して使うことができますが，これも一般的にはあまりよい方法とは言えません．たとえば，SQRT という名前を局所変数名に用いると，そのプログラム単位では平方根の計算ができなくなってしまいます．また，組込関数の名前を外部手続きの名前に用いると，その外部手続きを用いるすべてのプログラム単位でその名前を EXTERNAL 宣言しなければならず，これはさらに馬鹿げたやり方と言えるでしょう．

4.6 PROGRAM 文

PROGRAM 文は主プログラムの先頭だけに用いることができます．その機能はそれが書かれたプログラム単位の種類を示し，それに英字名を与えるだけです．この名前はプログラムのどこでも使うことができなくなってしまうのですが，実際には，Fortran システムがエラーメッセージの表示に使う程度の役割しかありません．その形式は次のように簡単なものです．

PROGRAM *name*

name は英字名です．この名前は通用範囲はグローバルであり，主プログラム内では局所名として，また，その他のプログラム単位の大域名としても使うことができません．Fortran66 との互換性のため，PROGRAM 文は省略可能になっています．このことによって次のような影響が生じます．もし，手続きの先頭に SUBROUTINE 文や FUNCTION 文を書き忘れると，コンパイラはそれを（名前なし）主プログラムと見なします．この場合，通常は主プログラムが二つあることになるため，リンカ (linker) が誤りを検出することになります．

4.7 END 文

END 文は全てのプログラム単位の最後の文でなければなりません．これは単に

```
END
```

とだけ書かれ，その後に継続行（やコメント）を加えてはいけません．END 文は実行文であり，文番号を付けることができます．もし，副プログラム単位（すなわち手続き）の END 文が実行された場合，呼び出した側のプログラム単位に制御を戻します．一方，主プログラムの END 文が実行された場合には，オープンされたままのファイルをクローズした後プログラムを停止し，オペレーティングシステムに制御を戻します．

5 定数，変数，配列

この章では定数 (constant) や変数 (variable)，配列 (array) などの Fortran のデータ記憶 (data storage) の要素について述べます．これらの要素には型 (data type) と呼ばれる重要な属性があります．型によって記憶される内容と演算の種類が決定されます．

5.1 型

ディジタルコンピュータで処理される情報はすべて，内部では二進数 (binary digit) あるいはビット (bit) の形式で内部に保持されています．数値や文字列 (string of character) など様々な種類の型を表現するために，複数のビットをまとめて使います．Fortran のプログラムを書く上では，データの型によって異なった内部表現が用いられていることを知っていればよく，内部表現そのものを知る必要はありません．型によってどのような演算を行うことができるかも決定されます．数値に対しては算術演算 (arithmetic operation)，文字列に対しては分割したり，連結したりする操作 (operation) が可能です．プログラムを書くときに決めた型は途中で変更することはできません．

数値計算に重点をおいた Fortran では数値データとして 4 つの型を持っています．算術式 (arithmetic expression) では異なった型を混ぜることができ，大抵の場合，自動で型変換 (type conversion) が行われます．一方，手続きの呼び出し (procedure call) などの場合には，自動の型変換が行われないため，厳密に型を合わせることが必要不可欠です．

算術型 (arithmetic data type) が表現可能な範囲と精度は規格では決められていません．以下には典型的な例を示しますが，正確な値はお使いの Fortran のマニュアルをご覧ください．

型を別の型に変換する組込関数が用意されています．数値を表す文字列から数値へ，あるいはその逆の変換は複雑ですので，内部ファイル (internal file) に `RAED` 文や `WRITE` 文を用いる方法 (10.3 節参照) によるのがよいでしょう．

今のところ，Fortran にはユーザ定義の型や構造体 (structure) はありません．

5.1.1 標準の型

下表に標準の Fortran でサポートされる 6 種の型の特徴をまとめました．

型	特徴
Integer （整数型）	整数．誤差なく正確に記憶される．
Real（実数型）	実数．小数部を持つ． 浮動小数点表現が用いられ，精度には制限がある．
Double Precision （倍精度実数型）	実数型と同様，ただし精度が高い．
Complex （複素数型）	複素数．実部と虚部を実数値の組で記憶する．
Logical （論理型）	ブール値 (Boolean value)．すなわち，真 (true) か偽 (false) のどちらか．
Character （文字型）	長さが固定された文字列 (string of character)．

最初の 4 つ (integer，real，double precision，complex) は数値を保持するもので，まとめて算術型（arithmetic data type）と呼ばれます．

5.1.2　整数型

整数型 (integer type) が表すことができるのは整数値だけですが，どのような場合でも誤差なく記憶することができます．整数型は配列の要素や文字列の文字数，ループの繰り返し回数など数えられるものを数えるのに頻繁に使われます．

整数型の値の範囲は使用しているシステムに依存します．多くのコンピュータは整数型に 32 ビット (1 ビットで正負，残りの 31 ビットで値の大きさを表す）使っており，扱える数値の範囲は-21, 4748, 3648 から+21, 4748, 3647 です．さらに大きな範囲を扱えるシステムもありますが，一方で非常に小さなシステムでは 16 ビットのものもあり，この範囲は-3, 2768 から+3, 2767 になります．

5.1.3　実数型

ほとんどの科学技術計算ではほかのどの型よりも実数型を多く使います．実数値は内部では浮動小数点表現 (floating-point representation) を用い，整数型よりも広い範囲を扱うことができますが，一般に値には誤差が含まれます．範囲と精度はシステムに依存します．

多くのコンピュータが 32 ビット以上を使って実数型を記憶しています．現在では多くのシステムが IEEE 標準表現 (IEEE standard representation) を用いており，32 ビットを用いた実数では有効桁 7 桁，範囲は大体 10^{-38} から 10^{+38} です．計算によっては，特に物理学や天文学の計算では 10^{40} を超えることもあり，この範囲では不十分である場合があります．科学技術用に特別に設計されたコンピュータ（普通，スーパーコンピュータと呼ばれる）では実数値に 64 ビットを割り当てており，精度の高い数値を扱えるとともに，範囲も拡大されています．このようなコンピュータでは倍精度実数型を使う必要はほとんどありません．

5.1.4　倍精度実数型

倍精度実数型 (double precision) も浮動小数点を用いる型です．Fortran 規格では実数型よりも高い精度を持つことだけが規定されていますが，実数型の倍の記憶単位 (storage unit) を使うため，プログラミングに際しては少なくとも倍の精度を持つと考えればよいでしょう．一方，値の範囲は実数型と同じという場合もあります．

倍精度実数型は実数型や整数型の倍の記憶領域を占めますが，計算に倍の時間を要するわけではありません．

5.1.5　複素数型

複素数型は一つの要素 (entity) で二つの実数値を記憶します．標準の Fortran では倍精度複素数型は規定されていません．

複素数は負値の平方根を求めると自然に必要になりますし，数学や物理学，工学の様々な分野で用いられています．複素数は $A + iB$ と表されます．ここで，A，B はそれぞれ実部と虚部，$i^2 = -1$ です．電気工学では i が電流を表すため，複素数は $A + jB$ と表します．

複素数の演算は複雑なものではありませんが，Fortran システムに任せた方が簡単です．コンピュータは複素数の演算の中間の結果を記憶しておく内部レジスタ (internal register) を使うことができるので，より効率的でもあります．複素数にはべき乗 (exponentiation) と四則演算が可能であり，また，複素数用の平方根関数や対数関数，三角関数などの組込関数が用意されています．

5.1.6 論理型

論理型 (logical type) は主に IF 文によって分岐 (conjunction) するときに用いられます．分岐とは，ある条件が真 (true) か偽 (false) によってプログラムの動作を選択するものです．論理型変数（あるいは論理型配列）は，このような論理値 (condition value) を後で用いるために記憶しておく目的に使われます．論理型変数は人の性別，ファイルのオープン・クローズ，電源のオン・オフなどの二値データを扱うのにも用いられます．

1 ワード（通常 32 ビット）にただ 1 ビットの情報を記憶させることの非効率を感じて，論理型変数や配列を使いたがらないプログラマもいます．しかし実際には，効率的にデータを押し込む手法を実装するには余計な命令が必要になるため，論理型変数を用いるよりも余計なメモリを使ってしまうのが普通です．

5.1.7 文字型

文字型 (character type) はそれぞれの要素に長さを定義する唯一のものです．この長さは変数が持つ文字の数のことです．一般に，要素が宣言されたときに長さが決められ，プログラムの実行中に変更することはできません．この例外は手続き (procedure) の仮引数 (dummy argument) の場合です．仮引数に対応する実引数 (actual argument) の長さを得る方法があるため，この方法を使うと，引数の文字列の長さに無関係な，汎用的な文字列操作の手続きを作成することができます．さらに文字の代入の規則では，長さが違うときに必要に応じて後ろを切ったり，空白を補ったりしてくれることになっています．すなわち，Fortran の文字型は本格的な可変長文字操作システムの特長を持つということができます．

文字型の最大の長さはシステムに依存します．すべてのシステムが 255 文字までは許していると考えるのが安全ですが，32767 文字まで（あるいはそれ以上）というシステムも多くあります．最小の長さは 1 文字で，空あるいはヌル (null) は許されていません．

5.1.8 記憶単位

Fortran ではそれぞれの型に割り当てるメモリの量を規定してはいないものの，相対的な量は規定されています．記憶単位 (storage unit) は重要な事項ではありませんが，書式なし直接アクセス記録 (unformatted direct-access record) を出力するときや COMMON 文と EQUIVALENCE 文を用いるときには知っていなければなりません．この規則は次のようなものです．

型	記憶単位
整数型, 実数型, 論理型	1 数値記憶単位 (1 *numerical* storage unit)
複素数型, 倍精度実数型	2 数値記憶単位 (2 *numerical* storage unit)
文字型 (CHARACTER*N)	N 文字記憶単位 (N *character* storage unit)

配列の記憶単位数は配列の全要素数との積になります．数値記憶単位と文字記憶単位の関係はシステムに依存するため定められていません．

5.1.9 ガイドライン

普通はプログラムの中でどの型を用いればよいかは明らかなものですが，算術型の中では判断基準のボーダーにあるようなものもあります．各区画内の発芽した種の数や一定時間間隔で検出される光子数などもともと整数であるものを処理する場合であっても，これらの値を記憶するのに整数型の配列を用いるか実数型の配列を用いるのかは明確に区別できるというわけではありません．この二つの

型は同じメモリ量を使いますが，コンピュータによっては整数型の方が浮動小数点の数値よりも加算，減算が速いものがあります．しかしながら実際には，この種の節約は通常続いて必要になる処理のための型変換によって消えてしまいます．整数型の持つ主要な問題点は値の範囲の制限です．ほとんど全てのコンピュータで浮動小数点のオーバーフローに対してエラーメッセージが出るのに対して，整数型のオーバーフローにはエラーメッセージの出ないものもあります．

コンピュータが 32 ビットの実数型を用いる場合の $1/10^7$ 程度の精度はプログラムによっては不十分であることがあり得ます．この不正確さは 100 万ポンドに対して数ペンスに，一日の長さに対して 10 ミリ秒に相当します．この誤差が大きな影響を及ぼすならば，倍精度実数型を使用することを考えなければなりません．倍精度実数型を使うことによってさらに誤差を少なくとも $1/10^7$ に減少することができます．異なった型の混用は間違いを犯す危険を増大させます．多少メモリと処理時間を多く使いますが，プログラムを通して倍精度実数型を使用する方が簡単で安全です．

算術式内の型に対しては自動的に型変換を行う機能がありますが，手続きの呼び出しなど他の場合には実引数は仮引数と同じ型であることが必要不可欠です．プログラム単位は独立にコンパイルされるため，コンパイラにとってもリンカにとっても外部手続きの呼び出しの際の型の違いを検出することが困難であるためです．

5.1.10　非標準の型

標準の Fortran は 6 種類の型を規定するだけですが，さらに多くの型が多くのシステムでサポートされています．`LOGICAL*1`，`INTEGER*2`，`REAL*8`，`COMPLEX*16` などの型を見かけることがあるかもしれません．アスタリクの後の数値はデータの記憶に使われるバイト数（1 バイトは 8 ビットをひとまとめにしたもの）を表しています．この表記法にはある種の合理性がありますが，完全に非標準の表記法です．`REAL*8` などの表記法を使用することは，それが単に `DOUBLE PRECISION` の同義語ならば全く無意味なことです．もちろん `COMPLEX*16` のような型が必要になる状況もありますが，この場合可搬性を失うという代償を払わなければならなくなります．

5.2　定数

定数はプログラムが書かれたときに決められた値に固定されます．定数の型は全てその形式から判断できます．算術型の定数は 10 進数で書かれ，8 進数や 16 進数はサポートされていません．

算術型の定数 (arithmetic constant) は一般に符号（正負）を持ちますが，符号を持たない定数が必要な状況もあります．定数がゼロであるときには符号は無視されます．

5.2.1　整定数

整定数 (integer constant) の一般的な形式は正負符号とそれに続く一つ以上の数字からなります．それ以外の文字（空白を除く）は使えません．もし値が正ならば正符号を省略することができます．正しい例を次に示します．

```
-100      42      0      +1048576
```

値が大きいときには 3 文字ずつ区切っておくと読みやすくなります．この目的にカンマ（国によってはピリオド）を普通用いますが，Fortran プログラム中では空白を用います（この方法はデータファイルの中では使えません）．

```
-1 000 000 000
```

この値は Fortran の規則に従って記されたものですが，コンピュータによってはこの値は大きすぎて記憶できないこともあることに注意してください．

5.2.2 実定数

実定数 (real constant) は整数と見分けがつくように小数点 (decimal point) か指数部 (exponent)（あるいは両方）がなければなりません．Fortran では E の文字が「掛ける 10 の何乗」を表します．例えば，1.234×10^{-5} は 1.234E-5 と書かれます．

実定数は一般的には次の形式になります．

sign *digits* . *digits* E *sign* *digits*
–integer-part– *–decimal-part–* *–exponent–*
—basic-real-constant— *—exponent-section—*

どちらの符号 (*sign*) も省略可能で，省略された場合正符号があるものと見なされます．整数部 (*integer-part*) と指数部に先行する 0 がある場合この 0 は無視されます．整数部あるいは小数部のどちらかは 0 ならば省略できますが，どちらか一方は必ずなければいけません．定数に小数点が書かれているならば，指数部分が 0 のときには指数部分全体 (*exponent section*) を省略することができます．

コンピュータが扱える以上の桁数の小数部を実定数（あるいは倍精度実定数）に与えても害はありません．値はコンピュータによって丸められ，余計な桁は無視されます．次は正しい実定数の例です．

```
.5        -10.       1E3       +123.456E4     .000001
```

小数点が定数の端にあると簡単に見落とされてしまいます（正しい記法ですが）．このため，定数を指数形式で書いて小数点の前に一つだけ数字を記す方法が慣例的に用いられています．この記法に従うと，先ほどの定数は次のように読みやすくなります．

```
0.5      -10.0     1000.0        1.23456E6      1.0E-6
```

5.2.3 倍精度実定数

倍精度実定数 (double precision constant) は実定数と似た形式を持ちますが，たとえ指数部が 0 であっても，E に換えて D を用いた指数部を必ず付けなければいけません．次は倍精度実定数の例です．

```
3.14159265358987D0        1.0D-12        -3.652564D+02
```

5.2.4 複素定数

複素定数 (complex constant) は丸括弧の中に二つの実定数あるいは整数をカンマで区切って並べたものです．最初の数値が実部，後が虚部を表します．次は複素定数の例です．

```
(3.14,-5.67)      (+1E5,0.125)      (0,0)      (-0.999,2.718E15)
```

5.2.5 論理定数

論理定数 (logical constant) の値は 2 つだけであり，.TRUE. と .FALSE. で表されます．両側の点はこの特殊な形式と，英字名として使われ得る単語 TRUE，FALSE とを区別するために必要です．

5.2.6　文字定数

文字定数 (character constant) は引用符（アポストロフィー）で囲まれた文字列で構成されます．引用符で囲まれた文字列 (string) にはコンピュータでサポートされた文字セットをすべて使うことができます．文字定数の中の空白文字は無視されず，その他の文字と同じように一つの文字として扱われます．次は文字定数の例です．

```
'X'
'$40 + 15%'
'This is a constant including spaces'
```

アポストフィー は 2 つ続けて（間に空白を入れずに）書くことで文字列の中に含めることができます．続けて書かれた 2 つのアポストフィーは文字列の長さの計算では 1 文字として扱われます．例えば，`'DON''T'` は 5 文字の文字定数になります．

5.3　型宣言

上に規則によって，文字として打ち込まれた定数はその形からその型を決めることができます．同様に，式 (expression) の型も式中に使われているオペランド (被演算数・operand)[13] と演算子 (operator) から決められます．組込関数もまたその型などを始めとした属性についてコンパイラがあらかじめ知っている特別なケースです．それ以外のオブジェクトはすべての英字名で参照されます．本節で説明する規則は全ての名前の付けられたオブジェクト（変数，配列，名前付き定数，文関数，外部手続き）に適用されます．

多くのプログラミング言語，特に Algol 系の言語では，プログラムで使われているほとんどすべてのオブジェクトの型を陽に示さなければなりません．多くのプログラマはこのような型宣言を億劫な作業ととらえていますが，しかし，このような型宣言によってコンパイラがプログラムの間違いを検出するのが容易になっているのです．

Fortran でも同様な型宣言文 (type statement) を用いて陽に型を指定することができますが，ディフォルトの型を設けることで手間を減らすことができるようになっています（暗黙の型宣言・default implied typing）．型宣言文で宣言されていないあらゆるオブジェクトの型は名前の最初の文字で決められます．このルールを次表に示します．

名前の最初の文字	暗黙の型
A〜H	実数型
I〜N	整数型
O〜Z	実数型

ほとんどのプログラムでは整数型と実数型のオブジェクトを特によく使いますので，英字名に適切な頭文字を使うと，このディフォルト値によって必要な型宣言文の数を減らすことができます．

この一文字目の規則は，次に説明するように，IMPLICIT 文を用いることでプログラムを通して一律に変更することもできます．

5.3.1　型宣言文

それぞれの型に対応した 6 種類の型宣言文 (type statement) があります．それは非常に単純な形式で，単に宣言したい型のキーワードとそれに続く英字名の並びを記したものです．例をあげると，

[13] 訳注：演算の対象となる値のこと．

```
INTEGER AGE, GRADE
LOGICAL SUPER
REAL RATE, HOURS, PAY, TAX, INSURE
```

この例では，実数型の型宣言文の始めの 4 つの変数は暗黙の型宣言に任せてしまっても実数型の変数になります．このように確認の意味で型宣言しても害はありません．

型宣言文の数に制限はありませんが，1 つのプログラム単位内で 1 つの英字名を 2 度以上陽に型宣言することはできません．型宣言文はすべての実行文の前に置かなければなりません．また，これは必ずというわけではありませんが，型宣言文は同じ英字名を参照する他の宣言文よりも前に置いた方がよいでしょう．型宣言文は副プログラム内で仮引数の型宣言，また，外部関数では関数の型宣言にも使います．組込関数に対して型宣言をしても何の効果もありませんので，このような使い方は避けましょう[14]．

文字型の型宣言文 (character statement) は長さ，すなわち記憶する文字数も併せて宣言するため，他の型と少し違いがあります．長さは，次の例のように，それぞれの名前に分けて宣言することができます．

```
CHARACTER NAME*15, STREET*30, TOWN*20, PCODE*7
```

また，いくつかの変数が同じ長さである場合には，型のディフォルトの長さとして始めに宣言することができます．

```
CHARACTER*20 STAR, GALAXY, COMET*4, PLANET
```

この宣言によって，COMET が 4 文字，STAR，GALAXY，PLANET が 20 文字の長さになります．長さを指定しない場合はディフォルトである 1 文字の長さになります．長さは丸括弧内に名前付き整定数 (named integer constant) あるいは整定数式 (integer constant expression) を記して，宣言することもできます．例えば，

```
PARAMETER (NEXT=15, LAST=80)
CHARACTER TEXT*(NEXT+LAST)
```

コンパイル時に文字変数の長さが確定していなければならないことに注意してください．次の特殊な形式

```
CHARACTER NAME*(*)
```

は次の二つの場合に可能です．一つは名前付き定数 (named constant) に対して PARAMETER 文中に記された文字列から長さがわかる場合（5.4 節），もう一つは，対応する実引数の長さが決まっている手続きの仮引数に対して用いる場合 (9.5 節）です．型宣言文は配列の大きさを宣言するのにも使うことができます．これについては 5.6 節で説明します．

5.3.2 IMPLICIT 文

IMPLICIT 文は暗黙の型宣言の規則をプログラム単位内で一律に変更するために使います．例えば

```
IMPLICIT DOUBLE PRECISION (D,X-Z), INTEGER (I-P)
```

は，D，X，Y，Z で始まるすべての英字名（型宣言文で宣言されたもの除く）を倍精度実数型にします．同様に I〜N ではなく，I〜P までが整数型になります．その他の文字 (A〜C，E〜H，Q〜W）は実数型のままで変わりません．

IMPLICIT 文は，例えば

[14] 訳注：これは総称名の組込関数に対しての話で，個別名の組込関数の型と違う型で型宣言するとエラーになります．

```
IMPLICIT CHARACTER*100 (C,Z), CHARACTER*4 (S)
```

のように，文字型のディフォルトの長さを指定することもできます．ただし，これはあまり実用的ではありません．ディフォルトの長さは型宣言文と同じ 1 です．

一つのプログラム単位内で IMPLICIT 文を複数使うことができますが，一つの文字の型の指定は一度だけでなければなりません．IMPLICIT 文で型の指定がなされない文字は Fortran の通常の暗黙の型宣言の規則に従います．それぞれの型名の後に続く文字の並びはアルファベット順でなければなりません．IMPLICIT 文はその他の宣言文の前に置かなければなりません．ただしこれには一つ例外があります．PARAMETER 文の中で定義された名前付き定数が IMPLICIT 文の影響を受けない場合には，この PARAMETER 文をその IMPLICIT 文の前に置くことができます．一方，仮引数や関数名はそれ以降に宣言された IMPLICIT 文の影響を受けます．IMPLICIT 文は組込関数名には影響を及ぼしません．

5.3.3 ガイドライン

型宣言に関してはまったく反対の考え方をする二つの流儀があります．一つはすべての名前に対して陽に型宣言すべきであるとするものです．これによって，それぞれの名前についてより注意深く検討するようになるため，確かにミスを避ける効果があります．特に，もしすべての名前があらかじめ宣言されるものと仮定できるならば，コンパイラがミスを見つけ出す作業はより容易になります．Fortran コンパイラの中には，ディフォルトの型をなくし，その結果として，すべての名前の型宣言が必要になる IMPLICIT NONE の形式の使用が許されているものがあります．他にもコンパイルスイッチで同様の働きをするものもあります．もしお使いのコンパイラがこの機能を持っていなくても，例えば次のような宣言をプログラム単位の始めの方に入れておくと陽に型宣言をしない限りほとんどエラーになるため，同様の効果を得ることができます．

```
IMPLICIT CHARACTER*1000000 (A-Z)
```

すべての名前の型をあらかじめ宣言する方法の欠点は，多くの型宣言文でプログラムがこみいってしまい，プログラムの構造やアルゴリズムを見えにくくしてしまうことがある点です．

もう一つの方法は暗黙の型宣言を最大限使って，文の数を少なくするやりかたです．当然一文字目は型に合うように選ばなければなりませんので，自由に使える文字は 5 文字までになります．このため，意味のある英字名を付けるのがより難しくなってしまいます．このため，Fortran のプログラムには RIMAGE，ISIZE，KOUNT のような名前がよく使われています．なお，様々な長さで使われる文字型に対しては，型宣言が依然として必要であることは明らかです．

経験からいうと，一貫して使われるならばどちらの方法でも満足できる結果が得られます．しかし，IMPLICIT 文を使って一文字目の型を大規模に変更してしまうのは一般にミスを犯す原因となります．IMPLICIT 文をもし使うならば，下の例のように，あまり使わない 1〜2 文字の型をあまり一般的でない型に変更するのにとどめた方がよいでしょう．

```
IMPLICIT DOUBLE PRECISION (Z), LOGICAL (Q),
COMPLEX (X)
```

また，手続きの呼び出し時の型の不一致を引き起こす原因になるので，それぞれのプログラム単位で同じ IMPLICIT 文を使う方が賢明です．

5.4 名前付き定数

PARAMETER 文は定数 (constant) に定数名（symbolic name of constant) を付けるのに使います．これは様々な状況で便利に使えます．

自然の定数（π など）や物理的な変換定数 (キログラムとポンドの比など）などは，実際の値をプログラムに一度だけ入力し，その名前をプログラム中で使うと，タイプの手間とタイプミスの危険を低減することができます.

```
REAL PI, TWOPI, HALFPI, RTOD
PARAMETER (PI = 3.14159265, TWOPI = 2.0 * PI)
PARAMETER (HALFPI = PI / 2.0, RTOD = 180.0 / PI)
```

このように宣言しておくと，プログラム中の定数の位置に PI，TWOPI などの名前を用いることができます．不注意な書き換えに対する保護機能があるため，変数を用いるよりも定数名を用いる方が安全です．定数には書き換えに対するハードウェア的な保護機能があるのが普通なのです．また，定数名を用いる方が定数を直接書くよりも多少プログラムが読みやすくなります．HALFPI という名前の意味を考えるよりも，1.570796325 という数字の意味を読みとる方が難しいでしょう.

もう一つの重要な使い方は，プログラムの定数であっても，常に固定された値ではない数値，すなわち，現在は定まっていますが，将来変更することがあり得る数値に使うことです．例えば下に示す例のように，名前付き定数は配列の大きさや文字列の長さを表す数値などによく使われます.

```
INTEGER MAXR, MAXC, NPTS
PARAMETER (MAXR = 100, MAXC = 500, NPTS = MAXR*MAXC)
REAL MATRIX(MAXR,MAXC), COLUMN(MAXR), ROW(MAXC)
```

配列の添字 (subscript) が宣言されている範囲にあるかチェックする場合（下の例）など，MAXR や MAXC などの定数をプログラムの実行文中で使うこともできます.

```
IF(NCOL .GT. MAXC .OR. NROW .GT. MAXR) THEN
    STOP  'Matrix is too small '
ELSE
    MATRIX(NROW,NCOL) = ROW(NCOL)
END IF
```

もし，宣言した配列が必要な計算に対して小さ過ぎるのが判明した場合には，単にこの PARAMETER 文の値を変えればよいのです．その後，コンパイルし直せばすべて自動的に変更されます.

文字型に関しては代入に関する規則が PARAMETER 文にも適用されます．これについては 7.4 節で述べますが，さらに，文字列の特殊な長さ指定 *(*) を使うことができます．これは実際に書かれた文字列定数の長さを定数の長さとして指定するものです．型宣言は定数を割り当てる PARAMETER 文の前に置かなければいけませんので，次のように使います.

```
CHARACTER*(*) LETTER, DIGIT, ALPNUM
PARAMETER (LETTER =  'ABCDEFGHIJKLMNOPQRSTUVWXYZ ',
$ DIGIT =  '0123456789 ', ALPNUM = LETTER // DIGIT)
CHARACTER WARN*(*)
PARAMETER (WARN =  'This matrix is nearly singular ')
```

上の例では定数 ALPNUM は 36 字の長さになり，A〜Z の英字と 0〜9 の数字を格納します[15].

名前付き論理定数 (named logical constant) もありますが，あまり実用的な例は思いつきません.

```
PARAMETER (NX = 100, NY = 200, NZ = 300, NTOT = NX*NY*NZ)
LOGICAL LARGE
PARAMETER (LARGE = (NTOT .GT. 1000000) .OR. (NZ .GT. 1000))
```

[15]訳注：//は左の文字列 (LETTER) に右の文字列 (DIGIT) を連結する演算子です.

5.4.1 PARAMETER 文

PARAMETER 文の一般的な形式として次のように書くことができます.

PARAMETER （ *cname* = *cexp* , *cname* = *cexp* , ... ）

ここで，*cname* は定数名となる英字名，*cexp* は適切な型の定数式 (constant expression) です.

定数式の各項は定数，あるいは同じプログラム単位の中であらかじめ定義されている名前付き定数でなければならず，変数や配列要素，関数の参照は許されません．式に対するその他の規則はそのまま適用されます．式中に丸括弧を使うこともできますし，異なった型を混在させることもできます．ただし，べき乗に関しては指数部分に整数型しか使えないという制限があります．代入文に対する通常の規則も適用されます．算術型に対しては必要に応じて適当な型変換が行われ，文字型に対しては必要な長さに打ち切られたり，空白文字が補われたりします．ただし，文字定数式 (character constant expression) 中で部分列 (substring) の参照は許されていませんので注意してください.

PARAMETER 文は宣言文であり，型宣言文の前あるいは後に置かなければなりません．しかし，名前付き定数の型や文字列の長さに影響する型宣言文（あるいは IMPLICIT 文）はその定数が定義されている PARAMETER 文の前に置かなければなりません．この規則に従う限り，PARAMETER 文を IMPLICIT 文の前に置くことができます．次の例のように，この規則によって，先頭文字が一定の範囲にある文字型定数のディフォルトの長さを名前付き定数として定義することができます.

```
PROGRAM CLEVER
PARAMETER (LENCD = 40, LENE = 2 * LENCD)
IMPLICIT CHARACTER*(LENCD)(C-D), CHARACTER*(LENE)(E)
PARAMETER (DEMO =  'This is exactly 40 chars long ')
```

5.4.2 ガイドライン

現在の Fortran 規格の一つの制限は動的なメモリの割り当て (dynamic memory allocation) ができない点です．この制限によって生じる問題を避ける最良の方法は配列の添字の上下限を名前付き定数で与えることです．この方法によって，場面に応じてプログラムを非常に簡単に変更することができます.

プログラム中で必要になる物理定数や数学定数も名前付き定数として与えておくべきです．もし複数のプログラム単位で同じ名前付き定数が必要になるならば，必要になる PARAMETER 文を一つのファイルに集めおき，それらが必要になるプログラム単位中に INCLUDE 文を使って読み込ませるようにするのがよいでしょう.

PARAMETER 文で倍精度実定数を定義するときには，定数の指数部を表す文字に D を使うのを忘れないようにしましょう.

Fortran では定数の配列は使えません．この制限に対しては，通常の配列として型宣言文で定義し，DATA 文 (11 章で説明します）を使って初期化するしか方法はありません.

5.5 変数

変数は単に型が固定された，名前の付けられたメモリ位置に過ぎません．すでに説明したように，変数名の一文字目で決定される暗黙の型で使うならば，あらかじめ型宣言をしておく必要はありません．それ以外の場合には型宣言が必要です.

プログラムの実行の開始時には，その変数の使われるプログラム単位内の DATA 文（11 章参照）で初期化しない限り，変数の値は不定です．値が不定の変数は式の中に現れてはなりません．手続きが再度呼び出されたときには，SAVE 文で宣言されない限り，以前に呼ばれたときの手続き内の局所変数の値が保存される保証はありません.

5.6 配列

配列は単一の名前の付されたひとかたまりのメモリ領域です．配列の要素 (element) はすべて同一の型を持ちます．

数学では配列の要素は a_1, a_2, a_3 のように表されます．Fortran では配列要素は A(1)，A(2)，A(3) のように，配列名直後の丸括弧内の添字式 (subscript expression) で識別されます．添字は整数型でなければなりませんが，その値を指定するには，関数の呼び出しも含めてどのような複雑な式でも構いません．

配列要素はほとんど全ての実行文中で変数と同じように使うことができます．配列要素は非常に多くの場合ループの中で使われます．典型的な例は，整数のループカウンタが順番に配列要素を選んでいくものです．

```
*Add array OLD to array NEW making array TOTAL
*配列 OLD と NEW をたして TOTAL に代入
      PARAMETER (NDATA = 1024)
      REAL OLD(NDATA), NEW(NDATA), TOTAL(NDATA)
*......
      DO 100, I = 1,NDATA
          TOTAL(I) = OLD(I) + NEW(I)
100   CONTINUE
```

5.6.1 配列の宣言

配列は最大で 7 次元 (dimension) まで許されます．それぞれの次元の添字の下限 (lower bound) は宣言されない限り 1 になります．添字の上限は，下限以上ならばいくらでも大きくとることができます．手続きの仮引数である配列は，その添字の範囲を同様に仮引数として与えられた整数型変数で指定することができます．それ以外の場合には，添字の範囲は整数型の定数式で与える必要があります．コンパイル時にこの配列の大きさが定められます．

型宣言文，DIMENSION 文，COMMON 文はどれも配列の宣言に使うことができますが，COMMON 文は特殊な目的のためのものです（12 章で説明します）．DIMENSION 文は型宣言文と似た形式を持ちますが，型は宣言せず，配列の添字の上下限を宣言するだけのものです．普通は次のように，型と添字の範囲を型宣言文で一度に宣言してしまう方が簡単できれいです．

```
      CHARACTER COLUMN(5)*25, TITLE*80
```

文字型の配列を宣言する場合には，配列の上下限の指定の後に文字列の長さを記します．上の例では，文字型配列 COLUMN は長さ 25 文字の要素 5 つを持ちます．TITLE は，すでに説明したように単に 80 文字の長さを持つ文字型変数です．文字列のディフォルトの長さを型宣言文で設定することはできますが，同じようにディフォルトの配列の大きさを設定することはできません．

一般に，後で変更することが容易になるため，配列の上下限の指定には名前付き定数を使うのがよいでしょう．

```
      PARAMETER (MAXIM = 15)
      INTEGER POINTS(MAXIM)
      COMPLEX SERIES(2**MAXIM)
```

これらの配列の添字の下限はすべて 1 です．任意の次元の下限値を 1 以外にするには，下の例のように，下限値と上限値をコロン「:」で区切ります．

```
REAL TAX(1985:1990), PAY(12,1985:1990)
LOGICAL TRIPLE(-1:1, -1:1, -1:1, -1:1)
```

TAX は TAX(1985) から TAX(1990) の 6 要素，PAY は PAY(1,1985) から PAY(12,1990) の 72 要素，TRIPLE は TRIPLE(-1,-1,-1,-1) から TRIPLE(1,1,1,1) の 81 要素を持ちます.

Fortran 自体には配列の大きさに制限はありませんが，ハードウェアの容量による制限が生じます. 仮想メモリ (virtual memory) をサポートしたオペレーティングシステムでは，物理メモリの容量よりも大きな配列を使うことができます．配列のアクセスされていない部分はディスク上のファイルなどの外部記憶装置に保持されます.

5.6.2 配列の使用

配列要素の参照には，宣言された次元の数と同じ個数の添字を使います．それぞれの添字は整数型ならばどのような複雑な式でも構いませんが，副作用 (side-effect) のある関数（9.3 参照）を使うときには制限があります.

配列要素の参照は，すべての添字式が確定値を有し，かつ，すべてが宣言された範囲にある場合のみ，意味のある結果が得られます．配列要素はその値が確定しているときにのみ，式中で使うことができます．配列全体，あるいは配列の一部の要素は DATA 文 (12 章参照）で初期化することができます.

配列は次の場合には添字を付けずに使うことができます.

- 型宣言文や DIMENSION 文，SAVE 文などの宣言文の中.
- 関数や CALL 文の引数の中（このとき配列全体が対応する仮引数に渡されます．仮引数は引き渡しが可能な正しい配列宣言がなされていなければなりません）.
- READ 文や WRITE 文の入出力並び (data transfer list・input/output list) の中で.（配列全体に入出力されます．疑似寸法仮配列 (assumed size dummy argument array) [16]には許されていません.）
- READ 文や WRITE 文の装置指定子 (unit identifier) として（文字型配列のそれぞれの要素がそれぞれの記録 (record) の内部ファイルになります）.
- READ 文や WRITE 文の書式指定子 (format identifier) として（文字型の配列要素に書式識別子[17]を入れておくと，書式識別子として順に使われていきます）.

5.6.3 記憶の順序

配列は常に連続したメモリに記憶されます．多次元配列では要素の順序は，最初の添字が最も速く変化するように，二番目の添字はその次に速く変化するように配置されます．たとえば，次のような二次元配列 (2-dimensional array) では,

$$X(2,3) = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \end{bmatrix}$$

次の順序でメモリに記憶されます（簡単のため全部で 6 要素にしています）.

```
X(1,1), X(2,1), X(1,2), X(2,2), X(1,3), X(2,3)
```

[16]訳注：仮引数の配列で，大きさが変数で与えられているもの
[17]訳注：簡単に言うと format 文の括弧の中に書く識別子のこと.

すなわち，一列目から順に下へ行き，その後次の列に移るように記憶されます．この列ごとに記憶順は，他の言語の記憶順と異なっていることがあります[18]．

大規模な多次元配列を用い，その配列のすべての要素に演算を行う場合には，要素の記憶順は重要です．それは記憶順，すなわち，行ではなく列ごとにアクセスする方が処理が速くなるためです．従って，次のように最後の添字を指すループ変数を最も外側のループに用い，順に内側のループにしていく方が処理が速くなります．

```
      DOUBLE PRECISION ARRAY(100,100), SUM
      SUM = 0.0D0
      DO 250,L = 1,100
          DO 150,K = 1,100
              SUM = SUM + ARRAY(K,L)
150       CONTINUE
250   CONTINUE
```

このようにループを構成すると連続したメモリを順にアクセスするため，プロセッサ (processor) の添字の計算の負担を軽減することができます．

6 算術式

Fortran は数値計算に関して十分な機能を持っています．算術式 (arithmetic expression) と代入文 (assignment statement) はその式中に整数，実数，倍精度実数，複素数を使うことができます．必要に応じて自動的に型変換が行われますし，型変換は組込関数を用いて陽に行うこともできます．その他にも，三角関数や対数関数，その他便利な関数が整備されています．

たとえば，余弦定理を用いると三角形の二辺の長さとその間の角度から，もう一辺の長さが次式で計算できます．

$$\sqrt{b^2 + c^2 - 2bc\cos A}$$

これを Fortran の式として記すと次のようになります．

```
SQRT(B**2 + C**2 - 2.0 * B * C * COS(ANGLEA))
```

式中には組込関数 SQRT と COS が使われています．SQRT 関数は X**0.5 と同じ結果になりますが，内部で EXP 関数と LOG 関数を用いる指数関数として 1/2 乗を求めるよりも，平方根関数を用いた方が簡単で，計算も速く，さらに多くの場合正確な結果が得られます．

代入文は（右辺の）式の値を評価し，(左辺の）変数や配列要素にその値を代入します．代入文はその他のほとんどの Fortran の文とは異なり，例えば次の例のように文の先頭にキーワードがありません．

```
A = SQRT(B**2 + C**2 - 2.0 * B * C * COS(ANGLEA))
TOTAL(N/2+1) = 0.0
FLUX = FLUX + 1.0
```

6.1 算術式

式 (expression) の最も簡単な形式はただ一つの定数や変数などのオペランドを持つものです．必要な計算を行うようにさまざまなオペランドと演算子を組み合わせて，次の例のような，より複雑な式を構成することができます．

[18] 訳注：たとえば，C 言語では右端の添字が最も速く変化するように記憶します．

```
RATE * HOURS + BONUS
```

Fortran の規則，特に式の評価順序は数学の規則に可能な限り従うように設計されています．上の例で，先に書いてあるからではなく高い優先度を持っているために，乗算は加算よりも常に先に実行されます．優先順位に自信がない場合や変更するときには，次の例のように丸括弧を使うことができます．

```
(ROOM + DINNER) * 1.15
```

丸括弧で囲まれた式 (sub-expression) は常に先に評価されます．丸括弧は必要なだけ入れ子 (nest) にすることができます．余分な丸括弧を加えても問題はありませんので，計算順に自信がないときには自由に括弧を加えて計算順を指示したり，複雑な式を読みやすくすることができます．

算術式には+ - * / **の五種類の算術演算子 (arithmetical operator) を使うことができます．二重のアスタリスクは指数を表します．従って，次の数式

$$(1 + \mathrm{RATE}/100)^{\mathrm{years}}$$

は Fortran では次のように表すことができます．

```
(1.0 + RATE/100.0)**YEARS
```

定数値に小数値 (.0) を付しているのは実数型にするためです．

算術式には異なった型のオペランドを含めることができます．その演算結果の型は次に説明する簡単なルールで決められます．

6.1.1 一般的な規則

算術式中には被算術演算数 (arithmetic operand)，算術演算子 (arithmetic operator)，丸括弧 (parenthesis) を使うことができます．最低限一つのオペランド（被演算数）が含まれなければなりません．それぞれのオペランドは整数型，実数型，倍精度実数型，複素数型のどれかになりますが，演算の結果も一つの型を持ちます．オペランドは次のどれかです．

- 符号の付されていない直定数 (unsigned literal constant)
- 名前付き定数
- 変数
- 配列要素
- 関数の参照
- 丸括弧で囲まれた式

さらに複雑な式は次の規則に従って構成することができます．算術式は次のような形式をとることができます．

operand
+operand
−operand
arithmetic-expression arith-op operand

ここで，算術演算子 *arith-op* は次のどれかです．

+	加算 (addition)
-	減算 (subtraction)
*	乗算 (multiplication)
/	除算 (division)
**	指数 (exponentiation)

この規則のいうところは，算術式は場合によっては先頭にプラス，マイナスの記号を持つ，演算子で区切られたオペランドを表す文字，からなるということです．先頭のプラス記号は何の働きもありませんが，マイナス記号は式の符号を反転させます．

式中に記されたすべての定数値には符号を記してはいけません．これは式の意味がわかりにくく曖昧になってしまうため，二つの演算子を続けて使わないようにするためです．

```
4 / -3.0**-1          （誤り）
```

この制約を回避するには丸括弧を使って演算順序を明示します．

```
4 / (-3.0)**(-1)
```

式の評価は次の順で行われます．

1. 丸括弧でくくられた式（sub-expression)
2. 関数の呼び出し (reference)
3. 指数の計算
4. 乗算と除算
5. 加算と減算，符号の反転 (negation)

同じ優先順位内では左のものから評価されます．ただし，指数は右から評価されます．従って，A/B/C は (A/B)/C と同じことに，X**Y**Z は X**(Y**Z) と同じことになります．

別の方法で評価できるならば，式の値をすべて評価する必要はありません．例えば，X が 0 ならば関数 FUNC の値を評価するまでもなく，

```
X * FUNC(G)
```

の結果を求めることができます．このことは，副作用のない関数だけを使っていれば何の問題も引き起こしません．

6.1.2 型変換

もし，演算子が二つの同じ型のオペランドに対して演算を行うならば，その結果の型はオペランドの型と同じになります．一方，二つのオペランドの型が異なるときには，暗黙の型変換 (implicit data type conversion) によって一つのオペランドが他のオペランドの型に変換されます．この型変換は

整数型 は 実数型 へ，実数型 は 複素数型あるいは倍精度実数型 へ

のように常に情報の損失を少なくするように行われます．

複素数型から倍精度実数型へ，あるいはその逆の変換は重要な情報の損失を伴わずに変換することができないため，これらの変換は禁止されています．すなわち，一つの演算子が，それぞれ複素型と倍精度実数型の二つのオペランドをとることはできません．それ以外の組み合わせは可能です．これらの型変換によって得られる結果は，適当な型変換の組込関数 REAL，DBLE，CMPLX を用いた場合と同

じです．これらの関数については後述します．なお，演算結果の型は演算の対象になる二つのオペランドのみから決まり，式中の他の変数，定数などからはその型が何であれ一切影響を受けません．

指数は型変換の例外になります．指数部が整数の場合にはその整数は型変換されず，その式はその整数回だけ乗算を行ったものとして評価されます．一方，指数部がそれ以外の型の場合には，内部で LOG と EXP の関数を呼び出して計算が行われます．

```
2.0**3    →   2.0 * 2.0 * 2.0        → 8.0
2.0**3.0  →   EXP(3.0 * LOG(2.0))    → 8.0
```

当然，上の計算の方が下よりも速く，かつ，正確に行われます．もし，指数部が負の場合には次に示すように，単に正の場合の結果の逆数を求めます．

```
2.0**(-3) → 1.0/2.0**3 → 1.0/8.0 → 0.125
```

実数型から倍精度実数型への型変換では，元々なかった情報を作り出すわけではないことに留意してください．たとえば，実数型変数 R と倍精度実数型変数 D を使って，

```
R = 1.0 / 3.0
D = R
```

としても，D の結果は例えば 0.3333333432674408... のようになり[19]，変数 R が元々持っていた値よりも 1/3 の真値に近い値になるということはありません．

6.1.3 整数同士の除算

整数同士の除算の結果は常に整数値になり，小数部分は切り捨て (truncate) られます（すなわち整数値に丸められる）．

```
 8 / 3   →  2
-8 / 3   → -2
 2**(-3) →  1/(2**3) → 1/8 → 0
```

このため，実数の定数ではたとえ小数部分が 0 であっても，小数部分を付けておくことが非常に重要になります．

前述の演算の規則によって，二つの演算の組み合わせによっては意図しない結果になることがあります．例えば，

```
(-2)**3  →  (-2) * (-2) * (-2)  →  -8
```

になりますが，(-2)**3.0 を評価するときにコンピュータによっては数学的に存在しない-2.0 の対数を求めようとする場合があり，この式は無効な式 (invalid expression) として扱われる場合があります[20]．同様に，

```
5.0 * 3 / 4 → 15.0 / REAL(4) → 15.0 / 4.0 → 3.75
```

ですが，

```
3 / 4 * 5.0 → REAL(3/4) * 5.0 → REAL(0) * 5.0 → 0.0
```

になってしまいます[21]．

[19]訳注:0.3333333 は実数型の有効桁 7 桁，それ以降は倍精度実数型しか保持できない桁であり，入っている数字はゴミ値です（ということを意図しての例示です）．

[20]訳注：g77+Windows XP では問題ありません．

[21]訳注：訳者が 3 つ目の式を加えました．また，この違いはどのコンパイラを用いても生じます．

6.1.4 制限事項

算術演算には数学的に定義できないため禁止されているものがあります．例えば，0 での除算 (division by zero)，負の値の実数乗，0 の負の指数などです．このようなエラー (error) が生じたときに，どのようにふるまうか Fortran の規格で定められているわけではありませんが，ほとんどのシステムでエラーメッセージを表示して，プログラムを停止 (abort) します．

コンピュータに記憶できる数値の範囲と精度が有限であるために生じるエラーもあります．二つの非常に大きな値の加算や乗算では記憶できる数値の範囲を超えてしまうことがあり，このエラーをオーバーフロー (overflow) と呼びます．絶対値が大きな負の整数値でも同様のエラーが生じることがあり，これをアンダーフロー (underflow) と呼びます．これに対しても，多くのシステムで警告 (warnig) メッセージを表示し，さらにプログラムを停止することもありますが，この種のエラーで整数演算を含んでいる場合には，エラーを検出できないプロセッサもあります．

式中で使われるすべてのオペランド（変数，配列要素，関数の参照）はその式が評価されるときに値が確定していなければなりません．変数と配列要素は，DATA 文を用いて初期化しない限り，初期値が確定されていないことに注意してください．式は，その式中の他のオペランドに依存する副作用のある関数の参照を含んではいけません．この詳細については 9.3 節を参照してください．

6.1.5 算術定数式

算術定数式 (arithmetic constant expression) は PARAMETER 文の中で，また，DATA 文中で DO 型並びのパラメータとして使うことができます．定数式 (constant expression) のすべてのオペランドは直接記された定数 (literal constant) とそれ以前に定義されている名前付き定数 (named constant) でなければなりません．変数や配列要素，関数の参照はすべて使えません．指数は整数乗に限って許されています．

同じ規則は整定数式 (integer constant expression) にも適用されますが，さらにこの規則に加えて，すべてのオペランドが整定数でなければなりません．整定数式は型宣言文や COMMON 文，DIMENSION 文中の配列の上下限の指定や CHARACTER 文での文字列の長さ指定に使うことができます．

6.1.6 整数に対するビット論理演算

Fortran プログラムで他のデジタル機器と直接通信するときには，多くの場合，ビットパターンに対するビットごとの論理演算が必要になります．標準の Fortran では，論理型変数は 1 ビットの情報だけを記憶することや整数型変数は算術計算にしか用いられないため，ビット演算の直接的な方法を一切提供していません．しかし，多くのシステムで整数型に対するビット演算の組込関数を拡張機能として用意しています．関数名はシステムによって変わり得ますが，典型的な名前は IAND，IOR，ISHIFT です．また，通常の論理演算子.AND.，.OR. を整数型のオペランドに対して使えるシステムも二三あります．ただし，これはかなり大きな拡張であり，あまり歓迎すべきものではありません．それは，単に可搬性を損なうということに加えて，コンパイラが算術式中の誤りを検出する機能を阻害するためです。

多くのシステムでは整数値を 8 進数や 16 進数表示するための書式記述子を使えますが，これも非標準です．

6.1.7 ガイドライン

異なった型を混在させた式を使うときには，型変換の規則が期待するように機能するか注意深く吟味する必要があります．これに対して，陽に型変換関数を用いても何の害もありませんし，式の働き

がより明確になる効果もあります.

定数を記すときには型に対して特別な注意を払う必要があります．実際には実数値でなければならない定数を整定数の形式で書いてはいけません．式中ならばこのように記しても問題が生じない場合が多いのですが，手続きの引数に，手続き中で定義された型と異なった型の定数を書くと致命的なミスになります.

複数行にわたる長く複雑な式は読むのが難しく，よりプログラミングの誤りの原因となりやすくなります．このようなときには，一時的な変数を使ってでも短い式に分けて計算を行った方がよい場合があります.

私たちは可能な限り効率的なプログラムを書くことにこだわってしまいがちです．現在のコンパイラを使う限り，実行速度を上げる (optimizing speed・最適化) つもりで式の順序を変えてもほとんど意味はありません．例外は，組込関数が提供されているときにはそれを用いるのが一番よいということで，例えば，SQRT 関数を用いた方が X**0.5 を計算するよりも速く，かつ，正確な結果を得ることができるはずです.

DATA 文によって値が定義されていない記憶領域は自動的に 0 で初期化されるシステムもありますが，このことを前提としてプログラミングをしてはいけません.

6.2 算術組込関数

組込関数は定義することなくシステムによって供給される関数であり，どのプログラム単位からでも使うことができます．組込関数の特別な属性については 9.1 節で説明します.

多くの算術組込関数は総称名 (generic name) を持っています．総称名の関数は複数の異なった型の引数に対して同じように使うことができます．例えば，平方根関数 SQRT は実数型，倍精度実数型，複素数型の引数をとることができます．Fortran システムは内部で個別名の関数 (SQRT，DSQRT，CSQRT) から適当なものを自動的に選択します．個別名 (specific name) はほとんどの状況で無視してしまって構いませんので，Appendix にリストを示すにとどめます．ほとんどの組込関数の型は引数の型と同じですが，型変換関数などの例外もあります.

以下の説明では，引数の数と型をアルファベットで示します．I は整数型 (integer)，R は実数型 (real)，D は倍精度実数型 (double precision)，X は複素数型 (complex) を示します.

左辺のアスタリスク (*) は結果の型が引数の型と同じであることを示します．もし複数の引数を渡す場合にはすべて同じ型でなければならないことに注意してください．以上の説明から，例えば，I=NINT(RD) の記述は NINT 関数が 実数型あるいは倍精度実数型の引数を一つ必要とし，計算結果は常に整数型であることを示します．一方，*=ANINT(RD) は結果の型が引数の型と同じになることを示します.

6.2.1 三角関数

これのグループの関数は実数型と倍精度実数型の引数をとることができます．SIN と COS は複素数型の引数もとることができます．結果の型はすべて引数の型と同じになります.

* = SIN(RDX)	正弦関数 (sine)，角度はラジアン単位
* = COS(RDX)	余弦関数 (cosine)，角度はラジアン単位
* = TAN(RD)	正接関数 (tangent)，角度はラジアン単位
* = ASIN(RD)	逆正弦関数 (arc-sine)，結果は $-\pi/2$ から $+\pi/2$ まで.
* = ACOS(RD)	逆余弦関数 (arc-cosine)，結果は 0 から $+\pi$ まで.
* = ATAN(RD)	逆正接関数 (arc-tangent)，結果は $-\pi/2$ から $+\pi/2$ まで.
* = ATAN2(RD,RD)	1 番目の引数/2 番目の引数で計算される逆正接関数，結果は $-\pi$ から $+\pi$ まで．少なくとも二つの引数のどちらかは非零でなければならない.
* = SINH(RD)	双曲線正弦関数 (hyperbolic sine)
* = COSH(RD)	双曲線余弦関数 (hyperbolic cosine)
* = TANH(RD)	双曲線正接関数 (hyperbolic tangent)

SIN，COS，TAN の引数は度単位ではなく，ラジアン単位の角度であることに注意してください．引数には正負構わずに任意の大きさの角度を与えることができますが，極端に値の絶対値が大きくなると精度が落ちます．同様に，逆三角関数 (inverse trigonometric function) もラジアン単位の角度を返します．ASIN と ACOS の引数は-1〜1 の範囲になければなりません．ATAN2 は正しい象限の角度が得られるので非常に便利な関数です．例えば次のような結果が得られます.

```
ATAN(0.5)         = 0.4636476
ATAN2(2.0,4.0)    = 0.4636476
ATAN2(-2.0,-4.0) = -2.677945 ( = 0.4636476 - π).
```

6.2.2　その他の超越関数

* = SQRT(RDX)	平方根関数 (square root)
* = LOG(RDX)	自然対数関数すなわち底が e の log （ここで $e = 2.718281828\ldots$）
* = EXP(RDX)	指数関数，すなわち e の指数．自然対数の逆関数
* = LOG10(RD)	底が 10 の対数関数

LOG10 はデシベル計算などに便利な関数ですが，このグループの中では唯一，複素数型の引数をとることができないことに注意してください.

6.2.3　型変換関数

これらの関数は 4 種類ある算術型の一つから他の算術型へ変換します．算術式や代入文中に異なった型が混在している場合には，これらの関数が自動的に呼び出されます.

I = INT(IRDX)	切り捨て (truncation) によって整数型へ変換
R = REAL(IRDX)	実数型へ変換
D = DBLE(IRDX)	倍精度実数型へ変換
X = CMPLX(IRDX)	複素数型へ変換
X = CMPLX(IRD,IRD)	複素数型へ変換

整数型への変換関数 INT は切り捨てによって整数化します．もし，近い整数値が必要なときには NINT 関数（次節）を使ってください．CMPLX 関数は二つの引数を渡す場合（あるいは複素数型を一つ渡す場合）以外には，虚部が 0 の複素数値に変換します.

型変換によって，多くの場合情報を失うことを認識することが重要です．特に倍精度実数型から他の型へ変換する場合には重要な意味のある桁を失う可能性があります．

6.2.4 最小値と最大値

MIN 関数と MAX 関数は二つ以上の任意個数の引数をとることができるかわった関数です．結果の型は引数の型と同じになります．

```
* = MIN(IRD,IRD,...)          引数の中の最小値を返す
* = MAX(IRD,IRD,...)          引数の中の最大値を返す
```

値をある範囲に限定する目的に，これら 2 つの関数を組み合わせて使うことができます．例えば，TEMPER の値を 32〜212 の範囲に限定するには次のようにします．

```
      MAX(32.0, MIN(TEMPER, 212.0))
```

ここで，範囲の最小値が MAX 関数の引数に，最大値が MIN 関数の引数になることに注意してください．

大きな配列の最大（あるいは最小）の要素を見つけるには，次のようにループを用いる必要があります．

```
*Find largest value in array T of N elements:
* N 要素の配列 T の最大値を探す
      TOP = T(1)
      DO 25,I = 2,N
         TOP = MAX(T(I), TOP)
25    CONTINUE
*TOP now contains the largest element of T.
*今，TOP に T の最大の要素が格納されている．
```

6.2.5 その他の関数

```
* = AINT(RD)          小数部分を切り捨てる（INT と同様）　ただし，型は保存される．
* = ANINT(RD)         四捨五入して整数値に丸める．
I = NINT(RD)          四捨五入し，整数型へ変換する．
* = ABS(IRD)          絶対値を返す（すなわち，負値の符号を反転する）．
R = ABS(X)            複素数の絶対値（実部と虚部の二乗和の平方根）を返す．
* = MOD(IRD,IRD)      一番目の引数/二番目の引数の余り．
* = SIGN(IRD,IRD)     符号を移す．もし，2 番目の引数 (A2) が負ならば，1 番目の引数の符号
                      を反転した結果 (-A1) を返す．A2 が 0 あるいは正なら A1 を返す．
* = DIM(IRD,IRD)      1 番目の引数 (A1) と 2 番目の引数 (A2) の大きい方から小さい方を引い
                      た差を返す．もし， A1 > A2 なら A1-A2 を返す．
D = DPROD(R,R)        二つの実数値の積を倍精度実数型で求める．
R = AIMAG(X)          複素数値の虚部を取り出す．実部は REAL 関数で取り出すことができる．
X = CONJG(X)          共役複素数を返す．
```

INT と AINT が切り捨て（0 の方向へ丸める）であるのに対して，NINT と ANINT は四捨五入します．従って次のようになります．

```
      INT(+3.5) =  3       NINT(+3.5) = 4
      INT(-3.5) = -3       NINT(-3.5) = -4
```

実数値の小数部は次のどちらかの方法で簡単に得ることができます.

```
X - AINT(X)
```

あるいは

```
MOD(X, 1.0)
```

どちらの場合も X が負の場合には結果も負になります．符号を換えたいときには ABS 関数を使います.

MOD 関数にはもう一つの使い方があります．たとえば，修正ユリウス日 (Modified Julian Date・MJD) などの通日 (経過日数）から曜日を調べるには次のようにします.

```
MOD(MJD,7)
```

は 0 から 6 の値をとり，それぞれ，水曜日から火曜日に対応します．同様に，もし ATAN2 を使って 0〜2π（-π〜+π でなく）の範囲の結果を得たい場合，2π として定義された TWOPI を用いて，次のようにします.

```
MOD(ATAN2(X,Y) + TWOPI, TWOPI)
```

6.3 算術代入文

算術代入文 (arithmetic assignment statement) は次の形式を持ちます.

arithmetic-var = *arithmetic-expression*

ここで，*arithmetic-var* は算術変数あるいは配列要素です．たとえば，次の代入文は，N，K，ANGLE のすべての値が確定しているならば正しいものです.

```
IMAGE(N/2+1,3*K-1) = SIN(ANGLE)**2 + 1.0
```

もし，左辺のオブジェクトが右辺の式と異なる型を持っているときには，左辺の型に合うように型変換関数 (INT，REAL，DBLE，CMPLX）を自動的に選択して，型変換を行います．多くの型変換が情報を失うものであることに注意してください．もし，左辺のオブジェクトが配列要素ならば，その添字には任意の整数式を使うことができます．ただし，この代入文が実行される前にこれらの整数式に含まれるすべてのオペランドの値が確定している必要があり，また，それぞれの式が配列の宣言時に示された添字の範囲になければなりません.

7 文字操作と論理型

この章では Fortran の非数値型データを扱う仕組みについて説明します．文字型のデータは，ファイル名やエラーメッセージの形式も含めれば，実際にほとんどすべてのプログラムに使われています．また，現在の Fortran の文字の操作の機能は非常に強力になっています．論理式は IF 文で使われるため，論理型はさらに重要なものです.

7.1 文字操作の機能

文字型は他の型と重要な点で異なっています．それは文字型があらかじめ定められた長さを持つということで，これは記憶する文字の数のことです.

文字定数の長さは，単にアポストロフィーで囲まれた文字の数（例外は，文字列中でアポストロフィーを二つ続けるとアポストロフィー一つになります）になります．すなわち，

```
'it''s'
```

は長さ 4 の文字定数「it's」になります.

すべての文字型変数や配列，関数はあらかじめその長さを指定しておく必要があるため，下の例のように，ほとんど全ての場合で CHARACTER 文を使って宣言する必要があります.

```
CHARACTER NAME*20, ADDRSS(3)*40, ZIP*7
```

名前付き文字定数についても同じですが，実際に記された文字定数の長さに自動的に合わせる特別な表記法 (notation) が可能で，文字数を数える手間を省くことができます.

```
CHARACTER TITLE*(*)
PARAMETER (TITLE = 'Latest mailing list')
```

文字型を固定長とすることによって，整形した表を出力する場合など，固定した書式で出力するのが容易になっています．しかし，他の言語のように可変長の文字列を使った方が便利な場合もあります．文字型の代入に関する規則はこのことを考慮したものと考えられます．もし，文字列が宣言よりも短い場合には空白が補われ，長い場合にははみ出た右の部分が切り捨てられます．従って大抵の場合，文字型変数が少なくとも扱わなくてはならない最長の文字列の長さを持つようにしておけば十分です.

文字情報を手続きに引き渡す場合には，仮引数の長さを対応する実引数の長さに自動的に合わせることができます．この「長さの情報を受け渡す表記法 (passed length notation)」によって，汎用的な文字列操作用の手続きを簡単に書くことができます.

文字列に対して最もよく行われる操作は，分割したり連結したりすることでしょう．部分列の表記法 (substring notation) を用いることで，文字列変数や配列要素の任意の部分を取り出すことができます．文字列（や部分列）は，文字演算子 (character operator) を用いた文字式 (character expression) によって 2 つの文字列を連結 (concatenation) することができます．これらについては次の 2 つの節で説明します.

重要なもう一つの文字操作は，長い文字列からある決まった文字の並びを探しだすことです．これは組込関数 INDEX を用いて行うことができます.

一方，組込関数 ICHAR や CHAR は，コンピュータの持つ文字セットに従って一つの文字から整数への変換や，逆の変換を行うものです．これよりも複雑な，数値から文字列への変換やその逆の変換には，書式指定の強力な機能を用いて内部ファイルに READ，WRITE する方法が一番です．この仕組みについては 10.3 節で説明します.

文字列は関係演算子や組込関数を用いて比較することができます．後者はコンピュータの文字セットとには依らずに，ASCII コード表の順 (ASCII collating sequence) に従って比較します．詳しい説明は 7.6 節で行います.

7.2　部分列

部分列 (substring) の表記法を用いると，文字型変数や配列要素から連続した一部分を選ぶことができます．文字列中の文字の位置は一番左を 1 として数えます．配列の添字の下限が変えられるのに対して，文字列の一番左は常に 1 で変えることはできません．部分列は，単に取り出したい部分列の最初と最後の文字の位置を与えるだけで指定できます．例えば,

```
CHARACTER METAL*10
METAL = 'CADMIUM'
```

に対して，METAL(1:3) は'CAD' に，METAL(8:8) は空白文字になります．後者は，宣言された長さになるように自動的に空白文字が補われるためです.

部分列は少なくとも一文字以上の長さでなければなりません．部分列は通常，文字型変数と同じように使うことができます．先ほどの例に続けて次の代入文

```
METAL(3:4) = 'ES'
```

を実行すると，METAL の値が'CAESIUM　'に変わります（全部の長さが 10 文字ですので，後ろの 3 文字に空白文字が入っています.）

7.2.1 部分列の規則

部分列を示す丸括弧の中にはコロン「:」が必ず必要で，その両側には定数だけでなく整数式も用いることができます．コロンの前の整数式が部分列の最初の文字の位置を示し，後の整数式が最後の文字の位置を示します．どちらの値も 1 から LEN の範囲になければならず（LEN は部分列を抜き出す元の文字列の長さ），また，結果として得られる部分列の長さは 1 以上でなければなりません．

コロンは必ず付けなければいけませんが，コロンの両側の整数式は省略可能です．コロンの前の値のディフォルト値は 1，後の値のディフォルト値は元の文字列の最後の文字の位置になります．従って，先ほどの例では METAL(:2) は'CA' に，METAL(7:) は後半に空白 3 文字がついた'M　'になります.

配列要素に対する部分列の表示は添字の表示に続けて記します．例えば，

```
CHARACTER PLAY(30)*80
PLAY(10) = 'AS YOU LIKE IT'
```

に対して，部分列 PLAY(10)(4:11) は'YOU LIKE' になります．部分列は文関数の定義文以外のすべての式中で用いることが可能であり，例えば，代入文の左辺として用いたり，入出力文によって値を与えることなども可能です.

7.3 文字式

文字演算子 (character operator) // は二つの文字列を連結 (concatenate/join) するのに使われ，この演算子が Fortran でサポートされた唯一の文字演算子です．例えば，'CUP'//'BOARD' は 'CUPBOARD' になります．結果として得られる文字列の長さはオペランドとなる二つの文字型変数の長さの和になります．丸括弧を文字式中に用いることができますが，結果には何ら影響を及ぼしません．文字列中の空白文字や後ろに補われた空白文字などは，結果となる文字列中にそのまま残されますので注意してください.

文字式の一般的な形式は次のようになります.

character-operand

あるいは

character-expression // *character-operand*

ここで，*character-operand* は次のどれかです.

- 文字型定数 (直定数，あるいは，名前付き定数）
- 文字型変数
- 文字型配列要素
- 部分列
- 文字型関数の参照

手続き (procedure) の中で文字を連結する場合には一つの特別な制約があります．長さの渡される仮引数 (passed-length dummy argument) は連結演算子のオペランドとしては，代入文中の連結演算子のオペランドだけになることができます．この一見気まぐれのような規則によって，コンパイラが必要な作業領域の必要量を決めることができるのです[22].

7.4　文字代入文

文字代入文 (character assignment statement) は一般的に次のような形式を持ちます.

char-var = *character-expression*

ここで，*char-var* は文字変数，文字型配列要素，部分列のどれか（文字型オブジェクト）です.

文字代入文には重要な制限事項があります．それは，左辺の *char-var* に右辺の文字式で参照される部分が含まれてはならないということ，すなわち，重複する部分があってはならないということです．従って，次の代入文

```
STRING(1:N) = STRING(10:)
```

は，N が 9 以下の場合に限って有効です．当たり前のことですが，この制限は適当な長さを持つ一時的な変数を使うことで簡単に回避できます.

（この例のように）部分列に文字が代入される場合，部分列を取り出す元の文字列の部分列の部分以外は全く影響されないことに注意してください．もし，文字列が初期化されていない場合には代入された部分列以外の部分は初期化されていないままですし，何らかの文字が記憶されている場合にはその文字がそのまま保持されます.

式とその値が代入される文字型のオブジェクトが互いに異なった長さを持つ場合があります．もし式の方が長い場合には右側のはみ出る部分が失われ，短い場合には空白文字が補われます．文字型のオブジェクトは適切な長さを持つように注意して宣言しないと，次の例のように予期せぬ結果になることがあります.

```
CHARACTER AUTHOR*30, SHORT*5, EXPAND*10
AUTHOR = 'SHAKESPEARE, WILLIAM'
SHORT = AUTHOR
EXPAND = SHORT
```

この例では，EXPAND の値は後半の 5 文字が空白文字の'SHAKE　　'になってしまいます.

7.5　文字操作組込関数

この節では主要な 4 つの文字操作組込関数について説明します．他に ASCII コード順に文字列を比較する 4 つの文字列比較関数がありますが，これについては 7.6 節で説明します.

7.5.1　CHAR，ICHAR

この二つの関数はそれぞれ，整数から文字型へ，文字型から整数型へコンピュータの文字コードに従って変換します．現在はほとんどのコンピュータが ASCII コードを使っていますが，決してすべてではないため，可搬性が重要なソフトウェアの作成に際してはこれらの関数の使用に十分注意しなくてはなりません.

CHAR(I) はコード表の I 番目の文字を返します．たとえば，ASCII コードを使っているコンピュータでは，ASCII コード表の 74 番目の文字が「J」であるため，CHAR(74) は'J' を返します.

[22]訳注：9.5 節も参照.

ICHAR(STRING) は文字型の引数 STRING[23]の最初の文字のコード表内の位置を返します．例えば，ASCII コードを使っているコンピュータでは，

```
ICHAR('JOHN') は  74
ICHAR('john') は 106
```

を返します．

7.5.2 INDEX

INDEX は二つの文字型の引数を取り，整数値を返す検索関数です．INDEX(S1,S2) は文字列 S2 を，普通それよりも長い文字列 S1 から探します．もし，S1 中に S2 が含まれれば，S1 中に見つかった最初の文字の位置を返します．もし，マッチしない（あるいは S1 の方が S2 よりも短い) 場合 0 を返します．次の例を考えます．

```
      CHARACTER*20 SPELL
      SPELL = 'ABRACADABRA'
      K = INDEX(SPELL, 'RA')
```

'RA' という文字列が SPELL 中に初めて現れるのが 3 文字目ですので，K は 3 になります．次にマッチする位置を探すためには，その次の文字から再度，検索を行わなければなりません．

```
      L = INDEX(SPELL(K+1:), 'RA')
```

部分列'ACADABRA' の中で'RA' にマッチするのは 7 文字目なので，この結果は 7 になります．全体の文字列中のこの位置を求めるためには，オフセットとした K を加算して 10 が得られます．

INDEX 関数は文字情報の操作では非常に便利です．例えば，NAME という文字変数に，名字とイニシャル，例えば，

```
      Mozart,W.A.
```

が格納されているとします．次のようにすると，イニシャルを名字の前に置きカンマを取り除いた書式に変更できます．

```
      CHARACTER NAME*25, PERSON*25
*...
      KCOMMA = INDEX(NAME, ',')
      KSPACE = INDEX(NAME, ' ')
      PERSON = NAME(KCOMMA+1:KSPACE-1) // NAME(1:KCOMMA-1)
```

結果として，PERSON には'W.A.Mozart'（全体が 25 文字になるように空白が補われています）が格納されます．代入文では左辺と右辺で重複する部分が許されない規則[24]のため，別の変数 PERSON が必要なことに注意してください．

7.5.3 LEN

LEN 関数は文字型の引数を取り，その長さを整数値として返します．引数が文字型の局所変数や局所配列要素であっても構いませんが，このときには単に定数値を返します．手続き (procedure) 中でその文字型の仮引数（と文字型の関数名）の長さが呼び出し側のプログラム単位から渡される場合には，

[23]訳注：STRING は文字列の意味.
[24]訳注：7.4 参照

手続きの呼び出しの度に長さが異なります．このような場合にも LEN は引数の文字型の宣言された長さを返すため，非常に有用な関数です．

また，LEN は文字列の後続の空白文字を除外した長さを得る目的にも，便利に使うことができます．次の関数は内部で LEN を使って，後続の空白文字を除外した長さを返すものです．

```
      INTEGER FUNCTION LENGTH(STRING)
*Returns length of string ignoring trailing blanks
*後続の空白を無視した文字列の長さを返す
      CHARACTER*(*) STRING
      DO 15, I = LEN(STRING), 1, -1
         IF(STRING(I:I) .NE. ' ') GO TO 20
15    CONTINUE
20    LENGTH = I
      END
```

7.6　関係式

関係式は 2 つの算術式，あるいは 2 つの文字式を比較します．結果は論理型で，真 (true) か偽 (false) のどちらかになります．次の例のように，関係式は IF 文で多く使われます．

```
      IF(SENSOR .GT. UPPER) THEN
          CALL COOL
      ELSE IF(SENSOR .LT. LOWER) THEN
          CALL HEAT
      END IF
```

通常使われる>や<などの文字が Fortran の文字セットに含まれないため，関係演算子 (relational operator) は .GT. や .LT. のように書きます．非常に多くの場合で関係式は IF 文で使われますが，次の例のように，後で参照するために論理型の変数や配列要素に記憶しておくこともできます．

```
      CHARACTER*10 OPTION
      LOGICAL EXIT
      EXIT = OPTION .EQ. 'FINISH'
*...
      IF(EXIT) STOP 'Finish requested'
```

論理式については次節で詳しく説明します．

7.6.1　関係式の一般的な形式

関係式の一般的な形式は次のようになります．

arithmetic-exprn　*rel-op*　*arithmetic-exprn*

or　*character-exprn*　*rel-op*　*character-exprn*

どちらの場合も結果は論理型 (logical type) になります．論理演算子 (relational operator) は次のどれかです．

.EQ. 等しい (=, equal to)
.GE. 以上 (≥, greater than or equal to)
.GT. より大きい (>, greater than)
.LE. 以下 (≤, less than or equal to)
.LT. 未満 (<, less than)
.NE. 等しくない (≠, not equal to)

英字名と区別するために，これらの演算子の両側にはピリオド「.」が必要であることに注意してください.

7.6.2 算術型の比較

比較する算術型の型が異なる場合には,（算術式で行われるように）どちらか一方に自動的に型変換が行われ，他方の型に合わせられます．型変換は常に,

整数型→実数型→複素型あるいは倍精度実数型

の方向に行われます．整数式を比較する場合には，.LE. と .LT.，同様に .GE. と .GT. では大きな違いがあります．従って，演算子を選択する前に，値が等しい場合にどのような動作が必要か十分に吟味する必要があります.

整数型以外の算術型を比較する場合，数値が正確に記憶されているわけではないことに留意する必要があります．すなわち，例えば，一方の値があらかじめ，0 あるいは他の小さい整数値でセットされている場合などの特殊な場合を除いて，.EQ. や .NE. を用いた比較をすることは賢明ではありません.

複素型については次の二つの制限があります．第一は倍精度実数型と比較が全くできないことです．第二は，複素数値には単純な大小関係がないため，.EQ. と .NE. 以外の関係演算子が使えないことです.

7.6.3 文字型の比較

文字型は文字型とだけ比較できます．両者で長さが違うときには，短い方に空白文字が補われ，同じ長さで比較が行われます．二つの文字列が等しい（あるいは等しくない）ことのテストは文字コードには依存せず，違いが見つかるまで一文字ずつ比較されていきます．その他の演算子 (.GE.，.GT.，.LE.，.LT.）を用いた比較はコンピュータの文字コードに依存します．二つの文字式は一度に 1 文字ずつ違いが見つかるまで比較が行われていきます．結果は比較される二つの文字の，コンピュータの文字コードの順序，すなわち文字コード表中の順に依存します.

Fortran 規格はすべてのシステムで使われる文字コード順として次のような基本的な条件を指定しています.

- すべての大文字は A<B<C... の順
- すべての数字は 0<1<2... の順
- 数字は英字の前，またはその逆
- 空白文字は英字と数字の前

しかし，文字と数字でどちらが先かは指定されていません．従って，英字と数字が混じったテキストを関係演算子を用いてソート（並べ替え）すると，その結果はコンピュータに依存したものとなります．例えば次の式

```
'APPLE' .LT. 'APRICOT'
```

は，二つの文字列の最初の違いは 3 文字目で，すべての Fortran の文字コード順で P は R の前にある

ため，常に真になります．しかし，

```
'A1' .GT. 'AONE'
```

は EBCDIC コードを使っているコンピュータでは真，ASCII コードを使っているコンピュータでは偽になります．これは前者では数字は文字の後，後者では文字の前にあるためです．

このような機種依存性のない文字比較を行うために，Fortran では 4 つの組込関数が別に用意されています．これらの関数は，コンピュータの文字コード順とは無関係に ASCII コードを用いて比較を行います．　これらの関数は次の 4 つです．

`LGE(S1, S2)`	大きいか等しい (greater than or equal to)
`LGT(S1, S2)`	大きい（greater than）
`LLE(S1, S2)`	小さいか等しい（less than or equal to）
`LLT(S1, S2)`	小さい（less than）

これらの関数は（任意の長さの）二つの文字型の引数を取り，論理値を返します．次の式，

```
LGT('A1', 'AONE')
```

は常に偽になります．

文字コードに小文字があるコンピュータでは，文字の比較は大文字小文字の違いに依存します．このようなときには，二つの引数をあらかじめ大文字あるいは小文字のどちらかに変換しておく方がよいでしょう．

7.6.4　ガイドライン

大文字と小文字の両方が使えるコンピュータではこの違いに影響を受けます．このようなコンピュータでは，比較関数を用いて実際にキーワードやコマンドの検索を行う前に，入力文字列を標準的な大文字に変換しておく方がよいでしょう．残念ながらこのような変換を行う標準組込関数はありませんが，多くのシステムで拡張機能として備えられています．

英文字や数字，その他の記号が混ざった文字列を並べ替える場合にはプログラムの可搬性を考慮して組込関数を用いるべきでしょう．しかしながら，その他の文字比較では関係演算子を用いた方がわかりやすく，また多少は効率的かもしれません．

7.7　論理式

論理式 (logical expression) は論理代入文 (logical assignment statement) で使われますが，次の例のような複数の条件が付された `IF` 文の中で最も使われています．

```
      IF(AGE .GE. 60 .OR. (STATUS .EQ. 'WIDOW' .AND.
     $   NCHILD .GT. 0) THEN
```

この例では，2 つの算術型の比較と文字型の比較の 3 つの関係式の値を組み合わせています．`.AND.`，`.OR.` などの論理演算子 (logical operator) も関係演算子と同様に英字名と区別するため両側にピリオドが必要です．`.OR.` 演算子は論理和 (inclusive or) を表し，`.NEQV.` は排他的論理和 (exclusive or) を表します．

7.7.1　規則

論理式は次のどれかの形式を持ちます．

- *logical-term*

- .NOT. *logical-term*
- *logical-expression logical-operator logical-term*

ここで，*logical-term* は次のどれかです.

- 論理型定数 (logical constant) (定数 あるいは 名前付き定数)
- 論理型変数 (logical variable)
- 論理型配列要素 (logical array element)
- 論理型関数の参照 (logical function reference)
- 丸括弧でくくられた論理式 (logical expression enclosed in parentheses)
- 関係式 (relational expression)

また，論理演算子 (logical operator) は次のどれかです.

.AND.	論理積 (logical and・logical conjunction)
.OR.	論理和 (logical inclusive or・logical inclusive disjunction)
.EQV.	論理等価 (logical equivalence)
.NEQV.	論理非等価 (logical non-equivalence) すなわち排他的論理和 (exclusive or)

論理式の規則では，オペランドの値を反転する単項演算子.NOT. が二番目にあるときだけ演算子を二つ続けることを許しています．次の表は，4 つの演算子とオペランド x と y の 4 つの組み合わせの結果をまとめたものです.

x	y	x .AND. y	x .OR. y	x .EQV. y	x .NEQV. y
偽	偽	偽	偽	真	偽
真	偽	偽	真	偽	真
偽	真	偽	真	偽	真
真	真	真	真	真	偽

論理式には関係式をオペランドとしてとることができ，関係式には算術型も含まれます．これらの演算子の優先順位は次のようになります.

1. 算術演算子 (6.1 節で説明した順位で計算が行われます)
2. 関係演算子
3. .NOT.
4. .AND.
5. .OR.
6. .EQV. と .NEQV.

.EQV. と .NEQV. が式中に同等に使われているときには，左から右の順に評価されていきます.
これらの規則によって必要な丸括弧が少なくなっています．例えば,

```
(X .GT. A) .OR. (Y .GT. B)
```

は，丸括弧をすべて取ってしまっても全く同じ意味になります.

Fortran では，もし論理式の値が確定するならば，その式に含まれるすべての項を評価することはありません．上の例では，もし X が A よりも大きいならば，式の値は次の比較の結果に依らずに真であり，Y と B を比較する必要はありません．これによって効率はよくなりますが，副作用のある関数は使うべきではありません.

7.7.2 ガイドライン

特に複数の行にわたるような，複雑な論理式や関係式は理解するのが困難になります．このような場合には，似たような条件を連続する行に並べて書き，丸括弧を使って計算順位を示しましょう．

7.8 論理代入文

論理代入文 (logical assignment statement) は次のような形式を持ちます．

logical-var = *logical-expression*

ここで，*logical-var* は論理型の変数か配列要素です．論理型変数や配列要素は主に，後で IF 文で使うために論理式の値を記憶しておくために使われます．

8 制御文

実行文は制御文で特に指定されない限り，上から順に実行されます．入出力文で END=や ERR=のキーワードが使われたときにも，実行順序が変わります．

8.1 制御構造

8.1.1 分岐

プログラムの実行手順を選択するにはブロック IF 文を用いるのが最もよい方法です．これは，指定した条件が真のときに実行される一つのブロック (block)（連続した文のかたまり）を設定したり，多様な可能性がある場合の複数のブロックを設定したりすることができます．IF ブロックに含まれる文が一つのときには,（歴史的な経緯から）論理 IF 文と呼ばれる省略形を用いることができます．

また，他の言語の “case” 文と似た，複数の分岐を行う計算 GO TO 文もあります．

8.1.2 ループ

もう一つの基本的な制御構造が繰り返し (repetition) です．もし，繰り返し回数があらかじめわかる場合には，DO 文を使いましょう．これは，DO ループと呼ばれるブロックを制御するものです．普通，DO ループの終端には CONTINUE 文が使われます．

Fortran には，他の言語において繰り返し回数が定まっていないときに使われる “do while” や “repeat unitl” に直接相当する文はありませんが，単純 GO TO 文と IF 文で同等のループを構成することができます．

8.1.3 その他の制御文

STOP 文はプログラムを終了するために使います．その他の実行順序に影響する文については他の節で説明します．END 文は 4.7 節，手続きの呼び出しに関する CALL 文や RETURN 文については 9 章で説明します．

8.2 IF ブロック

最も単純な IF ブロックは次のような形式のものです．

```
IF(N .NE. 0) THEN
   AVERAG = SUM / N
   AVGSQ = SUMSQ / N
END IF
```

ブロックに含まれる文は条件が真のときにだけ実行されます．この例では，0 での除算 (division by zero) を避けるため，N が 0 のときにはブロック中の文を実行しないようになっています．

IF ブロックには，その以外の場合を処理するために ELSE 文を付けることができます．

```
IF(B**2 .GE. 4.0 * A * C) THEN
    WRITE(UNIT=*,FMT=*)'Real roots'
ELSE
    WRITE(UNIT=*,FMT=*)'No real roots'
END IF
```

IF 文には論理式が使われ，その値は真か偽のどちらかであるため，どちらかのブロックは必ず実行されます．

もし，複数の条件がある場合には，ELSE IF 文を使って指定します．

```
IF(OPTION .EQ. 'PRINT') THEN
   CALL OUTPUT(ARRAY)
ELSE IF(OPTION .EQ. 'READ') THEN
   CALL INPUT(ARRAY)
ELSE IF(OPTION .EQ. 'QUIT') THEN
   CLOSE(UNIT=OUT)
   STOP 'end of program'
ELSE
   WRITE(UNIT=*,FMT=*)'Incorrect reply, try again...'
END IF
```

ELSE IF ブロックはいくつでも付けることができますが，IF 文が一度実行されるたびに，これらのブロックのうちただ一つのブロックだけが実行されます．ELSE ブロックがなく，条件が偽であった場合には何も実行されません．

8.2.1 ブロック IF 文の一般的な規則

ブロック IF 文の一般的な形式は次のようになります．

```
IF( logical-expression ) THEN
      a block of statements
ELSE IF( logical-expression ) THEN
      another block of statements
ELSE
      a final block of statements
END IF
```

IF THEN，ELSE IF，ELSE はそれぞれ一つのブロックを持っています．ELSE IF 文はいくつでも付けることができます．ELSE 文は（そのブロックとともに）なくても構いませんが，あっても一つだけです．

最初の文のブロックは最初の条件式が真のときだけ実行されます．それぞれの ELSE IF 以下のブロックは，それ以前のブロックが実行されておらず，かつ，その ELSE IF の条件が真のときだけ実行され

ます．ELSE ブロックがあるときには，それ以前のブロックが実行されていないときだけ，ELSE ブロックが実行されます．

一つのブロックが実行された後は，ブロック IF 文の制御構造の終端である END IF 文の次の文に制御が移ります（ただし，制御を他へ移す文でブロックから抜けた場合を除きます[25]）．

どのブロックにも，別の独立したブロック IF 文を入れてネスト（入れ子）(nest) にしたり，DO ループや他の全ての実行文 (END 文は除く）を含めることができます．

どのブロックにも外から制御を移すことはできませんが，ブロックの中から外へ制御を移すことに関しては全く制約はありません．

論理式に関する説明は 7.7 節をご覧ください．

8.2.2 ガイドライン

上で示したような字下げ (indentation) は必須ではありませんが，それぞれのブロックを数文字ずつ他の部分より下げておくことを強く勧めます．これによってブロックの構造がすぐにわかり，IF 文と END IF の対応関係を間違える危険を減らすことができます．字下げは，複数の IF ブロックがネスト (nest) されているときに特に有効です．例えば，

```
      IF(POWER .GT. LIMIT) THEN
          IF(.NOT. WARNED) THEN
              CALL SET('WARNING')
              WARNED = .TRUE.
          ELSE
              CALL SET('ALARM')
          END IF
      END IF
```

IF ブロックがあまりに深くまでネストされていると，文を書く欄 (statement field) の幅の制約が問題になることがよくあります．しかし，これはプログラム単位があまりにも複雑になりすぎた結果とも言えて，サブルーチンに分割した方がよい場合がほとんどです．END IF を付け忘れた場合にはコンパイラはエラーを出しますが，どこに付けなければならないかはコンパイラにはわかりません．このようなときにはコンパイラは混乱して非常にたくさんのエラーメッセージを出すかもしれません．

頻繁に実行される IF ブロックにたくさんの ELSE IF 文が含まれるときには，起こりやすい条件を最初の方に置いておくと，下の方の条件のテストが行われないため，効率が多少よくなります．

8.3 DO ループ

DO 文 (DO statement) はループ変数 (loop-control variable) のそれぞれの値に対して一度ずつ，文ブロックを繰り返し実行するものです．繰り返し回数はループの先頭の DO 文のパラメータで決定されます．次の例に示すように，“DO” のキーワードのすぐ後ろにはループの最後の文の文番号を付けます[26]．

```
*Sum the squares of the first N elements of the array X
*配列 X の最初の N 要素の二乗和を求める
      SUM = 0.0
      DO 15, I = 1, N
          SUM = SUM + X(I)**2
15    CONTINUE
```

[25]訳注：ブロックに GO TO 文がある場合など

[26]訳注：DO 文の文番号の直後にカンマがありますが，もちろんなくてかまいません．詳細は 1.4.2 節参照．

もし，配列の要素の一つおきに和を求めたいときには次のようにします.

```
      DO 15, I = 1, N, 2
          SUM = SUM + X(I)**2
15    CONTINUE
```

このとき，繰り返し中に I の値は 1, 3, 5, ... のように変化します．繰り返しの最後での値は N が奇数なら N，偶数なら N-1 になります．3 番目のパラメータが省略された場合には，増分 (step-size) は 1 になります．また，増分が負のときには I が減少するように繰り返しが行われます．例えば,

```
      DO 100,I = 5, 1, -1
          WRITE(UNIT=*,FMT=*) I**2
100   CONTINUE
```

は，25, 16, 9, 4, 1 の値を順に出力します.

ループは必要なだけネストすることができます．次の例は二次元配列 FIELD を 0 に初期化するものです.

```
      REAL FIELD(NX, NY)
      DO 50, IY = 1, NY
         DO 40, IX = 1, NX
            FIELD(IX,IY) = 0.0
40       CONTINUE
50    CONTINUE
```

8.3.1 DO 文の一般的な形式

DO 文には次の二通りの形式が可能です.

DO *label* , *variable* = *start* , *limit, step*

DO *label* , *variable* = *start* , *limit*

二番目の形式では増分が 1 になります.

文番号 *label* はループの終端を指定するもので，これより後の同一プログラム単位中の実行文に必ず付いていなければなりません．規則ではこの文番号は制御文を除くすべての実行文に付けることができますが，CONTINUE 文を使うことを強く勧めます．CONTINUE 文には，位置を示す以外の機能は全くありません.

DO 文の文番号の後のカンマはなくても構いませんが，1.4 節で述べたように有用なチェック機能があります.

その後の *variable* はループ変数 (loop control variable/loop index) と呼ばれ，変数でなければなりません（配列要素は不可）が，整数型，実数型，倍精度実数型の型の変数が使えます.

start , limit, step（初期値，終了値，増分）には整数型，実数型，倍精度実数型の式を使うことができます．*step* を指定するときには 0 の以外の値を指定し，省略すると 1 とみなされます．繰り返し回数は最初にループが実行される前に次の公式に従って計算されます.

$$\text{繰り返し回数} = \mathrm{MAX}(0, \mathrm{INT}((\mathrm{limit} - \mathrm{start} + \mathrm{step}) / \mathrm{step}))$$

もし，終了値が初期値よりも小さい場合には，増分が負でない限り繰り返し回数は 0 になります．繰り返し回数が 0 というのも許されてはいますが，これは，ループの中身が全く実行されずに単にループの次の文に制御が移されるだけ，ということを意味します．ループ変数は必ずしも終了値に達する必要なありません．これは特に増分が 1 より大きいときに起こり得ることです.

ループ内の文で初期値や終了値，増分を変えることも許されています．ただし，ループが実行される前に繰り返し回数は決定されているので，このことが繰り返し回数に影響を及ぼすことはありません.

ループ変数はループ内の式の中で使う（参照する）ことはできますが，ループ内で値を代入してはいけません．

別のループ変数を使うならば，DO ループはその内部に別の DO ループ全体を入れる（ネストする）ことができます．二つのネストされたループの終端を同じ文にすることも許されていますが，プログラムが非常にわかりにくくなるため，それぞれのループの終端には別の CONTINUE 文を使うことをお勧めします．同様に，IF ブロック全体を DO ループの中に入れることもできますし，その逆も可能です．

他の制御文を使って DO ループの外へ制御を移すことはできますが，外から DO ループの中にジャンプすることは許されていません．ループが終了する前に他の制御文を使って外に出た場合，ループ変数はそのときの値を保存しており，ループの外でこの変数を参照すると実際にループが実行された回数を知ることができます．

DO ループが普通に終了した場合，ループ変数には最後の繰り返しのときの値に増分を一つ加えた値が入っています．従って次の例では，最後の繰り返しのときの NUMBER は 99 ですので，ループを抜けた後の NUMBER は 102 になります．

```
      DO 1000, NUMBER = 1,100,3
1000  CONTINUE
```

何らかのエラーでループを抜けてしまった際には，この規則がとても便利な機能を発揮します．

```
      PARAMETER (MAXVAL = 100)
      REAL X(MAXVAL)
      DO 15, I = 1, MAXVAL
          READ(UNIT=*, FMT=*, END=90) X(I)
15    CONTINUE
90    NVALS = I - 1
```

この例では，文番号 90 の文によって，ファイル終端 (end-of-file/EOF) を検出してループを抜けた場合と所定の配列要素数 MAXVAL だけ読み込んで所定回数を終了した場合の双方で（特別な処理を加えることなしに），NVALS に実際にファイルから読み込んだ値の個数をセットすることができるのです．

8.3.2 ガイドライン

整数型以外のループ変数を用いる場合には，増分を加えていくうちに丸め誤差が蓄積してしまう危険があります．さらに，初期値や終了値，増分が整数型でない場合には，算定式が（NINT 関数でなく）INT 関数を用いているため繰り返し回数が期待する回数にならないこともあります．DO 文にすべて整数型の値を使う場合にはこれらの問題は起こり得ません．

8.4 論理 IF 文

論理 IF 文は，ブロック IF 文が一つだけの文を持つ場合の特別な形式と考えるのが一番です．例えば，

```
      IF(E .NE. 0.0) THEN
          RECIPE = 1.0 / E
      END IF
```

は，論理 IF 文を用いて次のように置き換えることができます．

```
      IF(E .NE. 0.0) RECIPE = 1.0 / E
```

論理 IF 文の一般的な形式は次のようになります.

IF(*logical-expression*) *statement*

statement（文）は *logical-expression*（論理式）が真のときだけ実行されます. *statement* には，DO，IF，ELSE IF，ELSE，END IF，END 以外の全ての実行文を使うことができます.

8.5 単純 GO TO 文

単純 GO TO 文 (unconditional GO TO statement) は同一プログラム単位の文番号の付された実行文に無条件に制御を移します. 一般的な形式は次のように書けます.

GO TO *label*

IF ブロックや DO ループの中に制御を移すことはできませんので注意してください.

8.5.1 ガイドライン

単純 GO TO 文は使い方によっては，プログラムの構造が非常に無規律 (undisciplined) なものになってしまいます. このようなプログラムは理解するのも，維持していくのも困難になります. 熟練したプログラマは GO TO 文と飛び先を表す文番号をまれにしか使いません. 残念ながら，Fortran にはこれに替わる制御構造が完全には整備されていないため，これらを全く使わないことはできません.

次の例は整数 M と N の最大公約数 (highest common factor) をユークリッドの互除法[27](Euclid's algorithm) で求めるものです. 簡単に説明すると，M と N の小さい方を大きい方から引いて，その値を大きい方の新たな値とし，両者が等しくなるまで繰り返し行うものです.

```
      PROGRAM EUCLID
      WRITE(UNIT=*, FMT=*) 'Enter two integers'
      READ(UNIT=*, FMT=*) M, N
10    IF(M .NE. N) THEN
         IF(M .GT. N) THEN
            M = M - N
         ELSE
            N = N - M
         END IF
         GO TO 10
      END IF
      WRITE(UNIT=*, FMT=*)'Highest common factor = ', M
      END
```

8.6 計算型 GO TO 文

計算型 GO TO 文 (computed GO TO statement) はたくさんの選択肢があり，かつ整数式の値に基づいて選択できる場合にブロック IF と同様の目的に用いるものです. 一般的な形式は以下のようになります.

GO TO(*label1, label2, ..., labelN*) , *integer-expression*

右の丸括弧の後のカンマはなくても構いません.

[27]訳注：この例ではわり算はしていませんが，割って余りを求めるのと引き算を繰り返すのは同じことです.

expression を評価し，もしその値が 1 のときには文番号リスト[28]の最初の文番号の文に，2 のときには 2 番目の文番号の文に，というように制御が移されます．もし値が 1 未満，あるいは，N より大きい場合（ここで N は文番号リストに列挙された文番号の個数です）には，その計算型 GO TO 文には何の効果もなく，ただ単にその次の実行文に制御が移されます．文番号リストには複数の同じ文番号を書くことができます．

計算型 GO TO 文には，分岐を無制限に使うと理解不能のプログラムになってしまうという，単純 IF 文と同様の問題点があります．これを避けるには，計算型 GO TO 文の直後から分岐先のプログラムを順番に並べ，最後の分岐先以外はすべて単純 GO TO 文で全体の構造から抜けるようにするのが最もよい使い方です．

あらゆる計算型 GO TO 文は適当な数の ELSE IF 節を持つ IF ブロックでで置き換えることができます．しかし，たくさんの節が必要なときには IF ブロックでは多少非効率になります．従ってその選択には，多数の文番号をもつ GO TO 文と比べたときの IF 文の明快さとのバランスを考える必要があります．

ここで，計算型 GO TO 文の例としてあげたのは 1〜12 月の月と西暦年を与えてその月の日数を計算するサブルーチンです．最後の分岐の部分以外の分岐プログラムはすべて単純 GO TO 文で分岐から抜けていることに留意してください．

```
      SUBROUTINE CALEND(YEAR, MONTH, DAYS)
      INTEGER YEAR, MONTH, DAYS
      GO TO(310,280,310,300,310,300,310,310,300,310,300,310)MONTH
*           Jan Feb Mar Apr May Jun Jly Aug Sep Oct Nov Dec
      STOP 'Impossible month number'
*February: has 29 days in leap year, 28 otherwise.
*2 月は閏年の年は 29 日，それ以外は 28 日
280   IF(MOD(YEAR,400) .EQ. 0  .OR. (MOD(YEAR,100) .NE. 0
     $                         .AND. MOD(YEAR,4) .EQ. 0)) THEN
          DAYS = 29
      ELSE
          DAYS = 28
      END IF
      GO TO 1000
*   Short months
*   小の月
300   DAYS = 30
      GO TO 1000
*   Long months
*   大の月
310   DAYS = 31
* return the value of DAYS
* DAYS の値（日数）を返す
1000  END
```

8.7 STOP 文

STOP 文はプログラムの実行を停止し，オペレーティングシステムに制御を返します．次は一般的な

[28]訳注：*label1, label2, ..., labelN* のこと．

形式です.

STOP '*character constant*'

文字定数 *character constant*（直定数のみ，名前付き定数は不可）はなくても構いません．指定された場合は通常端末に表示されます．Fortran66 との互換性のために，文字定数に変えて 0〜5 桁の数字列も使うことができます.

理想的にはプログラムはプログラムの最後だけから OS に制御を戻すべきですが，このときには END 文が必要なことを行ってくれます．実際にはよく構成されたプログラムでも，実行を続けることが無意味になる状況が生じ得ます．もし，主プログラムでこのような状態にあることを検出した場合には END 文にジャンプする方法が使えますが，手続きでは STOP 文を使う以外の方法はありません.

9 手続き

どのような計算も手続きとしてカプセル化することができます．手続きの主な目的は，プログラムのさまざまな場所から呼び出して同じ計算を行うことです．また手続きによって，異なったプログラムでも同じコードを使うことができるようになります．プログラムが大きくなって一つのパートとして扱いにくくなったときには，いくつかの部分に分割していくのがよい方法です．一つのプログラム単位の最適な長さは非常に短く，おそらく 100 行以下でしょう.

Fortran では次の 4 種類の手続き (procedure) を使うことができます.

- 組込関数 (intrinsic function)
- 文関数 (statement function)
- 外部関数 (external function) (関数副プログラム (function subprogram) とも呼ぶ)
- サブルーチン (subroutine)

組込関数は Fortran システムにあらかじめ備えられているものです．一方，他の 3 つの形式の手続きはユーザが定義する (user-written) ものです．文関数は文関数文で定義され，定義されたプログラム単位でのみ使うことができるほか，特別な制約があります．外部関数とサブルーチンは外部手続き (external procedure) の二つの形式であり，どちらも独立したプログラム単位として定義され,（ごくわずかな制限の下で）プログラムのどこからでも使うことができます.

9.1 組込関数

組込関数 (intrinsic function) にはたくさんの変わった特徴があります．それぞれの組込関数の型は Fortran システムが知っているため，通常のルールが適用されません．IMPLICIT 文や型宣言文は単独では組込関数には影響を及ぼしません．組込関数には総称名 (generic name) を持つものがあり，これが用いられた場合にはコンパイラが引数の型から適当な個別名の関数を選んでくれます.

MAX，MIN や CMPLX などは引数の数が固定されていません．しかし，これらの引数はすべて同じ型でなければなりません．ユーザ定義の手続きはオプショナルな引数や総称名を持つことはできません.

組込関数はどのプログラム単位でも使うことができるにも拘わらず，その名前は大域名でもなければ予約語にもなっていません．ただし，組込関数と同じ名前を変数や配列の名前にすることはやめましょう．混乱の元になり，結局はプログラムの拡張を妨げることになります．名前の衝突 (clash) はさらに重要な問題で，外部関数やサブルーチンの名前と衝突している場合には，曖昧さをなくすためにこれらの外部手続き名を EXTERNAL 文で宣言しておかなくてはなりません．これによって組込関数をユーザ定義の外部関数で置き換えることができます.

Fortran 規格では備えるべき多数の組込関数が規定されていますが，さらにそれに追加することは禁止されていません．多くのシステムで，例えば，現在の日時を取得する関数，疑似乱数 (pseudo-random number) を発生させる関数，正規分布を求める関数などが追加されています．このような非標準の関数を使うことの問題点は，同じ関数が備えられていない別のシステムにプログラムを移植するときに，その代わりを探す必要が生じることです．

算術計算用の組込関数は 6.2 節に，文字列用の組込関数は 7.5 節に詳しい説明があります．また，すべての組込関数のアルファベット順のリストを巻末に示しています．

9.2 文関数

文関数 (statement function) は全ての実行可能なプログラム単位の中で文関数定義文 (statement function statement) で定義されます．ただし，文関数はそれが定義されたプログラム単位でのみ使うことができます．文関数はその利用に際して制限があるのも事実ですが，それにしてもあまり合理的な理由なしに文関数を使わないプログラマがほとんどです．

文関数定義文は通常の代入文に似ています．例えば，

```
FAHR(CELS) = 32.0 + 1.8 * CELS
```

関数 FAHR は摂氏 (Celsius) の温度を華氏 (Fahrenheit) に変換します．FAHR(20.0) は 68.0 を返します．

文関数は任意個数の仮引数（上の例では CELS）を取ることができます．仮引数は必ず右辺の式中に現れなければなりません．右辺の式はそのプログラム単位で使われる定数や変数，配列要素などを含むことができます．関数が呼ばれたときのこれらの値がそのまま使われます．例えば次の例で，

```
REAL M1, M2, G, R
NEWTON(M1, M2, R) = G * M1 * M2 / R**2
```

代入文

```
FORCE = NEWTON(X, Y, DIST)
```

でこの関数を参照すると，引数 X，Y，DIST の値と，関数が参照された時点での G の値に基づいた結果が返されます．

文関数の定義では組込関数や外部関数，すでに定義されている文関数の参照を行うことができます．

```
PARAMETER (PI = 3.14159265, DTOR = PI/180.0)
SIND(THETA) = SIN(THETA * DTOR)
COSD(THETA) = COS(THETA * DTOR)
TAND(THETA) = SIND(THETA) / COSD(THETA)
```

上の定義によって rad 単位でなく度単位の三角関数が使えるようになります．

それぞれの仮引数（上の例では THETA）の通用範囲 (scope) は文の中だけに限られます．仮引数に使われた名前はプログラム単位内のどこでも同じ型を持った変数名として使うことができ，関数の値には何の影響も及ぼしません．

文関数はあらゆる型を持つことができます．関数名と引数は通常の型に関する規則に従います．文関数は次のように文字操作に便利に使えます．

```
LOGICAL MATH, DIGIT, DORM
CHARACTER C*1
DIGIT(C) = LGE(C, '0') .AND. LLE(C, '9')
MATH(C) = INDEX('+-*/', C) .NE. 0
DORM(C) = DIGIT(C) .OR. MATH(C)
```

この 3 つの関数は 1 文字の文字型の引数に対して論理型の値を返します．DIGIT は渡された文字が数字かどうか，MATH は演算子かどうか，DORM は数字あるいは演算子であるかをテストします．DIGIT はその定義に使われている文字比較の関数 LGE，LLE によって，コンピュータの文字コードに全く依存しないことに注意してください．

9.2.1 文関数の規則

文関数定義文はプログラム単位内の宣言文 (specification statement) の後，かつ，すべての実行文の前に置かなければなりません．DATA 文や FORMAT 文とは混ぜて使用することができます．一般的な形は次のようになります．

function（ *dummy1, dummy2, ... dummyN* ） = *expression*

function はあらゆる型を取ることができます．*expression* は *function* と同じ型を持つのが普通ですが，異なる場合にはどちらも算術型ならば通常の代入文の場合の型変換の規則が適用されます．

関数の名前はプログラム単位内の他の英字名と区別可能な名前でなければなりません．型宣言文で関数名を宣言することはできますが，その他の宣言文に現れてはいけません．(例外として，共通ブロックには文関数名と同じ名前を使うことができますが，共通ブロック名は常にスラッシュ「/」の間に置かれるため，混乱する危険はほとんどありません.）もし関数が文字型のときにはその長さの指定は整定数式 (integer constant expression) でなければなりません．

仮引数 (dummy argument) は（単に型と個数，位置を示すだけの）英字名 (symbolic name) です．この名前は仮引数の並びに二度以上現れてはいけません．これと同じ名前はプログラム単位内ならこの文関数定義文以外のどこでも同じ型を持つ変数名として使うことができます．

右辺の式 (expression) は仮引数をオペランドとして含まなければなりません．オペランドには他にも，

- 直定数 (literal constant)，名前付き定数，配列要素．これらは関数が実行された時点のこれらの変数の値が使われますので，値が確定していなければなりません．
- 組込関数や外部関数の参照．
- 同じプログラム内ですでに定義されている文関数の参照．
- 丸括弧でくくられた式 (complete expression)．

を含むことができます．

部分列 (character substring) はオペランドとして使えませんので注意してください．関数が実行されたときには，右辺の式中の変数と配列要素の値が確定していなければなりません．

9.2.2 ガイドライン

文関数は一つの文で定義しなければならないためプログラム中で限られた役割しか担っていませんが，文関数の参照は外部関数を参照するよりも効率的です．最近の多くのコンパイラは文関数の参照を（その方が効率的であるときには）インラインコード (in-line code) に展開[29]します．

もし，同じ文関数を複数のプログラム単位で使う必要がある場合には，INCLUDE 文を使うことでそれぞれに同じ定義を行うことができますが，普通は文関数の代わりに外部関数を用いた方がよいでしょう．

9.3 外部手続き

外部手続き (external procedure) には次の二つの形式があります．どちらも完全なプログラム単位の形式をもっています．

[29]訳注：関数が参照された位置に関数の計算を直接埋め込むこと．

- 外部関数 (external function). これは FUNCTION 文で始まるプログラム単位として定義されます. 関数は式の中でオペランドとして使われたときに実行されます.
- サブルーチン (subroutine). これは SUBROUTINE 文で始まるプログラム単位として定義されます. CALL 文で呼ばれたときに実行されます.

どちらの形式もプログラム単位の最後の文は END 文でなければなりません. その他の（PROGRAM 文, BLOCK DATA 文を除く）文もこれらのプログラム単位中で使うことができます.

外部手続きで使うための二種類の文が用意されています. SAVE 文は, 手続きが制御を呼び出し側に戻した後も手続き内の局所変数や配列の値を保存しておくための文です. これらの値は次に手続きが呼ばれたときに引き続き使うことができます. RETURN 文は手続きの実行を終了し, 制御を直ちに呼び出し側に返すものです. 手続きの最後の END 文を実行した場合も全く同じことになります. どちらについても後節で詳しく説明します.

ほとんどの Fortran システムでは Fortran 以外の言語で定義された外部手続きを呼ぶことも可能です. このような外部手続きは Fortran の手続きと同じように呼ぶことができますが, 具体的な操作については当然本書の範囲を超えてしまいます.

サブルーチンをより一般的な手続きと考えておくのがよく, 外部関数は呼び出すプログラム単位からただ一つの値だけを返してもらえばよいときだけ使う特殊な形式と考えるべきでしょう.

ここに示す例は, 時, 分, 秒から 0 時から経過した秒数に変換するものです. 一つの値だけ返せばよいので, 手続きは外部関数になっています.（このような簡単な例では, 文関数としても定義することができるでしょう.）

```
*TSECS converts hours, minutes, seconds to total seconds.
*TSECS は時, 分, 秒から経過した秒数に変換する.
      REAL FUNCTION TSECS(NHOURS, MINS, SECS)
      INTEGER NHOURS, MINS
      REAL SECS
      TSECS = ((NHOURS * 60) + MINS) * 60 + SECS
      END
```

もし TSECS(12,30,0.0) のように, プログラムの式中でこの関数を参照すると与えられた時刻を経過した秒数に変換します（この例では 45000.0 秒）. 関数名の後の丸括弧内の並び 12,30,0.0 は関数の実引数 (actual argument) と呼ばれ, 対応する仮引数 (dummy argument)

```
      (NHOURS, MINS, SECS)
```

に渡されます. この例では, 引数の並びは関数の外部から関数の内部に値を渡すことのみに使われており, 関数名を介して呼び出し側に値を返しています. 一方, サブルーチンでは値を返す関数名がありませんが, 引数によってどちらの方向, あるいは両方向に値をやりとりできます. 規則としては関数でも, この万能の方法を使うことは可能ですが, 避けた方がよいでしょう.

次の例は TSECS 関数の逆の変換を行うものです. この場合, 3 つの値を呼び出し側のプログラム単位に返す必要があるため, 外部関数の形式は不適当であり, 代わりにサブルーチンを使わなければなりません.

```
*Subroutine HMS converts TIME in seconds into hours, mins,secs.
*サブルーチン HMS は総経過秒数 TIME を時, 分, 秒に変換する.
      SUBROUTINE HMS(TIME, NHOURS, MINS, SECS)
      REAL TIME, SECS
      INTEGER NHOURS, MINS
```

```
NHOURS = INT(TIME / 3600.0)
SECS = TIME - 3600.0 * NHOURS
MINS = INT(SECS / 60.0)
SECS = TIME - 60.0 * MINS
END
```

このサブルーチンは，例えば次のように実行することができます.

```
CALL HMS(45000.0, NHRS, MINS, SECS)
WRITE(UNIT=*, FMT=*) NHRS, MINS, SECS
```

この例では，最初の引数が値をサブルーチンに渡し，残りの 3 つの引数で計算された結果を返しています．それぞれの変数がどちらの方向へ（あるいは両方向へ）値を渡すのかを指定する必要はありませんが，どちらへ値を受け渡すのかによって，実引数の形式についていくつかの規則があります．この規則について次に詳しく説明します.

9.3.1 手続きの独立性

それぞれのプログラム単位は他のプログラム単位から独立した英字名と文番号を持っています．型宣言文と IMPLICIT 文はそれぞれのプログラム単位で独立した型宣言をすることができます.

外部手続きはその中で他の手続きを呼ぶことができますし，さらにそこから別の手続きを呼ぶこともできます．しかし，直接的にも間接的にも自分自身を呼び出すことは許されていません．これは再帰呼び出し (recursive call) と呼ばれ，Fortran では許されていないのです[30].

9.3.2 値の受け渡し

外部手続きと値を受け渡す方法として次の 3 つが可能です.

- 引数並び (argument list) を使う方法．上記 2 つの例を参照してください．これは最もお勧めできる方法で，柔軟性，独立性 (modular) に優れます．これについては本節の後半で詳しく説明します.
- 共通ブロック (common block) を使う方法．複数のプログラム単位で共有されるメモリ領域に配置された変数や配列を使う方法です．手続き同士が互いに強く関連する場合に便利な方法ですが，それ以外の場合にはあまりお勧めできません．共通ブロックについては 12 章で説明します.
- 外部ファイル (external file) を使う方法．外部ファイルを用いて受け渡しをする方法は，便利でも効率的でもありません．しかし，外部ファイルは「大域的」であるという利点があるためここに挙げました．一旦あるプログラム単位からファイルをオープンすると，もしその入出力装置番号 (I/O unit number) を知ることができるならば，どのプログラム単位からでもそのファイルをアクセスすることができます．装置番号は整数型の引数として手続きに渡すことができます.

9.3.3 手続きの実行

Fortran を使う上で，実際に Fortran システムがどのように値の受け渡しを行うかを知る必要はありません．しかし，手続きを扱うための規則は多少複雑なものですが，それが実現されている基本的な仕組みを理解すると，この規則がより簡単に理解できると思います．Fortran の規格では，値の受け渡

[30]訳注：最近のコンパイラではできるようですが，g77 ではサポートされていません.

しは実引数のアドレスを渡して行うことになっています．実際にはそうしている場合としていない場合があるようですが，値の受け渡しに見られる特徴はアドレス渡しをした場合の特徴と同じです．

このことは，仮引数を参照する場合や仮引数に値を代入する場合には，実引数が配置されているメモリに操作しようとしているということになります．この方法を用いているため，非常に大きな配列であっても効率的に手続きに渡すことができるのです．文字型の受け渡しには少し違いがあり，アドレスに加えて文字型の長さも渡すようになっています．

関数の参照を行った場合や CALL 文を実行した場合，まず，引数の並びに含まれる式が評価されます．その後引数のアドレスが手続きに渡されます．制御が戻されるときには，引数並びの対応する引数には自動的に最新の値が保持されることになります．

9.3.4 副作用のある関数

Fortran の規則では，その実引数や共通ブロックの変数の値を変えてしまう副作用 (side-effect) のある関数を許しています．副作用のある関数は，式中のその他のオペランドが影響されてしまう式の中や，添字や部分列の参照において，その参照に含まれる他の式が影響されてしまう場合にはその参照の中で使うことはできません．この規則は，コンピュータが式の値を評価する順番に式の値が依存しないようにするためのものです[31]．

また，入出力文を使用している関数（内部ファイルに入出力する場合も含む）は他の入出力文の式には使うことができないという制限があります．これは，入出力の操作が再帰的に起こらないようにするためのものです[32]．

副作用を持つ手続きにはサブルーチンの形式にするのが間違いなく最善の方法です．

9.4 外部手続きの引数

引数は手続きに値を渡したり，受け取ったり，あるいは双方向の受け渡しをすることができます．これは，手続き中で仮引数がどのような使われ方をするかで決まります．列挙する引数の順序は任意ですが，手続きに値を渡す引数が先，次に双方向の受け渡しをする引数，最後に手続きから値を受け取る引数，というのが一般的な順序です．

引数の結合 (argument association) に関する規則は外部関数もサブルーチンも同じです．外部手続きの仮引数 (dummy argument) の並び（formal argument とも呼ばれる）は FUNCTION 文あるいは SUBROUTINE 文で宣言されます．引数はいくつでも設定することができますし，引数がない場合も可能です．CALL 文や SUBROUTINE 文で引数がない場合には丸括弧を省略できますが，FUNCTION 文や関数の参照では省略できません．

仮引数の並びは単に，次に示す英字名の（混在した）リストです．

- 変数
- 配列
- 手続き

当然，一つの仮引数並びに同じ名前が複数あってはなりません．

仮変数 (dummy variable)，仮配列 (dummy array)，仮手続き (dummy procedure) の違いは手続き内での使われ方によって判断されます．仮配列の添字の範囲は続く型宣言文や DIMENSION 文の中で宣言

[31]訳注：g77+WindowsXP で試してみましたがコンパイル・実行ともに可能のようです．ただし，このような関数を使用すると結果が使用したコンパイラによって変わってしまうとのことですので避けるべきでしょう．

[32]訳注：これも g77+WindowsXP で試してみたところ，コンパイルはできますが，`I/O recursion: I/O started while already doing I/O` というランタイムエラーになりました．

されていなければなりません．仮手続きは CALL 文や EXTERNAL 文の中で，あるいは関数の参照として使われていなければなりません．それ以外は消去法によってすべて仮変数になります．仮変数や仮配列は局所変数や局所配列と全く同様に実行文中で使用することができますが，SAVE 文，COMMON 文，DATA 文，EQUIVALENCE 文には使うことができません．

9.4.1　引数の結合

関数の参照や CALL 文中の実引数は FUNCTION 文や SUBROUTINE 文の対応する仮引数に結合されます．主な規則は次の通りです．

- 実引数と仮引数は同じ数だけなければなりません．実引数と仮引数の二つの並び中の同じ位置にある引数同士が結合されます．Fortran 77 では可変長引数（引数の個数が必要に応じて変わる）はサポートされていません．
- 仮引数が変数，配列，関数として使われる手続きである場合，対応する実引数は同じ型を持たなければなりません．
- 仮引数が配列である場合には，その上下限は対応する実引数のそれよりも広い範囲であってはなりません．その代わり，仮配列（仮引数の配列・dummy array) の上下限を手続きの引数として渡すことができます．これによって仮配列を整合配列 (adjustable array) とすることができます．整合配列と疑似寸法仮配列 (assumed-size dummy array) は 9.6 節で説明します．
- 仮引数が文字型の場合にはその長さは対応する実引数の長さを超えてはなりません．その代わり，文字型の引数の長さを渡す方法を用いることができます．9.5 節を参照してください．

プログラム単位はそれぞれ独立にコンパイルされるため，コンパイラが実引数と仮引数の不一致をチェックするのは不可能です．一方，リンカが不一致を検出するのは原理的には可能ですが，この機能はほとんど実装されていないようです．特に型や配列の寸法の不一致は頻繁に犯す誤りですが，この誤りを検出するのは非常に困難です．プログラムが誤った結果を出すことから，誤りに気づくということもしばしばです．このことからも，いかに手続きのインターフェースをチェックすることが重要であるかわかります．

9.4.2　重複した引数

もし対応する仮引数に値が代入される場合には，その手続きの呼び出しで同じ実引数を二度以上使うことはできません．例えば，次の関数

```
SUBROUTINE FUNNY(X, Y)
X = 2.0
Y = 3.0
END
```

に対して，

```
CALL FUNNY(A, A)
```

とすることはできません．これは，関数が同じ変数 A にどちらか定められていない順序で 2.0 と 3.0 を代入されることから，その結果に不確定要素が生じるためです．

同様の制限は共通ブロックや手続きの引数並びを通して返される変数についても適用されます．

9.5 変数を引数にする場合

もし手続きの仮引数が変数で，かつ，手続き内で値が代入される場合には，対応する実引数は次のどれかでなければなりません．

- 変数
- 配列要素
- 文字型の部分列

一方，手続きの実行に際して仮引数の値が保存される場合には，実引数には上記の 3 つに加えて，

- 定数も含む任意の形式の式

を使うことができるようになります．

この制限の理由は次の例を見れば簡単に理解できます．

```
SUBROUTINE SILLY(N, M)
N = N + M
END
```

もし，このサブルーチンが次のような文

```
NUMBER = 10
CALL SILLY(NUMBER, 5)
```

で呼ばれるならば，CALL 文の結果として NUMBER は 15 になります．しかし，最初の引数を定数にした

```
CALL SILLY(10, 5)
```

は正しくありません．これはサブルーチンの終了時に，実引数である定数 10 に 17 の値を返そうとするためです．

このような誤りに対する動作はシステムに依存します．定数値の書き換えを検出してエラーメッセージを出力するシステムもありますし，書き換えを無視してそのままプログラムの実行を続けるものもあります．さらに，定数値を別の値に書き換えて実行をそのまま続けてしまうシステムもあります．この場合，その後にはこの定数値として 17 が使われることになります．このような動作はわかりにくく，また，検出するのが困難であるため，不用意にこのような誤りを犯すことのないように注意することが大切です．

他の人が作った手続きを使うときには，このような意図しない影響を心配するかもしれません．一般には，次のように定数を式に書き直すことで，手続きが定数の値を書き換えてしまうのを防ぐことができるはずです．

```
CALL SILLY(+10, +5)
```

あるいは，

```
CALL SILLY((10), (5))
```

このように書くことで定数値の書き換えを防止できるはずですが，そもそも，定数に対応する仮引数の値を書き換えることは手続きに関する Fortran の規則に反しています．Fortran の規則に反したプログラムを書くことで得られる利益と，定数値を書き換えるリスクを秤にかけて判断する必要があります．

9.5.1 式，添字，部分列

もし実引数に式が含まれるときには，手続きの実行に先立って式の評価が行われます．たとえ，手続きの中でこの式中のオペランドが書き換えられる場合でも，仮引数に渡される値には何の影響もありません．同じ規則は配列の添字や文字型の部分列の式にも適用されます．例えば，次のような手続きの呼び出し

```
CALL SUB( ARRAY(N), N, SIN(4.0*N), TEXT(1:N) )
```

に対して，手続き内で 2 番目の引数 N に別の値が代入される場合でも，他の引数の式の値にはすべて N の元の値が使われるため何の影響もありません．もちろん，手続きの終了後には，呼び出し側に書き換えられた N の値が返されます．

9.5.2 長さの引き渡される文字型引数

文字型の仮引数に特殊な長さ指定*(*) が用いられた場合，対応する実引数の長さを自動的に取得します．

この例として，文字列に含まれる母音 (vowel) の数をカウントする手続きを示します．仮引数の長さを得るのに組込関数 LEN を用い，それぞれの文字が「AEIOU」かそれ以外かを調べるために INDEX 関数を用いています．

```
      INTEGER FUNCTION VOWELS(STRING)
      CHARACTER*(*) STRING
      VOWELS = 0
      DO 25, K = 1,LEN(STRING)
         IF( INDEX('AEIOU ', STRING(K:K)) .NE. 0) THEN
              VOWELS = VOWELS + 1
         END IF
25    CONTINUE
      END
```

この関数は名前の最初の文字から決まるディフォルトの型とは異なる型を持つため，この関数を参照するプログラム単位で INTEGER 文を用いて型宣言する必要があることに注意してください．

長さを自動的に引き渡す仕組みは，実引数の長さがわからない汎用のプログラムを書くためだけではなく，わざわざ仮引数の長さを指定して固定長とする十分な理由がない限り，あらゆる場合で利用すべきです．

長さの渡される仮引数には，代入文以外で使われる連結演算子 // のオペランドとして使うことができないという制限があります．名前付き定数でも長さ指定に *(*) と同じ表記法を用いることができますが，その意味は全く異なっており，名前付き定数はこの制限を受けません．

9.6 配列を引数にする場合

手続きの仮引数が配列のときには実引数は次のどちらかです．

- 配列名（添字なし）
- 配列要素

最初の形式は配列全体を渡します．2 番目の形式は指定された要素を始点とする配列の一部を渡します．これについては後で詳しく説明します．

単純かつ多くの場合に必要とされることは手続きの中で配列全体を利用できるようにすることです. もし，実引数の配列がいつも同じ大きさであるならば，手続きの仮配列の上下限を固定値にすることができます. 例えば,

```
      SUBROUTINE DOT(X, Y, Z)
*Computes the dot product of arrays X and Y of 100 elements
* producing array Z of the same size.
*X, Y の 100 要素の積を求め，Z に代入する.
      REAL X(100), Y(100), Z(100)
      DO 15, I = 1,100
         Z(I) = X(I) * Y(I)
15    CONTINUE
      END
```

この手続きは，他のプログラム単位の中から次のように使うことができます.

```
      PROGRAM PROD
      REAL A(100), B(100), C(100)
      READ(UNIT=*,FMT=*)A,B
      CALL DOT(A, B, C)
      WRITE(UNIT=*,FMT=*)C
      END
```

このやり方は配列の大きさが変わると機能しないため，配列の大きさが変わらない場合のみ使うことができます.

9.6.1 整合配列

手続きを任意の大きさの配列に使えるように一般化することができます. この目的には整合配列 (adjustable array) 宣言を使います. 仮配列の添字の上下限には手続きの引数（あるいは共通ブロックのメンバ）として渡された整数型の変数を含めることができます. 次はこの実例を示したものです.

```
      SUBROUTINE DOTPRO(NPTS, X, Y, Z)
      REAL X(NPTS), Y(NPTS), Z(NPTS)
      DO 15, I = 1,NPTS
* etc.
```

この例では呼び出しは次のようになるでしょう.

```
      CALL DOTPRO(100, A, B, C)
```

整合配列は配列が仮引数である（すなわち仮配列）ときだけ使うことができます. これは，対応する実引数の配列が呼び出し側，あるいは，それより上位のプログラム単位ですでにメモリ割り当て (allocate) されているためです. この方法は，次の例のように，多次元配列，上限値と下限値が与えられる場合に対して拡張できます.

```
      SUBROUTINE MULTI(MAP, K1, L1, K2, L2, TRACE)
      DOUBLE PRECISION MAP(K1:L1, K2:L2)
      REAL TRACE(L1-K1+1)
```

もちろん，整合配列はすべての型の配列に使うことができます．整合配列は別の手続きに実引数として（必要ならば渡された配列の添字の上下限とともに）渡すことができます．

仮配列の添字の上下限を表す整数式には，定数だけでなく手続きに引数として，あるいは共通ブロックとして渡された整数型変数を含めることができます．このように指定される配列の添字の範囲は 1 以上でなければならず，また，対応する実引数の配列の大きさを超えてはなりません．

もし，暗黙の型として整数型にならない名前をもつ整数型変数（あるいは名前付き定数）が配列の添字の範囲指定に使われる場合には，これに先だって INTEGER 文で整数型の型宣言をしておかなければなりません．

9.6.2 擬寸法仮配列

手続きが実行されたときに実際の配列の大きさが不明のため，固定値の大きさの配列や整合配列を使うのが不可能である状況もあります．

このような場合には擬寸法仮配列 (assumed-size array) を使うことができます．これも手続きの仮引数として渡された配列（仮配列）だけに許されており，次の例のように，大きさが不明，あるいは，不定の配列として宣言することができます．

```
REAL FUNCTION ADDTWO(TABLE, ANGLE)
REAL TABLE(*)
N = MAX(1, NINT(SIN(ANGLE) * 500.0))
ADDTWO = TABLE(N) + TABLE(N+1)
END
```

この例では，手続きの中では配列 TABLE が添字の下限値 1 の一次元配列ということだけが知られており，これが N と N+1 の要素に正しくアクセスするのに必要な情報のすべてです．手続きの実行に際しては，プログラマの責任で ANGLE の値が範囲外の値にならないようにしなければなりません．これが擬寸法仮配列を用いる場合のリスクです．コンパイラは擬寸法仮配列の添字の上限値の知ることができないため，一般に配列の添字範囲のチェックが可能なシステムであっても，擬寸法仮配列のチェックはできません．擬寸法仮配列は最後の添字のみをアスタリスク「*」にすることができ，その他の添字は（もし存在するならば）通常の規則に従わなければなりません（あるいは整数型引数を使った整合寸法とします）．

擬寸法仮配列は最後の添字（一次元の場合はその唯一の添字）の上限をアスタリスクにすることで宣言されます．（もし存在するならば）他の添字の範囲は通常の局所配列や整合寸法配列の規則に従わなければなりません．

擬寸法仮配列には重要な制限が一つあります．それは，添字の付かない形では入出力文（READ 文の入力並び (input list) や WRITE 文の出力並び (output list) など）には使えないということです．これは手続きをコンパイルするときに，コンパイラが配列の大きさを知ることができないためです．

9.6.3 部分配列

Fortran の規則では配列の添字（多次元の場合にはそれぞれの添字）の範囲について，実引数の配列は，仮配列のそれと少なくとも同じだけの範囲を持たなければならないことになっています．しかし，下限値と上限値がそれぞれ一致している必要はありません．例えば，

```
PROGRAM CONFUS
REAL X(-1:50), Y(10:1000)
READ(UNIT=*,FMT=*) X, Y
```

```
      CALL OUTPUT(X)
      CALL OUTPUT(Y)
      END
      SUBROUTINE OUTPUT(ARRAY)
      REAL ARRAY(50)
      WRITE(UNIT=*,FMT=*) ARRAY
      END
```

を実行すると，X(48) が ARRAY(50) に相当するため，まず，X(-1) から X(48) が出力され，その後，Y(10) から Y(59) が出力されます．このサブルーチンは二次元配列の一部 (slice) に対しても同様に機能します．

```
      PROGRAM TWODIM
      REAL D(100,20)
* ...
      NSLICE = 15
      CALL OUTPUT(D(1,NSLICE))
```

この例では配列の D(1,15) から D(50,15) までの部分がファイルに出力されます．この動作を理解するためには，Fortran の配列は最初（左）の添字が最も速く変化するように記憶されること，引数の結合は指定された要素のアドレスを仮配列の先頭のアドレス (base address) とすることの 2 点を知っていれば十分です．

ただし，仮引数が配列であるときに実引数として配列要素を使うことは誤解の原因になるので，配列の部分を渡すことはできる限り避けるべきでしょう．

9.6.4 文字型配列

仮引数が文字型の配列であるときには，文字型変数の場合と同じように長さを渡すことができます．仮配列のすべての要素が実引数から渡された長さを持つことになります．

例えば，文字列の配列を順に並べ変えるサブルーチンでは，次のように宣言されます．

```
      SUBROUTINE SORT(NELS, NAMES)
      INTEGER NELS
      CHARACTER NAMES(NELS)*(*)
```

一方，実引数が文字型変数，あるいは部分列の場合もあり得ます．この場合には，通常，長さを渡す方法は使わない方が実際的でしょう．例えば，実引数が次のように宣言されているとき，

```
      CHARACTER*80 LINE
```

次のように 20 文字-4 要素の配列として定義しているサブルーチンに渡すことができます．

```
      SUBROUTINE SPLIT(LINE)
      CHARACTER LINE(4)*20
```

ただし，これは正しい Fortran プログラムではありますが，手続きの呼び出しによって変数などの形式を大きく変更してしまうもので，決してお勧めできるプログラミングテクニックではありません．

9.7　手続きを引数にする場合

Fortran では手続きを他の手続きの実引数にすることができます．これは強力な機能ですが，ほとんどのプログラマがあまり利用していません．普通，手続きは異なったデータに対して同じ操作を行うときに用いられますが，同じ操作をさまざまな関数に対して行いたい場合もあります．この例として，関数の勾配を求める場合，曲線の下の面積を求める場合，グラフを描く場合などが考えられます．曲線が点の集合として与えられるときには，単に配列として渡すことができます．しかし，計算手順 (algorithm) が与えられている場合には，その手順に従って評価を行う手続きそのものを引数として渡すことができます．

次の例のサブルーチン GRAPH は引数として与えられた範囲を 100 等分して関数 MYFUNC を描きます．簡単のために現在位置から位置 (X,Y) まで線を引くサブルーチン PLOT があるものと仮定しています．実際には描画のためには他にも多くのサブルーチンが必要になります[33]．

```
      SUBROUTINE GRAPH(MYFUNC, XMIN, XMAX)
*Plots functional form of MYFUNC(X) with X in range XMIN:XMAX.
*X が XMIN:XMAX の範囲で MYFUNC(X) の関数形を描画する．
      REAL MYFUNC, XMIN, XMAX
      XDELTA = (XMAX - XMIN) / 100.0
      DO 25, I = 0,100
          X = XMIN + I * XDELTA
          Y = MYFUNC(X)
          CALL PLOT(X, Y)
25    CONTINUE
      END
```

手続き GRAPH は，呼び出しの最初の引数に関数名を与えることでその関数を描画します．その他に必要なことは，次の例のように，実引数として与える関数名を INTRINSIC 文（組込関数を与える場合）あるいは EXTERNAL 文（外部関数を与える場合）で宣言しておかなければならないことだけです．

```
      PROGRAM CURVES
      INTRINSIC SIN, TAN
      EXTERNAL MESSY
      CALL GRAPH(SIN, 0.0, 3.14159)
      CALL GRAPH(TAN, 0.0, 0.5)
      CALL GRAPH(MESSY, 0.1, 0.9)
      END
      REAL FUNCTION MESSY(X)
      MESSY = COS(0.1*X) + 0.02 * SIN(SQRT(X))
      END
```

上の例では，まず sin 関数，tan 関数をそれぞれ異なった範囲で描き，次に，外部関数 MESSY を描きます．当然のことながら，これらの関数は同じインターフェースを持ち，サブルーチン GRAPH が正しく関数を呼び出せなければなりません．

この方法で関数だけでなくサブルーチンも実引数として渡すことができます．両者の違いは仮手続きを実行するときに関数の参照ではなく，CALL 文を使うことだけです．手続きをさらに別の手続きに複数回渡していくことも可能です．最後の例に続けて，もう一段深いレベルへ渡す例を示します[34]．

[33] 訳注：標準の Fortran には作図の機能はありません．Fortran で作図するにはグラフィック用ライブラリをリンクする必要があります．

[34] 訳注：原著では EXTERNAL MESSY になっています．

```
      PROGRAM CURVE2
      EXTERNAL PRETTY
      INTRINSIC SIN, TAN
      CALL GRAPH2(PRETTY)
      CALL GRAPH2(TAN)
      END
      SUBROUTINE GRAPH2(PROC)
      EXTERNAL PROC
      CALL GRAPH(PROC, 0.1, 0.7)
      END
```

手続き GRAPH2 は範囲を指定して，手続き名を GRAPH に渡しています．英字名 PROC は仮引数として使うため，EXTERNAL 文で宣言しておかなければなりません．この場合には，最初の呼び出しの実引数が組込関数か外部関数かに拘わらず，EXTERNAL 文で宣言しなければなりません．INTRINSIC 文と EXTERNAL 文の書き方は 9.12 節で説明します．

9.8 SUBROUTINE 文と CALL 文

この2つは互いに一致するようにしなければならないため，一緒に説明する方が便利です．SUBROUTINE 文の一般的な形は次のようになります．

SUBROUTINE *name*（ *dummy1, dummy2, ... dummyN* ）

or

SUBROUTINE *name*

2 番目の形式に示すように，引数がない場合には丸括弧は省略可能です．

サブルーチンの英字名は大局名になります．そのプログラム単位でその名前を使うことはできませんし，実行プログラム中では他の大局的な項目の名前として使うこともできません．

仮引数は単なる英字名です．仮引数が解釈される方法は次節で説明します．

CALL 文の一般的な形式は SUBROUTINE 文と似た形式を持っています．

CALL *name*（ *arg1, arg2, ... argN* ）

or

CALL *name*

CALL 文でも引数がない場合には丸括弧を省略できます．

name はサブルーチン名（あるいはサブルーチンの仮手続き名）です．*arg1, arg2, ... argN* は実引数で，変数，配列，部分列，配列要素，任意の式を使うことができます．その形式は，前節までに説明したように，対応する仮引数の形式やサブルーチンの中での仮引数の使われ方に依存します．

9.9 RETURN 文

RETURN 文 (RETURN statement) は次のように単にそのキーワードを書くだけです．

```
      RETURN
```

その働きは手続きの実行を終了し，呼び出したプログラム単位に制御と戻り値を返すことです．プログラム単位の最後の END 文を実行しても全く同じ結果になりますので，それぞれ 1 つの入り口 (entry) と終了箇所を持つ手続き（手続きの理想です）では省略することができます．しかし，RETURN 文を使って異常終了させる方がよい場合もあります．次の例はこのことを示すための非常に単純な例です．

```
      REAL FUNCTION HYPOT(X, Y)
```

```
*Computes the hypotenuse of a right-angled triangle.
*直角三角形の斜辺の長さを求める.
      REAL X, Y
      IF(X .LE. 0.0 .OR. Y .LE. 0.0) THEN
          WRITE(UNIT=*,FMT=*)'Warning: impossible values'
          HYPOT = 0.0
          RETURN
      END IF
      HYPOT = SQRT(X**2 + Y**2)
      END
```

この関数は他のプログラム単位で例えば次のように使われます.

```
       X = HYPOT(12.0, 5.0)
       Y = HYPOT(0.0, 5.0)
```

この場合，最初の関数の参照によって X に 13.0000 の値が代入された後，2 番目の関数の参照で警告メッセージが表示されるとともに Y に 0 が返されます.

上の外部関数の例では，次に示すように，2 つの終了箇所のある問題を解消して，1 箇所で終了するように書き換えることができます.

```
      IF(X .LE. 0.0 .OR. Y .LE. 0.0) THEN
          WRITE(UNIT=*,FMT=*)'Warning: impossible values'
          HYPOT = 0.0
      ELSE
          HYPOT = SQRT(X**2 + Y**2)
      END IF
      END
```

しかし，実際には計算部分が上記のように 1 文で済むことはほとんどなく長くなるため，計算のほとんどすべての部分を ELSE ブロックに入れてしまうよりも，RETURN 文を使った方が動作がわかりやすくなります．異常終了の処理を行うためのもう一つの選択肢は単純 GO TO 文を使って END 文へジャンプしてしまう方法です.

9.10 FUNCTION 文

すべての外部関数の最初の文は FUNCTION 文でなければなりません．その一般的な形は次のようになります.

type FUNCTION(*dummy1, dummy2, ... dummyN*)

type の宣言は省略可能です．省略した場合には暗黙の型の規則に従って戻り値の型が決められます．関数は *type* によって宣言された型，あるいはそのプログラム単位内の IMPLICIT 文（従って FUNCTION 文の後に書かれる）に従って決められる型を持つこともできます.

文字型の関数の場合にはその長さを直定数 (literal constant) で宣言するか（名前付き定数は不可），CHARACTER*(*) の形で与えます．後者では，呼び出し側のプログラム単位でその関数名の長さとして宣言された長さが自動的に渡されます.

仮引数はない場合も含めていくつでも置くことができます．ただし，仮引数がない場合にも丸括弧は省略できません．仮引数は 9.4 節で説明したように，変数，配列，手続きのどれかでなければなりません.

関数名は関数副プログラムの中では変数のように使うことができます．関数が呼び出し側に制御を戻す前にこの変数に戻り値を代入しておかなければなりません．文字型の関数名で長さが渡されるものとして宣言されている場合には，代入文中以外の連結演算子のオペランドとして使うことはできません．同じプログラム単位からの呼び出しでは長さは必ず同じでなければならないため，文字型関数に長さを渡す方法は引数の場合と比べてあまり利用価値はありません．例えば次の例[35]

```
      PROGRAM FLEX
      CHARACTER CODE*8, CLASS*6, TITLE*16
      CLASS = CODE('SECRET')
      TITLE = CODE('ORDER OF BATTLE')
      END

      CHARACTER*(*) FUNCTION CODE(WORD)
      CHARACTER WORD*(*), BUFFER*80
      DO 15, K = 1,LEN(WORD)
           BUFFER(K:K) = CHAR(ICHAR(WORD(K:K)) + 1)
15    CONTINUE
      CODE = BUFFER
      END
```

では，関数自体は 80 文字までエンコードすることができるにも拘わらず，FLEX プログラムから呼ばれる場合には 8 文字しか返すことができないため，それよりも長い文字列が与えられた場合には期待した結果が得られません．この制約は，サブルーチンを用いて，長さを渡す引数として別の引数を新たに設けて戻り値を返させることで回避できます．

引数のない関数にはそれほど広い用途はありませんが，乱数を発生させる場合や入力ファイルから数値を読み込む場合（次の例）などがその適用対象となる問題として挙げられます．

```
      PROGRAM COPY
      REAL NEXT
      DO 10,I = 1,100
            WRITE(UNIT=*,FMT=*) NEXT()
10    CONTINUE
      END

      REAL FUNCTION NEXT()
      READ(UNIT=*,FMT=*) NEXT
      END
```

関数の呼び出しでは変数と見分けるために丸括弧が必要です．また，おそらく関数の呼び出しに合わせるためと思いますが，関数定義文（FUNCTION 文）でも空の丸括弧が必要です．

9.11 SAVE 文

SAVE 文 (SAVE statement) は，連続した手続きの呼び出し間で変数や配列の値を保存しておく（ことを保証する）ための宣言文です．普通の状態では手続きから呼び出し側に制御が返されたときに局所的な項目（変数など）は「不定 (undefined)」になります．特別な項目では手続きが一度呼ばれたとき

[35] 訳注：ABC→BCD のように文字列を変換する例です．

の値を保持しておくと次の呼び出しで余計な作業を省略できることがあるため，この宣言文は様々な場面で便利に使えます．

```
SUBROUTINE EXTRA(MILES)
INTEGER MILES, LAST
SAVE LAST
DATA LAST /0/
WRITE(UNIT=*, FMT=*) MILES - LAST, ' more miles.'
LAST = MILES
END
```

この例では，それぞれの呼び出しで与えられる引数 MILES の値を保持し，次の呼び出しで与えられる値から保持した値を引いて，その増分を出力しています．最初の呼び出しで不定値を使わないようにするために，DATA 文を使って LAST に初期値を与えておかなければいけません．

局所変数や局所配列，名前付き共通ブロック全体は SAVE 文を使って保持することができますが，仮引数や共通ブロックに属している変数や配列は保持することができません．

SAVE 文には次の二種類の形式があります．

SAVE *item, item, ... item*
SAVE

ここで，*item* は局所変数,（添字の付いていない）配列，スラッシュで囲まれた共通ブロック名です．二番目の形式ではプログラム単位中の保持可能なすべての項目の値が保持されます．二番目の形式は，そのプログラム単位で使われているすべての共通ブロックが主プログラムで使われている場合，あるいは，それらの共通ブロックが使われているすべてのプログラム単位で SAVE 文で宣言されている場合以外には，使うべきではありません．SAVE 文は主プログラムでも使うことができます（このため，他の宣言文と一緒に INCLUDE ファイルにまとめておくことができる）が，これには何の機能もありません．

現在の多くの Fortran システムが単純な静的メモリ割り当て方式 (static storage allocation scheme) を採っているため，SAVE 文の有無に拘わらずすべての変数の値が保持されます．しかし，Disc Overlay を用いる小規模なコンピュータや仮想記憶 (virtual memory system) を用いる大規模なコンピュータには当てはまらないでしょう．SAVE 文はプログラムを安全かつ可搬的にするためのものですので，なるべく SAVE 文を使うように配慮すべきです．たとえ，お使いのシステムが厳密な取り扱いをする安全な（冗長な）システム上であっても，プログラムを読む側に手続きの前の呼び出しから次の呼び出しの間に情報が保存されることで手続きがうまく機能していることを示すことができるため，SAVE 文には依然として大きな価値があります．

9.12　EXTERNAL 文と INTRINSIC 文

EXTERNAL 文は指定されたプログラム単位を実行するために必要な外部手続きに名前をつけるものです．EXTERNAL 文には外部関数やサブルーチンの名前を指定することができます．EXTERNAL 文は次の 3 通りの多少異なった状況で必要になります．

- 外部手続きあるいは仮手続きがその他の手続きの呼び出しの実引数であるとき
- 組込関数と重複した名前を持つ手続きを呼ぶとき
- 名前付き初期値設定副プログラム (named block data subprogram) が実行プログラムに確実にリンクされるように用いる．この使用法については 12.4 節で述べます．

INTRINSIC 文は組込関数 (intrinsic function) の名前を宣言するために使います．通常，INTRINSIC 文は組込関数を他の手続きの呼び出しの実引数にするときのみ必要になりますが，その他の用途として，非標準の組込関数を使うときに，それと同じ名前を持つ外部関数があるとどちらを使うのか曖昧になりますが，その場合にはこの宣言を用いて組込関数を使うことを明示するとよいでしょう．

EXTERNAL 文と INTRINSIC 文は次のようにどちらも同じ形式を持ちます．

```
EXTERNAL  ename, ename, ... ename
INTRINSIC  iname, iname, ... iname
```

ここで，*ename* は外部関数，サブルーチン，あるいは仮手続き，また，*iname* は組込関数の個別名 (specific name) でなければなりません．たとえば，実数型と倍精度実数型の三角関数を実引数として使うためには次の宣言が必要です．

```
INTRINSIC SIN, COS, TAN, DCOS, DSIN, DTAN
```

上の宣言によって，関数名 SIN を実引数に用いたときには，実数型の sin 関数が参照されます．宣言のない場合には SIN は総称名としての属性を持ったままになります．手続きを実引数として方法については 9.7 節，組込関数のリストは巻末 Appendix をご覧ください．

10　入出力の機能

Fortran の入出力の機能は比較的強力で，柔軟性があり，明確なものです．入出力の機能を多く使ったプログラムでも可搬性を維持することができ，装置に依存しないプログラムを書くことができます．このことは他の多くの言語では不可能ではないにしても難しいことには違いありません．もちろん，ハードウェアやオペレーティングシステムの違いによる影響をまったく無視するわけにはいきませんが，この影響はファイル名の形式や使える最大のレコード長などきわめて限られたものです．

よく使われる READ 文と WRITE 文は次のようなものです．

```
READ(UNIT=*, FMT=*) NUMBER
WRITE(UNIT=13, ERR=999) NUMBER, ARRAY(1), ARRAY(N)
```

READ や WRITE の後の丸括弧は制御並び (control-list) と呼ばれ，データの転送される場所と送り方（受け取り方）を指定するものです．この並びはキーワードとともに指定するのが普通です．制御並びの後に，読み込みや書き出しの対象になるデータ項目 (data item) のリストを書きます．

その他の入出力文 (input/output statement) も制御並びのみを持つこと以外は同じ形式で書かれます．その他の入出力文とはファイル操作のための文で OPEN 文，CLOSE 文，INQUIRE 文や，ファイルの現在位置を変更する REWIND 文，BACKSPACE 文などのことです．

これらの文について詳しく説明する前に，その考え方と必要な用語について説明しておきます．

10.1　ファイル，装置番号，記録

Fortran でいうファイル (file) とは READ 文や WRITE 文で扱われるすべてを指し，磁気ディスクや磁気テープに記憶されるデータファイルだけでなく，プリンタや端末などの周辺装置 (peripheral device) も含んでいます．厳密に言えば，これらのファイルは，内部ファイル (internal file) と区別するため，外部ファイル (external file) と呼ばなくてはなりません．

内部ファイルはプログラムの実行時に一時ファイル (temporary file) として使われる文字型変数あるいは文字型配列に過ぎません．内部ファイルは READ 文や WRITE 文とともに，文字情報を書式仕様 (format specification) を用いて扱うために使われます．それ以外の入出力文には使うことはできません．

外部ファイルはあらかじめ入出力装置 (I/O unit) に「接続」されていなければなりません．入出力装置 (I/O unit) の装置番号は 0 からシステムによって定められた上限値（通常 99）までの範囲の自由に

決めた整数値です．OPEN 文と INQUIRE 文以外は，この装置番号を通してファイルが参照されます．

OPEN 文はファイルと装置番号を接続します．普通は，ファイルが既存のものか，新たに生成するものかを，次のように指定します．

```
OPEN(UNIT=1, FILE='B:INPUT.DAT', STATUS='OLD')
OPEN(UNIT=9, FILE='PRINTOUT', STATUS='NEW')
```

簡単のためにこの章の例では装置番号に整数値を用いますが，装置番号に名前付き定数や変数を用いるとプログラムの独立性が高まり融通が利くようになります．

入出力装置は大域的な資源 (resource) です．ファイルはどのプログラム単位からでもオープンすることができ，一旦オープンされるとその装置番号を使ってどのプログラム単位からでも入出力の操作ができます．装置番号は整数型の変数に保持することができ，引数として手続きに渡すことも可能です．

一旦ファイルと装置番号の接続が確立すると，次のどれかに該当するまで接続が維持されます．

- (STOP 文や END 文で）プログラムが正常終了した場合
- OPEN 文によって，同じ装置番号に新たに別の名前のファイルが接続された場合
- その装置に CLOSE 文が実行された場合

プログラムが終了するときにはすべてのファイルがクローズされますが，必要な入出力が終わったら直ちに，CLOSE 文を用いてクローズする方がよいでしょう．もしエラーが生じたりユーザがプログラムを停止した場合などプログラムが異常終了した場合には，オープンしているファイル，特に出力ファイルに不完全，あるいは無意味な記録 (record) が書き出されたままになります．

INQUIRE 文を用いると，外部ファイルや装置番号の現在の状態を取得することができます．INQUIRE 文はさまざまな状態で実行されるような，ライブラリに含める手続きを書くようなときに特に便利です．INQUIRE 文によって，例えば，どの装置番号が使われていないか，あるファイルが存在するか，もし存在するならばどのような属性を持つかなどを調べることができます．

10.1.1 記録

ファイルは一連の記録 (record) の集まりとして構成されています．テキストファイルでは 1 つの記録は 1 行に相当し，その他の形式のファイルの場合には，記録はそのプログラムにとって便利なように値を集めたもので，物理的な意味はありません．記録を磁気ディスクのセクタや磁気テープのブロックに対応づける必要はありません．READ 文や WRITE 文は記録の先頭から処理を行い，記録のすべての値に必要な転送を行います．

Fortran の規則には記録の長さに上限はありませんが，オペレーティングシステムが上限値を持つ場合があります．この制限は記録の形式の違いによっても異なるでしょう．

10.1.2 書式付き記録と書式なし記録

外部ファイルはその記録が書式付き (formatted) か書式なし (unformatted) かによって二種類に分けられます．書式付き記録はデータを文字情報として，すなわちテキストの行として記録します．この形式は内容にもよりますがコンピュータだけでなく人間にも読めるため，広範囲に使うことができます．プログラマにとって面倒なことは，いかにコンピュータの内部形式からこの形式に，あるいはその逆に変換するかを，それぞれの WRITE 文や READ 文で指定しなければならないことです．普通この変換は書式指定 (format specification) で行います．

書式なし記録はコンピュータの内部形式のまま記録しますので，書式の変換は行われません．この形式には，数値データ，特に浮動小数点の数値データのファイルに対していくつかの利点があります．

書式なしの転送はプログラムが簡単で，実行が速く，丸め誤差がありません．さらに，生成されるファイルはバイナリファイル (binary file) とも呼ばれ，一般にファイルサイズが小さくなります．例えば，実数値をその精度を維持して書式付き記録に保存するには 10 文字から 15 文字の文字列に変換する必要がありますが，普通は実数型は 4 バイトであるため，書式なし記録に保存する場合にはそれと同じ 4 文字分しか使いません．書式なし記録の欠点はシステムに強く依存してしまうことです．通常その記録は人間や他の機種のコンピュータには読めませんし，同じコンピュータ上の他のプログラミング言語で生成されたファイルとも互換性がないこともあります．書式なしファイルは同じタイプのコンピュータ上の Fortran で読み書きする場合のみ使うべきでしょう．

10.1.3　逐次アクセスと直接アクセス

すべての周辺機器はファイルを先頭から次の記録へと順々に処理していくこと，すなわち逐次処理を行うことができます．逐次ファイル (sequential file) の最も重要な利点は記録ごとに長さが違ってよいことで，最小の長さは 0，最大の長さはシステムに依存します．

逐次ファイルは常に次に転送される記録を指している「ポインタ」のようなものがファイルに付いているようにふるまいます．端末やプリンタなどの装置では読み書きが完全にシーケンシャルに行われますが，ファイルをディスクやテープに記録するときには，REWIND 文を用いてこのポインタをファイルの先頭に戻して，読み直したり再出力したりすることができます．ファイルによっては BACKSPACE 文を用いてポインタを一つ戻して，最後の記録を読み直したり上書きしたりすることもできます．

残念ながら Fortran 規格には，既存の逐次ファイルがオープンされたときのそのポインタの位置について定義されていません．ほとんどの Fortran システムでは実用性を考えて，ファイルの先頭から処理が行われるようになっていますが，中には OPEN 文の後に REWIND 文を使ってまず先頭に戻さなければならないような困ったシステムもあります．もう一つの問題は既存の逐次ファイルに新しい記録をどのように加えていくかということです．OPEN 文に（拡張として）append（追加）オプションが使えるシステムもありますが，標準 Fortran を使う場合の最良の方法はファイルをオープン後，ファイルの終端 (end-of-file) を検出するまで記録を読んでしまい，その後に BACKSPACE 文でポインタを一つ戻して end-of-file の状態をクリアしておくことです．このようにしておくと，後は普通に出力すれば，新しい記録を加えていくことができます．

もう一つのアクセス方法は任意の順に読み書きができる直接アクセス (direct-access) です．ほとんどのシステムは磁気ディスクなどの直接アクセス装置に記憶されるファイルに対してのみ直接アクセスを許しています．テープ装置に対して使える場合もあります．システムが記録番号 (record number) から記録の位置を割り出せるように，直接アクセスファイルのすべての記録は同じ長さでなければなりません．記録の長さは一度決められてファイルが生成されると（ほとんどのシステムで）ファイルがある限り変更することはできません．Fortran では直接アクセス記録は 1 から昇順に番号が付けられており，転送を行う記録の番号を READ 文や WRITE 文で指定します．

直接アクセスのファイルには任意の順序で記録を出力することができます．もし記録があるならば，すなわち，ファイルが生成されてからその時までに記録が書き出されていれば，これらの記録から任意に読み出すことができます．一度記録が書き出されるとそれを消去することはできませんが，その内容を更新，すなわち書き直すことはいつでも可能です．

原始的なオペレーティングシステムには，直接アクセスのファイルの最大の長さをファイルがオープンされるときに指定する必要があるものがあります．しかし，Fortran 規格に完全に準拠しているシステムではこのような必要はありません．

10.2　外部ファイル

書式付き記録と書式なし記録を同じファイル混在させることはできません．また，ほとんどのシステムで直接アクセスとして生成されたファイルと逐次アクセスファイルは区別されます．このため，4種類の外部ファイルが存在することになります．標準 Fortran では，それ以外の例えばインデックス付き逐次ファイルや name-list ファイルなどの形式はサポートされていません．

10.2.1　書式付き逐次ファイル

これはテキストファイルと呼ばれるものです．端末やプリンタは常に書式付き逐次ファイルとして扱われます．この形式のデータファイルは例えば，直接キーボードから入力したり，テキストエディタを利用するなど，様々な方法で生成することができます．Fortran によっては通常の行の長さ（例えば 132 文字）を超える記録は使えないものもあります．あらかじめ接続されているテキストファイルでない限り，OPEN 文を用いてファイルをオープンする必要があります．しかし，FORM=や ACCESS=などのキーワードはディフォルト値が適当であるならば次の例のように省略することができます．

```
OPEN(UNIT=4, FILE='REPORT', STATUS='NEW')
```

すべてのデータ転送は書式制御 (format control) の元行われます．この形式のファイルには，プログラマが書式指定 (format specification) を行う方法と並びによる書式指定 (list-directed formatting) のどちらかを使って入出力を行います．

並びによる入出力のよい点は，Fortran にすべて任せてしまうことでプログラミングの手間をかけずに簡単にデータ転送ができることです．この指定は書式識別子 (format identifier) としてアスタリスクを書いておきます．

```
WRITE(UNIT=*, FMT=*)'Enter velocity: '
READ(UNIT=*, FMT=*, END=999) SPEED
```

並びによる入力は，端末から数値を読み込むときに「自由書式 (free format)」のデータ入力が可能なため，特に便利です．レイアウトが十分に整えられておらず，書式指定でうまく扱えないデータファイルから読み込むときにも便利に使えます．

並びによる出力は上の例のように単に文字列を出力する場合には十分ですが，数値を出力する場合には，行中の出力位置や出力桁数，小数部の桁数などを制御する方法がないため，あまり満足できる出力は得られません．例えば，

```
WRITE(UNIT=LP, FMT=*)' Box of',N,' costs ',PRICE
```

とすると次のような結果になってしまいます．

```
 Box of           12 costs          9.5000000
```

並びによる出力は，簡単なメッセージやプログラムの開発時の確認メッセージの出力以外には使わない方がよいでしょう．並びによる書式指定に関する規則は 10.10 節で説明します．

もう一つの方法は書式指定によってデータ転送を完全に制御する方法です．先ほどの例は書式指定を用いて例えば次のように修正すると，

```
      WRITE(UNIT=LP, FMT=55)'Box of',N,' costs ',PRICE
55    FORMAT(1X, A, I3, A, F6.2)
```

次のような出力が得られます． `Box of 12 costs  9.50`

この例では書式指定を FORMAT 文で行っています．FORMAT 文の文番号が WRITE 文の書式識別子 (format identifier) になります．書式指定を行う別の方法は 10.6 節で説明します．

書式指定で制御された入力の妙な点は，実際のデータの記録長に拘わらずに，読み込む行の右側に必要な数だけ空白文字が補われることです．このことによっていろいろな現象が生じますが，中でも

空の記録と空白で埋められた記録の違いを区別できないことになります．このため，もし空の記録から数値を読み込むとその結果は単に 0 になります.

10.2.2　書式なし逐次ファイル

書式なし逐次ファイル (unformatted sequential file) は別のプログラムにデータを渡すときによく使われます．またこの形式は，スクラッチファイル (scratch file)，すなわちプログラムの実行中に使われる一時ファイルにも適しています．書式なし記録の長さの制限はオペレーティングシステムによるものですが，ほとんどのシステムでは少なくとも数千のデータ数を一つの記録に含めることができます．OPEN 文にはファイルの書式を指定しなければなりませんが，ディフォルト値は “squential” です．READ 文や WRITE 文は一文で一つの書式なし記録を転送します.

次の例では，既存の書式なしファイルをオープンし 2 つの記録を読み込みます.

```
OPEN(UNIT=15, FILE='BIN', STATUS='OLD', FORM='UNFORMATTED')
READ(15) HEIGHT, LENGTH, WIDTH
READ(15) ARRAYP, ARRAYQ
```

一般的には BACKSPACE 文や REWIND 文は書式なし逐次ファイルに用いられます.

10.2.3　書式なし直接アクセスファイル

直接アクセスファイルはコンピュータだけが読むことができるため，効率を上げるためには書式なし記録を用いるのがよいでしょう．OPEN 文で ACCESS='DIRECT' とするとともに，記録長を指定する必要があります．残念ながら記録長の単位は規格に規定されておらず，一つの記録が保持できるバイト数と記録単位 (storage unit, 5.1 節参照)，すなわち実数型あるいは整数型のデータの個数を用いるシステムがあります．これは可搬性を妨げる一因となり，数値記録単位 (numerical storage unit) に相当するバイト数を知らなければならないということを意味します．ほとんどシステムで，ファイルが作成されたときと同じ記録長が指定された場合にのみ，既存のファイルをオープンすることを許しています.

それぞれの READ 文や WRITE 文は正確に 1 記録を転送し，その記録の番号（1 から昇順に番号が付される）を指定しなければなりません．記録長は OPEN 文で宣言した長さを超えてはなりません．もし，指定した記録長に満たない場合は残りの部分は不定値になります.

直接アクセスファイルの使い方の説明として，在庫記録などの簡単なデータベースを処理するプログラムを示します．記録長は 4 バイトの数値記録単位 (numerical storage unit) で与えられるものと仮定しています．このとき必要な記録長は次のように計算できます.

NAME	文字型 (CHARACTER*10) 1 つ	10 文字 = 10 バイト
STOCK	整数型 1 つ	1 単位=4 バイト (byte)
PRICE	実数型 1 つ	1 単位=4 バイト (byte)

全記録長は 18 バイトあるいは 5 数値記憶単位（切り上げられています）になります.

```
      PROGRAM DBASE1
      INTEGER STOCK, NERR
      REAL PRICE
      CHARACTER NAME*10
*Assume record length in storage units holding 4 chars each.
*記録長が記憶単位で与えられるものとし，4 バイトを 1 記憶単位と仮定する
      OPEN(UNIT=1, FILE='STOCKS', STATUS='OLD',
```

```
     $ ACCESS='DIRECT', RECL=5)
100   CONTINUE
*Ask user for part number which will be used as record number.
*ユーザに記録番号として使われる部品番号を尋ねる.
      WRITE(UNIT=*,FMT=*)'Enter part number (or zero to quit):'
      READ(UNIT=*,FMT=*) NPART
      IF(NPART .LE. 0) STOP
      READ(UNIT=1, REC=NPART, IOSTAT=NERR) NAME, STOCK, PRICE
      IF(NERR .NE. 0) THEN
          WRITE(UNIT=*,FMT=*)'Unknown part number, re-enter'
          GO TO 100
      END IF
      WRITE(*,115)STOCK, NAME, PRICE
115   FORMAT(1X,'Stock is',I4, 1X, A,' at ・, F8.2, ' each')
      GO TO 100
      END
```

このプログラムの出力は次のようになります.

```
      Stock is 123 widgets    at    556.89 each
```

このプログラムは非常に簡単に，在庫の変化に応じて記録の内容を更新することができるように拡張することができます.

10.2.4 書式付き直接アクセスファイル

書式付き直接アクセスファイル (formatted direct-access file) はその記録長が常にバイト単位であるため，書式なしのものよりも多少可搬性があります．それ以外の点はあまり言うことがありません．この形式では OPEN 文で ACCESS='DIRECT' と FORM='FORMATTED' の両方を指定するとともに，READ 文と WRITE 文では書式と記録番号を指定する必要があります．並びによる転送 (list-directed transfer) はできません．もし，指定された書式が長く，複数の記録が対応する場合には REC=で指定された記録から順に充てられていきます．記録の終端を超えて読み込もうとするとエラーになりますが，記録の一部を残して出力すると残りの部分には空白が補われます.

10.3 内部ファイル

内部ファイル (internal file) はあたかも書式付き逐次アクセスファイルのように扱われる，メモリ上に割り当てられた領域です．当然，プログラムの実行中だけ存在するものです．内部ファイルは様々な目的で使われていますが特に文字型からあるいは文字型への変換によく用いられています．Fortran の初期のバージョンでは ENCODE 文や DECODE 文が使えるものがありましたが，内部ファイルに対する READ 文（DECODE 文の代わり）や内部ファイルに対する WRITE 文（ENCODE 文の代わり）は簡単で，かつ，より柔軟で可搬的でもあります.

内部ファイルは READ 文と WRITE 文だけに使うことができ，さらに，書式指定を明示する必要があり，並びによる転送はできません．装置 (unit) は文字型である必要がありますが，文字型ならば変数や配列要素，あるいは配列全体のどれでも構いません．もし配列全体が使われたときには配列のそれぞれの要素が 1 記録 (record) に対応し，それ以外の場合にはファイルは 1 つの記録だけから構成されることになります．データの転送は常に内部ファイルの先頭から行われます．すなわち，毎回自動的に REWIND（巻き戻し）が行われるということです．記録長は文字型のアイテムの長さになります．文

字型のアイテムの長さ以上の転送は許されませんが，出力される文字数が少ない場合には空白文字が補われます．END 文と IOSTAT 文を使ってファイル終端 (end-of-file) を検出することができます．

例えば，次のような内部ファイルに対する WRITE は数値データを適当な書式で文字列に変換する目的で頻繁に用いられます．

```
      CHARACTER*8 CVAL
      RVALUE = 98.6
      WRITE(CVAL, '(SP, F7.2)') RVALUE
```

WRITE 文によって文字型変数 CVAL は' +98.60 ' になります（数値が 7 桁のフィールドに右詰されること，宣言されている 8 文字の長さになるように空白文字が補われることの 2 点から，数値の右には空白文字が 1 つあることに注意してください）．

一度，数値が文字列に変換されると，7 節で説明した様々な処理が可能になります．例えば，数値をフィールドに左詰で出力したり，銀行の取引明細のように負の数値に DR を付けたり，貸借対照表のように括弧でくくったりすることも可能です．適当な計算を行えば，8 進数や 16 進数などの 10 進数以外の表記も可能です．さらには，まず適当な書式指定を文字列に出力しておき，その書式を用いて必要な変換を行うというような，凝った変換も可能になります．

内部ファイルに対する READ 文は数値を含んだ文字列をデコード (decode) するのに使うことができます．典型的な利用方法はコマンドベース (command-driven) のプログラムのユーザの入力を扱うことです．コマンドライン[36]が 1 つの単語とそれに続くオプション（省略可能）の数値（整数値あるいは実数値の形式）で与えられるものとします（1 つ以上の空白文字を両者の区切りとします）．コマンドラインの入力は例えば次のようになるでしょう．

```
UP 4
RIGHT 123.45
```

これを扱うには，まず文字型変数に行全体を読み込んで，INDEX 関数を使って最初の空白文字の位置を調べるのが簡単です．最初の文字列は命令を表す単語であり，それに続く数字列は内部ファイルの READ 文で実数値に変換できます．

```
      CHARACTER CLINE*80
* . . .
100   WRITE(UNIT=*,FMT=*)'Enter command: '
      READ(*, '(A)', IOSTAT=KODE) CLINE
      IF(KODE .NE. 0) STOP
      K = INDEX(CLINE, ' ')
*The command word is now in CLINE(1:K-1); Assume the
* number is in the next 20 characters: read it into RVALUE
*命令を表す語は CLINE(1:K-1) まで，数値はそれ以降の 20 文字以内にあると仮定し，
*その値を RVALUE に読み込む
      READ(CLINE(K+1:), '(BN,F20.0)', IOSTAT=KODE) RVALUE
      IF(KODE .NE. 0) THEN
          WRITE(UNIT=*,FMT=*)'Error in number: try again'
          GO TO 100
      END IF
```

後続の空白文字を確実に無視するようにするため編集記述子 (edit descriptor)BN が必要なことに注意してください．このようにすると F20.0 によって 20 文字のフィールドのどこに実数値や整数値が書かれていても処理することができます．フィールド全体が空白の場合には 0 に変換されます．

[36]訳注：例えば g77 でコンパイルするときのコマンド入力 “g77 prog.f -o prog” をコマンドラインと呼びます．

10.4　接続済みファイル

10.4.1　端末に対する入出力

多くのプログラムはユーザとの対話形式のプログラムですので，ユーザの端末へのアクセスが必要です．端末は OPEN 文で接続されるファイルとして扱われますが，その名前はシステムに依存します．Fortran では標準入力ファイル (standard input file)，標準出力ファイル (standard output file) と呼ばれる特殊ファイルを用意することで，この問題を回避しています．これらのファイルはあらかじめ接続されており，OPEN 文は不要です（または，OPEN 文を使わなくてもよい）．どちらも書式付き逐次ファイルで，対話形式のシステムでは端末からの入力と出力を扱います．これらのファイルに対しては，装置指定子 (unit identifier) をアスタリク “* ” とした READ 文と WRITE 文を使うことができます．これらのファイルによって端末との入出力が簡単，かつ，可搬的になります．この使用例はこの本の至る所にあります．

プログラムがバッチモードで実行されるときには，ほとんどのシステムで標準出力はログファイル (log file) やプリンタに変更されます．標準入力ファイルに対しても同様の変更がなされることがあります．

アスタリスクを用いる記法には 1 つ小さな欠点があります．それは，装置番号は多くの場合，整数型の変数で指定されるため，入力元や出力先のファイルを別のファイルに変更するには単にこの変数の値を変えるだけで済むのに対して，アスタリスクを用いて指定した標準入力や標準出力のファイルはこの変更ができないことです．

10.4.2　その他の接続済みのファイル

Fortran66 との互換性を保つため，多くのシステムで上記以外の接続済みファイルが使えるようになっています．かつては慣習として装置番号 5 はカードリーダに，装置番号 6 はラインプリンタに接続されていました．その他の装置は磁気ディスク上の適当な名前のファイルに接続されており，例えば装置番号 39 は FTN039.DAT あるいは TAPE39 のような名前のファイルに接続されたりしていました．このような接続済み装置は完全に過去のものになっており，現在では無視すべきものです．どの番号の接続済み装置に対してでも，OPEN 文の指定が優先されます．残念ながら，このような遺物となった接続済み装置が思いもよらない副作用をもたらすことがあります．例えば出力ファイルをオープンするのを忘れていると，プログラムがエラーなく動いていても，実行結果が前述したような特殊な名前のファイルに出力されていて見つからないということもありえます．

10.5　エラーとファイル終端の条件

ほとんどの実行文では十分注意してプログラムを書けばエラーの発生を防ぐことができますが，入出力の場合にはプログラマの力の及ばないところでエラーが生じます．例えば，すでに存在しないファイルをオープンしようとしたり，空きのないディスクに書き込もうとしたり，誤った書式で書かれたデータファイルを読もうとしたりすることでエラーが生じます．入出力はこのような影響を受けやすい部分であるため，Fortran ではエラーの処理機能が用意されています．エラーを処理する方法として二つの方法があり，それぞれ独立に，あるいは，組み合わせて用いることができます．

まず，入出力の制御並び (I/O control list) に次のような項目を加えることができます．

```
IOSTAT=integer_variable
```

この文が実行されたときに，整数型変数 integer_variable（あるいは整数型配列要素）に入出力の状態を表す値が代入されます．この入出力文が正常に実行されたときには integer_variable に 0 がセットされます．それ以外の場合には，エラーが起きたときには正の値が，入力ファイルでファイル

の終端を検出したときには負の値がセットされます．この具体的な値はシステムに依存するため，可搬的なプログラムを作成するときには 0，正，負だけを調べる（次の例を参照）にとどめ，具体的な値（エラーコード）はユーザに示すことぐらいしかできません．

```
100   WRITE(UNIT=*, FMT=*)'Enter name of input file: '
      READ(UNIT=*, FMT=*) FNAME
      OPEN(UNIT=INPUT, FILE=FNAME, STATUS='OLD', IOSTAT=KODE)
      IF(KODE .NE. 0) THEN
          WRITE(UNIT=*,FMT=*)FNAME, ' cannot be opened'
          GO TO 100
      END IF
```

この簡単なエラー処理でもある程度プログラムを使いやすくすることができます．例えばファイルのオープンに失敗したときには，多くの場合そのファイルがないためですので，ユーザに別のファイル名を入力してもらうことができます．

もう一つの方法は，エラーが生じたときに飛ぶ文番号を指定する，次の形式の項目を使うことです．

```
ERR=label
```

当然飛び先は実行文で，かつ，同じプログラム単位内になければなりません．

```
      READ(UNIT=IN, FMT=*, ERR=999) VOLTS, AMPS
      WATTS = VOLTS * AMPS
* rest of program in here . . . . . and finally
* 中略，以降はプログラムの最後の部分
      STOP
999   WRITE(UNIT=*,FMT=*)'Error reading VOLTS or AMPS'
      END
```

この方法は使いやすいのですが，ジャンプを多用した構成のよくないプログラムになりやすいという GO TO 文と同じような問題があります．

同じ文に IOSTAT=と ERR=の両方を用いると，エラー番号を調べ，エラー処理のルーチンにジャンプすることができます．このどちらかのキーワードを用いると入出力エラーが生じてもプログラムの実行を続けることができます．また，多くのシステムではエラーメッセージを抑制することも可能です．

IOSTAT=と ERR=はすべての入出力文で使うことができます．プログラマは，よりロバスト，かつ，使いやすいプログラムにするために，このエラー処理の機能を最大限活用すべきです．

非常によく発生するエラーでこの方法で検出できないエラーがあります．それは，書式付き記録に数値を出力するときに，指定されたフィールド（欄）が狭すぎて出力できないと，その欄にはアスタリスクが出力されるというものです．

データの転送中にエラーが起きるとファイルの現在位置が不確定になります．大きな配列や多くの記録を転送しているときにエラーが検出されると，問題の記録を見つけ出すのは非常に困難になるかもしれません．

10.5.1　ファイル終端の検出

READ 文で逐次ファイルや内部ファイルの終端 (end-of-file/EOF) を超えて読み込もうとすると，end-of-file の状態 (end-of-file condition) になります．入出力の制御並び (I/O control list) に IOSTAT=整数型変数 の項目がある場合には，整数型変数に状態値 (status value) として負の値がセットされます．ま

た，制御並びに END=文番号 の項目がある場合には，ファイルの終端が検出されるとこの文番号に制御が移されます．

END= は READ 文だけに用いることができますが，ERR= や IOSTAT= と一緒に用いることができます．ファイル終端の検出機能は大きさのわからないファイルを読むときに非常に便利ですが，使用に際しては注意が必要です．書式付きファイルから一度に複数の記録を読み込んでいるときには，どの部分でファイルの終端を検出したのか知る方法がありません．ファイル終端を超えた部分の変数では元の値がそのまま残されます．また，直接アクセスファイルにはファイル終端の概念はありません．ファイルの「終端」を超えても超えなくても，存在しない記録を読もうとすれば単にエラーになります．

大抵のシステムでは端末からファイル終端のシグナルを送る方法が用意されています．ASCII コードを用いるシステムでは「ETX」コード（Control キー+Z を押す）（あるいは「EOT」コード（Control キー+D を押す））がよく用いられます．一度この方法でファイル終端の状態にすると，この状態が維持されて，それ以降はプログラムに端末から入力できなくなるかもしれません．

正式には，Fortran の規格ではファイルの終端に “end-of-file” の記録がある場合のみに，Fortran システムが外部ファイルのファイル終端を検出できるものとしています．END FILE 文は “end-of-file” の記録を出力するために用意されたものです．しかし実際には，ファイル終端を超えて読み込もうとしたときにはほとんどすべての Fortran システムでファイル終端を検出できるため，実質的にこの文を使う必要はなく，使うことをお勧めもしません．

10.6　書式指定

書式付き外部ファイルや内部ファイルに対する READ 文や ERITE 文には書式識別子 (format identifier) を付けなければなりません．次のような形式で行います．

FMT=* 並びによる書式による転送 (list-directed transfer) を表します (逐次アクセスの外部ファイルの場合のみ可)．詳しい規則は 10.10 節で説明します．

FMT=*label* 文番号 (label) は同一プログラム単位の FORMAT 文に付けられたものです．FORMAT 文は書式指定を行うものです．

FMT=*char-exp* 文字式（character expression）*char-exp* の値（内容）は完結した書式指定です．

FMT=*char-array* 文字型配列（character array）*char-array* の要素は書式指定です．書式指定は，必要な分だけ配列の要素を占めることができます．

書式指定子が入出力制御並び (I/O control list) の二番目の項目になっているときには，FMT=を省略することができます．また，装置番号が入出力制御並びの最初の項目になっているならば UNIT=を省略することができます．

書式指定は丸括弧で囲まれた編集記述子 (edit descriptor) の並びで表されます．FORMAT 文で使われる場合を除いて，左の括弧の前の空白文字は無視されます．また，対応する右括弧の後の文字もすべて無視されます．

ほとんどの場合，書式はプログラムの作成時に決定できますし，最も簡単な方法は次のように文字定数で与えるものです．

```
      WRITE(UNIT=LP, FMT='(1X,A,F10.5)') 'Frequency =', HERTZ
```

上の例は FORMAT 文を用いて次のように書くこともできます．

```
      WRITE(UNIT=LP, FMT=915) 'Frequency =', HERTZ
915   FORMAT(1X, A, F10.5)
```

後者の方法を用いると，同じ書式を複数の入出力文で使うことができます．また，書式指定が長い場合や複雑な場合，あるいは，アポストロフィー編集記述子を含む場合にはFORMAT文を使うことで整ったプログラムになります．アポストロフィー編集記述子を含む場合は，囲んでいるアポストロフィーを二重にしなければならないためです[37]．

プログラムの実行時に適当な文字式を使って書式指定子 (format specification) を生成することもできます．この方法によって，例えばデータファイルの書式を最初の記録から読み込み，その書式を用いてデータを読み込むようなことができます．次のプログラム例（部分）は実数値を，値が小さいときには固定小数点形式 (F10.2) で出力し，値が大きいときには指数形式 (E18.6) に切り替えるものです．書式を切り替える値は 100,0000 としました．

```
      CHARACTER F1*(*), F2*12, F3*(*)
*Items F1, F2, F3 hold the three parts of a format specification.
*F1 and F3 are constants, F2 is a variable.
*F1, F2, F3 は書式指定子の三つの部分を記憶します.
*F1 と F3 は定数で，F2 は変数です.
      PARAMETER (F1 = '(1X,''Peak size ='',')
      PARAMETER (F3 = ')')
*... calculation of PEAK assumed to be in here
*... ここで PEAK の値が計算されるとします.
      IF(PEAK .LT. 1.0E6) THEN
          F2 = 'F10.2'
      ELSE
          F2 = 'E18.6'
      END IF
      WRITE(UNIT=*, FMT=F1//F2//F3) PEAK
```

文字定数'Peak size ='は，他の文字定数の中に置かれているためPARAMETER文中でアポストロフィーが二重になっていることに注意してください．次は空白文字も含めた二通りの出力例です．

```
Peak size =  12345.67
Peak size =      0.987654E+08
```

10.6.1　FORMAT 文

FORMAT文 (FORMAT statement) は非実行文に分類され，原則的にはプログラム単位のどこにでも置くことができます．当然，FORMAT文は継続行にすることができるため，最大の長さは 20 行ということになります．同じFORMAT文を複数の入出力文で使うことができます．また，アポストロフィー編集記述子 (character constant descriptor) が含まれていないならば，入力と出力の双方で使うことができます．入出力文の並びと書式指定子中の編集記述子との対応をよく間違えるため，FORMAT文をなるべく，それを用いるREAD文やWRITE文の近くに置く方がよいでしょう．

10.7　編集記述子

編集記述子 (format edit descriptor) にはデータ記述子 (data descriptor) と制御記述子 (control descriptor) の二種類があります．データ記述子はREAD文やWRITE文で転送される項目のそれぞれに１つずつ与えなければなりません．使えるデータ記述子はデータの型に依存します．データ記述子は型を表す

[37] 訳注：5.2.6 節参照.

英字とそれに続く，欄の幅 (field width) を表す符号なし整数値から構成されます．例えば I5 は 5 文字幅の整数値の欄を，F9.2 は 9 文字幅，小数点以下 2 桁の浮動小数値の欄を表します．それぞれのデータ記述子の詳細の説明は次節で行います．

制御記述子は，特定のカラムにタブをしたり，記録を生成したりスキップしたり，続く数値の転送を制御したりと，様々な目的で使われます．詳細は 10.9 節で説明します．

書式指定では直定数 (literal constant) だけが許され，名前付き定数や変数は許されないことに注意してください．

10.8　データ記述子 A, E, F, G, I, L

データ転送並び中のそれぞれの項目に対してデータ記述子を与えなければなりません．実数型，倍精度実数型，複素数型の項目は E，F，G 記述子を使うことができます．一方，それ以外の型ではその型専用の記述子を用いなければなりません．複素数型の数値に対しては二つの浮動小数点数値用の記述子が必要です．

データの型	データ記述子
整数	Iw，Iw.m
実数型，倍精度実数型，複素数型	Ew.d，Ew.dEe，Fw.d，Gw.d，Gw.dEe
論理型	Lw
文字型	A，Aw

データ記述子の w，m，d，e の文字は符号なしの整数値を表し，w，e は 0 より大きくなくてはいけません．

- w　欄全体の幅
- m　出力される最低の桁数
- d　小数点以下の桁数
- e　指数部分の桁数

すべてのデータ記述子には，反復回数を表す数値を前に付けることができます．例えば 3F6.0 は F6.0,F6.0,F6.0 と同じ意味になります．この機能は特に配列を扱うときに便利です．

10.8.1　数値の入出力に関する一般的な規則

数値の出力では常に十進数表記に変換されます．標準の Fortran では 8 進数や 16 進数など 10 以外の基数の表記に変換する機能はありません．このような複雑な表記への変換は内部ファイルを用いて行います．

出力される数値は欄に右詰で出力され，必要に応じて左に空白文字が加えられます．負の値には常にマイナス符号がつきます（この分のスペースを確保しておかなければいけません）．0 には符号がつきません．SP と SS 形記述子を用いて正の数値にプラスを付けるか（SP），付けない（SS）かを指定することができます．数値が大きく，与えられた欄に収まらない場合には欄の幅 w いっぱいにアスタリスクが出力されます．

数値を入力する場合にはそれぞれの欄に右詰で与えた方がよいでしょう．空白文字が途中に含まれていたり，後に加わったりしていない限り，Fortran で許された定数の形式はすべて，対応する型を有する入力欄に問題なく使用することができます．始めの空白文字は常に無視されます．欄全体が空白文字であるときには 0 の値が読み込まれます．数字の間や後ろの空白文字の扱いは BN 形記述子と BZ 形記述子で制御することができます．これらの規則はごく初期の Fortran の名残です．

（BLANK='ZERO' の指定のない）OPEN 文によって接続されたファイルを読む場合には数値の入力欄の数字の間や後ろの空白文字はヌル (null) として扱われ，単に無視されます．それ以外の場合，例えばあらかじめ接続されているファイルや内部ファイルなどからの入力では，数字の間や後ろの空白文字は 0 として扱われます．このディフォルトの設定は BN 形記述子や BZ 形記述子で変更することができます．なお，数字の間や後ろの空白文字を 0 として扱うのが望ましい状況は思い浮かびません．例えば，数値を端末から入力する場合には左詰で入力される場合が多く，この場合，入力値の右に 0 が加えられてしまうことになるため，このディフォルトの設定は間違った選択と言えるでしょう．これが原因のエラーは，入力の書式指定の先頭に BN 記述子を入れることで回避できます．

10.8.2　整数形（Iw，Iw.m）

Iw の記述子で出力された整数値は w 文字幅の欄に右詰で出力されます（始めは空白文字で埋められます）．Iw の記述子も同様ですが，少なくとも m 個の数字が出力されるように必要数の 0 が出力の先頭に付加されます．これは，例えば時刻（時間，分）を出力する場合などに便利です．

```
NHOURS = 8
MINUTE = 6
WRITE(UNIT=*, FMT='(I4.2, I2.2)') NHOURS, MINUTE
```

これによって出力される記録は次のようになります．

```
0806
```

入力では Iw と Iw.m 形で違いはありません．また，入力時に整数型の欄には小数点や指数，カンマなどの句読点を含んではなりません．

10.8.3　浮動小数点型（Ew.d，Ew.dEe，Fw.d，Gw.d，Gw.dEe）

すべての浮動小数点型（実数型，倍精度型，複素数型）のデータは E，F，G 形記述子で転送することができます．複素数型の数値には実部と虚部それぞれに 1 つずつ，二つの記述子を与えなければいけません．必要ならば制御記述子を用いて二つの成分を離して出力することもできます．出力値は指定された桁数に丸められて出力されます．すべての浮動小数点データの転送はスケールファクタの設定に依存します．スケールファクタの初期値は 0 ですが，10.9 節で説明するように P 形記述子を用いて変更することができます．

Fw.d の編集記述子を用いると w 桁の欄に小数点以下 d 桁の固定小数点の値が出力されます．d が 0 の場合にも小数点は必ず出力されます．このため，符号も入れると最大でも小数点の前には w-2 文字の数字しか出力できないことになります．小数点の出力を抑制する必要がある場合には，(F15.0,TL1) のような書式指定を用いるのも一つの方法です．この指定によって次の出力が一つ左から行われるため小数点が上書きされます．もう一つの方法は整数型の変数に値をコピーしてから I 形記述子で出力することです．しかし，多くのシステムで整数の値の範囲が限られることに注意が必要です．F 形の書式は小数点の位置を下の行と揃えることができるため，表形式で出力するときに特に便利です．しかし，極端に大きい値や小さい値に対しては適していません．

Ew.d を用いると，指数形式，あるいは「科学技術」形式で出力されます．(スケールファクターが 0 の場合には）仮数部 (mantissa) は 0.1〜1 の間になります．Ew.dEe の形式は指数部に e 桁の数値を出力することを指定するものです．この形式は指数部が 3 桁以上になる場合に用います（この問題は扱える数値の範囲が小さいコンピュータでは生じません）．E 形の書式は任意の値の数値を扱うのに使うことができます．この形式の欠点は例外的に大きい値や小さい値があっても，出力された数値の中で目立たないことです．

Gw.d は汎用の記述子で，もし値が 0.1 よりも大きく，かつ，欄に納まらないような大きな値でもない場合には全部で d 文字の数字（有効数字）を固定小数点形式で出力し，さらに後ろに 4 つの空白文字を出力します．それ以外の場合には Ew.d と同じ出力になります．Gw.dEe を用いると指数部の桁数を指定できます．もし，固定小数点の形式になる場合には後ろに e+2 個の空白文字が出力されます．

次の例は，３種類の書式による出力の違いを示したものです．

```
X = 123.456789
Y = 0.09876543
WRITE(UNIT=*, FMT='(E12.5, F12.5, G12.5)') X,X,X, Y,Y,Y
```

によって次の二つの記録（行）が出力されます（␣ は空白文字を表します）．

```
␣0.12346E+03␣␣␣123.45679␣␣123.46␣␣␣␣
␣0.98766E-01␣␣␣␣␣0.09877␣0.98766E-01
```

入力では E，F，G 形で違いはありません．入力の場合に欄に小数点があれば，それが優先されます．それ以外の場合には最後の d 桁が小数部として扱われます．もし，指数部分を入力するときには指数を表す文字 E あるいは D の文字の後に入力します（ただし，指数部に符号を付けるときには指数を表す文字は省略できます）．入力欄がコンピュータ内部で扱える桁よりも多くの桁を持つ場合には余分な桁は無視されます．入力では様々な浮動小数点や整数形式の入力にも対応できる Fw.0 を使うのが一番です．

10.8.4　論理型（Lw）

論理値が Lw の書式で出力されると，(w-1) 個の空白文字の後に T あるいは F が出力されます．入力では欄に T あるいは F の文字が含まれていなければなりません．この文字の前には小数点や空白文字が含まれていても構いません．T あるいは F の文字のあとの文字は無視されます．従って，.TRUE. や .FALSE. の表記も可能です．

10.8.5　文字型（A，Aw）

欄の幅を指定せずに A 型の記述子を用いると，転送されるデータの長さが出力欄の幅として使われます．これは妥当な規則と思いますが，文字型の長さが変更されると，同一記録内でその後に出力される項目の位置が変わってしまうことに注意が必要です．

もし，カラムの位置を固定して出力する必要がある場合には，つねに w 文字の幅を用いる Aw の編集記述子を用いた方がよいでしょう．文字型の長さ len が欄の幅 w よりも小さい場合，文字列は欄内に右詰で出力され，その前には (w-len) 個の空白文字が補われます．逆に大きい場合には始めの w 文字が出力され，残りは無視されます．入力の場合には，文字型の長さ len が欄の幅 w よりも小さいときには右から len 文字が入力され，逆に大きい場合には w 文字が読み込まれ，その後に len-w 個の空白文字が補われます．

10.9　書式制御記述子

制御記述子には転送並びに対応する項目を持たず，実行が書式の該当箇所に達したときに制御記述子の指定に従って入出力が行われます．制御記述子のみを持つ書式指定は転送並びを持たない READ 文や WRITE 文に対してのみ可能です．

制御機能	制御記述子 (Control Descriptor)
次の記録（行）へ移動	/
指定したカラム位置へ移動	Tn, TLn, TRn, nX
文字定数の出力	'any char string'
データ並びが空のときに書式の探査を終了する	:
正の値に対して+の出力を制御する	SP, SS, S
空白を null とするか，ゼロとするか	BN, BZ
数値の転送時に使われるスケールファクタを指定する	kP

ここで n，k は整定数です．k は符号を持つ場合がありますが，n は非零の符号なしの整定数です．SP，BN，kP などの制御記述子はそれ以降のすべての数値の転送に影響します．明示的に指定された設定はこれによって影響されません．また，次の READ 文や WRITE 文ではシステムのディフォルトに戻ります．

すべての編集記述子は括弧でくくって，繰り返し回数を表す整数値をその前に付けることができます．例えば，

```
      2(I2.2, '-'),I2.2
```

は

```
      I2.2, '-', I2.2, '-', I2.2
```

と等価です．このような括弧でくくられた並び (sub-list) は必要なだけネストすることができますが，その内部にさらに括弧でくくられた並びがある場合には，後述するように，強制復帰 (forced reversion) が行われるときにその影響が現れます．

スラッシュ編集記述子 (/) やコロン編集記述子 (:) の隣や，D，E，F，G 形編集記述子の直前にあるスケールファクタ（P 形編集記述子）の直後のカンマ (,) は無視されます．

10.9.1　記録の制御（/）

スラッシュ編集記述子（/) は，出力では新たな記録を開始し，入力では，現在の記録に残されているデータを無視して新たな記録に移動します．テキストファイルでは記録は行に相当します．書式付き転送は少なくとも一つの記録を必ず処理（入出力）します．このため，書式に N 個のスラッシュが含まれている場合には，計 N+1 個の記録が処理されます．出力用の書式に連続した N 個のスラッシュがある場合には N-1 個の空行が出力され，入力の場合には N-1 個の行が無視されます．書式付き逐次ファイルがプリンタに送られる場合には，すべての記録の最初の文字は行送り制御文字 (carriage-control) として使われる場合があることに注意してください（10.11 節参照）．シングルラインスペーシング[38]で確実に出力するにはすべての書式指定の先頭とスラッシュ編集記述子の直後に 1X を入れておくとよいでしょう[39]．次の例は，カラムの見出しの次に空行を出力する例です．

```
      WRITE(UNIT=LP, FMT=95) (NYEAR(I), POP(I), I=1,NYEARS)
95    FORMAT(1X,'Year  Population', //, 100(1X, I4, F12.0, /))
```

10.9.2　カラム位置の制御（Tn，TLn，TRn，nX）

これらの編集記述子は次の数値の転送を記録中の現在位置と異なったカラムから始めるためのものです．例えば各コラムを上の見出しに揃えた表を作成する場合などに使えます．これらの編集記述子の n は 1 以上の値です．カラムは左端を 1 として番号づけられています（ただし，プリンタに出力する場合には 1 カラム目は行送り制御文字 (carriage-control) に当てられている場合がありますので注意してください）．

[38]訳注：行間を空けない形式のこと．
[39]訳注：UNIX や DOS などのシステムでは不要です．1X を入れても 1 カラム目に空白文字が出力されるだけです．

Tn	次の出力を n カラム目から開始する.
TRn	n カラム右へ移動する.
TLn	n カラム左へ移動する（ただし，移動できるのは 1 カラム目まで）.
nX	TRn と等価.

入力では，TLn 編集記述子は同一の欄を（多くの場合異なった書式で）再度読み込む場合に使います．出力では，必然的に出力される記録に直接的な影響は与えません．これらの記述子は出力された文字を空白文字で置き換えることはありませんが，記録の出力が完了した時点で記録に含まれるカラムに何も出力されていない場合，そのカラムは空白文字に置き換えられます．例えば，

```
      WRITE(UNIT=LP, FMT=9000)
9000  FORMAT('A', TR1000, TL950, 'Z')
```

によって作成される記録は 52 文字からなり，間の 50 文字は空白文字になります.

10.9.3　文字定数の出力 (' 文字列')

文字定数記述子（アポストロフィー編集記述子・character constant descriptor）は WRITE 文だけに用いることができ，単に文字列が出力記録にコピーされます．文字列中のアポストロフィー「'」は文字列中に続けて二つアポストロフィーを書いて表します．空白文字も無視されず，スペースが出力されます.

10.9.4　符号の制御 (SP，SS，S)

SP 編集記述子が指定されると，以降の正の数値の出力において先頭に正符号が出力されます．SS 編集記述子が指定されると正符号が出力が抑制されます．S 編集記述子が指定されるとコンピュータに依存する初期値に戻されます．これらの編集記述子は入力に対しては何の効果も持ちません．新たな書式付き転送が行われるたびに初期値に戻されます.

10.9.5　空白の制御 (BN，BZ)

BN 編集記述子が指定されると，以降の数値入力欄の数字の間や後の空白文字を無視します．BZ 編集記述子が指定されると，以降の数値入力欄の数字の間や後の空白文字を 0 の文字とみなします．これらの編集記述子は出力では何の効果も持ちません．初期値は OPEN 文の BLANK=の項目に依存し，新たな書式付き転送が行われるたびにこの初期値に戻されます.

10.9.6　スケールファクタの制御 (kP)

スケールファクタ（けた移動数・scale factor）は内部表現と外部表現の変換に E，F，G 形の編集記述子が使われるとき，10 の任意の整数乗を用いてその桁数を制御するものです．もともと浮動小数点値の指数部分がコンピュータの限界と比較して極端に大きな値や小さな値を扱う場合に有用なはずなのですが，別の問題を引き起こすために通常ではあまり実用的な機能ではありません．スケールファクタは特に F 形の編集記述子を用いるときに間違いを犯しやすく，なるべく使用を避けた方がよいでしょう．スケールファクタに関する規則は次のようなものです.

それぞれの書式付き転送のスケールファクタの初期値は 0 です．kP 形編集記述子が指定されると（k は小さな整数値，符号を付けることもできます），スケールファクタは k になります．それ以降に転送される浮動小数点値はすべてこの影響を受けます．入力では，入力される数値に指数部分が明示さ

れている場合には何の影響も与えませんが，指数部分がない場合には

$$\text{内部表現} = \text{外部表現} / 10^k$$

となります.

出力ではその効果は用いられる編集記述子に依存します．E 形編集記述子を用いた場合に対しては，小数点は右に k 桁移動し，指数値が k 減少します．従って実際の値としては変化しません．一方，F 形編集記述子に対しては常に次のようなスケーリングが行われます.

$$\text{外部表現} = \text{内部表現} * 10^k$$

G 形編集記述子の場合には，F 形の形式で出力される数値の範囲では何の効果も及ぼさず，それ以外では E 形編集記述子に対する効果と同じ効果を及ぼします.

10.9.7　入出力制御と強制復帰

新たな入出力が開始されると（括弧の効果を除けば）編集記述子は左から右へ並びの先頭から順にスキャンされていきます．入出力の動作は次に来る編集記述子と入出力並びの次の項目の両方に依存します．次の編集記述子がデータ記述子であるときには，もし存在するならば次のデータ項目が転送されます．次のデータ項目が存在しないときには書式のスキャンは終了になります．次の編集記述子がコロン編集記述子 (colon descriptor) で，かつ，次のデータ項目が存在しない場合には書式のスキャンはそのまま終了されます[40] データ項目が存在する場合にはコロン編集記述子は何の効果も及ぼしません．次の編集記述子が制御記述子である場合には，次のデータ項目の有無に拘わらず，制御記述子の指定が使われます.

編集記述子の並びを使い切ってもデータ項目が続く場合には,「強制復帰」(forced reversion)[41]が適用されます．すなわち，ファイル位置が次の行の先頭に移動し，最後に現れた右括弧に対応する左括弧の位置から，書式並びが再び使われます．もしこの部分に繰り返し回数を表す整数値が付されている場合にはこの数値も再度使われます．もし，右括弧が現れなかった場合には書式全体が再度使われます.「強制復帰」はスケールファクタや符号，空白の制御記述子には何の影響も持ちません.「強制復帰」の機能はあらかじめデータ数を知っておく必要がないため，連続的な記録を保持する配列の入出力に便利です．しかし，印刷の出力を得る場合には「強制復帰」の機能を用いて作成する場合であっても，それぞれの記録に行送り制御文字 (carriage-control) が必要であることに注意してください[42].

10.10　並びによる書式

並びによる出力では，出力する項目に従ってシステムが自動的に選択する書式が使われます．書式の細かな点はシステムに依存するため，一般的な規則について説明します.

10.10.1　並びによる出力

WRITE 文は常に新しい記録に出力を開始し，場合によっては複数の記録に出力することもあります．それぞれの記録は新たな行を始める（行送りの制御・carriage-control）ため，記録の先頭に空白文字 1 文字が出力されます．算術型のデータは内部表現の精度に適した桁数の十進数で出力されます．整数は小数点なし，浮動小数点型は多くの場合その大きさに応じて固定小数点形式か指数形式（科学技術形式）が選択されます．複素数型は実部と虚部をカンマで区切り，全体を丸括弧でくくった形式で出力されます.

論理型は T あるいは F の 1 文字で出力されます．文字型は引用符なしで出力され，記録長に対して

[40]訳注：たとえば `WRITE(*,'(3(I1:'',''))')(I,I=1,2)` によって `1,2` の出力が得られる.
[41]訳注：訳語がわかりません.
[42]訳注：DOS や UNIX ではプリンタも形式的にはファイルとして扱えるようになっているため問題にならないと思います.

文字列が長い場合には次の記録に引き続き出力されます．文字型を除いて，それぞれの項目の出力の後にカンマまたは空白文字（あるいは両方）が，次の出力との区切りとして出力されます．

10.10.2　並びによる入力

並びによる入力の規則によって事実上フリーフォーマットの入力が可能になっています．READ 文は常に新しい記録から入力を開始し，入力項目並び（data-transfer list) の中の項目数に見合う分だけ（必要に応じて複数の）記録から入力を行います．入力される一連の記録には必要なだけの数値と区切り文字が含まれていなければなりません．

数値は，対応する型の定数として Fortran が許す形式であるならばどのような形式でも構いません．ただし，入力値中に空白文字を入れることは文字型として入力する場合以外許されていません．実数型や倍精度実数型の入力に対して整数形式も可能です．論理型の入力に対しては T あるいは F の文字を与えます（それに先行するドット「.」や後に続くいかなる文字[43]も無視されます）．文字型は引用符「'」で囲まなければならず，複素型は実部と虚部をカンマ「,」で区切り，全体を丸括弧で囲っておかなければならないことに注意してください．文字型が長く一つの記録に書けない場合には複数の記録に亘って書くことができます．また，複素型も実部と虚部をそれぞれ二つの記録に分けて書くこともできます．

列挙される数値の区切りは一つ以上の空白文字または一つのカンマ（あるいはその両方）で与えます．新しい記録は空白文字が許されているところならばどこからでも始めることができます．列挙される数値に同じ値が続く場合には，繰り返し回数 (repetition factor) を用いた表現 *n***constant*（*n* は符号なし整数）を使うことができます．アスタリスクの両側には空白文字を入れてはいけません．

カンマを二つ続けて書いた場合にはヌル (null) を表し，READ 文の対応する項目の値は変化しません．*n** の表現も可能で，*n* 個のヌルを表します．

項目の区切り文字の代わりにスラッシュ「/」も可能で，未入力の項目を無視して現在の READ 文を終了し，残された項目には以前の値がそのまま保存されます．一般に，並びによる出力で生成されたファイルは並びによる入力で読むことができますが，文字型の入力で必要なアポストロフィー「'」が並びによる出力では出力されないため，文字型を含む場合にはそのままでは読めません．

10.11　行送りの制御と印刷

書式指定によってテキストの行中でのレイアウトは完全に制御できますが，ページの割付け (pagination) に関する機能は含まれていません．ページの割付けを行うためには初期の Fortran に遡る特別な機能を使う必要があります．ページの割付けが必要のない人でも，行送りの制御 (carriage-control) の規則については生成されるテキストファイルと関連するため，これについて知っておく必要があります．

書式付き出力が「プリンタ」に送られる場合には常に，すべての記録の 1 文字目は印字されず，鉛直方向のスペーシングの制御情報として使われます．この行送り制御文字 (carriage-control character) は次表の 4 つのうちのどれか 1 つでなければなりません．

文字	印字する直前に行われる行送り
空白文字	次の行へ移動
0	2 行進める
1	次のページの先頭へ移動
+	行を進めない．すなわち，同じ行に印字する．

空の記録は一つの空白文字で始まる記録として扱われます．例えば，次の文は新たなページを起こし，先頭にページ番号を，3 行目にタイトルを印字します．

[43] 訳注：カンマ，空白文字などの区切りを表す文字を除く．

```
      WRITE(LP, 55) NUMBER, 'Report and Accounts'
55    FORMAT('1PAGE', I4, /, '0', A)
```

この行送り制御の規則は現実には多くの問題を引き起こす古い Fortran の名残です．第一にシステムが「プリンタ」と呼ぶものに差異があります．ヴィジュアル端末装置やディスクに保存した後に印字されるテキストに対して，この規則が適用されたり，されなかったりします．オペレーティングシステムによっては Fortran の書式付き出力用にプリンタ（や端末）とは異なった扱いがなされる特別のファイル形式を持つものもあります．それ以外のシステムではシステムプリンタに送られるファイルはすべて最初の文字が欠落してしまうため，通常のファイルを印字するには特別なユーティリティが必要になります．

より安全なプログラムを作成するためには，それぞれの書式指定の最初とスラッシュ編集記述子の後に行送り制御文字を必ず加えておくことです．特に「強制復帰」で使われる書式には注意が必要です．通常のシングルスペーシングの行送り制御文字は空白文字ですので，`1X` の編集記述子で簡単に生成することができます．もしこの規則を忘れて，記録の最初に `1` で始まる数値を出力してしまうと，その記録は新しいページを起こしてしまいます．

行送り制御文字の `+` はその結果が予測できるならば便利に使うことができるかもしれません．装置によっては前の記録に重ね書きができるものもあり，文字を組み合わせること（例えば，`=`と`/`の重ね書きで $\neq$ の記号になります）が可能になります．それ以外の装置では単に追加 (append) するだけ，さらにいくつかの装置では（ビジュアル端末もこれに含まれると思いますが）以前の出力が消えてしまいます．明らかにこの機能をあまり当てにすることはできませんが，可搬性を重視したプログラムではこれ以外の方法はありませんので，`+` については忘れてしまうしかないでしょう．

標準の Fortran では上の表の 4 つの行送り制御文字しかありませんが，多くのシステムで鉛直方向のスペーシングやフォントの切り替えなどの特殊な初期設定用に他の記号が使えるようになっています．広く用いられている拡張は通貨記号「`$`」を用いて行末の復帰改行 (carriage-return) を抑制するものです．これによって出力された行に引き続き入力することができるので，プロンプトを生成するのに便利です．残念ながら標準の Fortran にはこの機能はありません．

並びによる出力の規則ではそれぞれの記録の先頭に少なくとも一つの空白文字が付加されるため，必ず生成される行はシングルスペーシングになります．

10.12　入出力文とキーワード

入出力文 (I/O statements) は次の 3 つに分類することができます．

- データ転送文 (data transfer statement)：`READ` 文，`WRITE` 文
- ファイル接続文 (file connection statement)：`OPEN` 文，`CLOSE` 文，`INQUIRE` 文
- ファイル位置付け文 (file positioning statement)：`REWIND` 文，`BACKSPACE` 文

これら文はすべて同じような一般形をとります．唯一の違いは，次のように `READ` 文，`WRITE` 文にはデータ転送並び (data-transfer list) を付けることです．

`READ(` *control-list* `)` *input-list*
`WRITE(` *control-list* `)` *output-list*

それぞれの並び (list) 中の項目はカンマで区切ります．制御並び (control list) の項目は通常キーワードで与えます．キーワードの順序は任意ですが，習慣的に装置番号 (unit identifier) を最初に置いています．Fortran66 との互換性のため，装置番号を並びの最初に置くときにはキーワード `UNIT=` は省略することができます．さらに，`READ` 文や `WRITE` 文でこのキーワードが省略され，書式指定子が二番目に書かれるときにはこのキーワード `FMT=` も省略することができます．従って，次の二つの文は全く同じ意味になります．

```
READ(UNIT=1, FMT=*, ERR=999) AMPS, VOLTS, HERTZ
READ(1, *, ERR=999) AMPS, VOLTS, HERTZ
```

どちらを使うかは好みの問題ですが，より明確にするために本書の他の例ではすべてキーワードを明示しています．

制御並びの多くのキーワードはその引数として文字式をとります．このような場合，文字式の後に続く空白文字は無視されます．これによって，ファイル名やその属性を文字型の変数で与えることが容易になります．

入出力文に使われる式については制限が一つあります．それは，入出力文では入出力文を含む外部手続きの呼び出しを行うことができないということです．これは再帰的に入出力システムを呼び出してしまう危険を避けるためのものです．

10.13 OPEN 文

OPEN 文はファイルと入出力装置 (I/O unit) を接続し，その属性を指定するために使います．OPEN 文で既存のファイルをオープンすることも，新たにファイルを作成 (create) することもできます．装置がすでに別のファイルと接続されているときには，新たに接続が行われるのに先立って既存の接続がクローズ (close) されます．このため，一つの装置を同時に二つのファイルに接続することはできません．また，複数の装置を一つのファイルに接続しようとするとエラーになります．特別な場合として，ある装置とファイルがすでに接続されている場合，OPEN 文を用いて接続の属性を変更することができます．しかし実際には，BLANK=だけがこの方法で変更可能です（RECL=も変更可能なこともあります）．

Fortran 規格では既存の逐次ファイルをオープンしたときのファイル位置の初期値を規定していません．ほとんどのオペレーティングシステムでファイルの先頭に位置づけられますが，可搬性を重視するソフトウェアでは読み出す前に REWIND 文を用いてファイル位置を先頭に戻しておく方がよいでしょう．

一般的な OPEN 文は次のような単純な形をとります．

OPEN(*control-list*)

ここで，制御並び (control list) には次の項目 (item) を順不同で含めることができます．

UNIT=*integer-expression* は入出力装置番号 (I/O unit number) を指定します．整数式 (integer expression) は 0 以上で，上限値はシステムに依存しますが，多くが 99 あるいは 255 になっています．この装置指定子 (unit identifier) は必ず指定しなければならず，ディフォルト値はありません．

STATUS=*character-expression* （STATUS= 指定子）はファイルの属性 (status) を指定します．文字式 (character expression) の値は次のうちのどれかです．

'OLD'	指定されたファイルは既存のものでなければなりません．
'NEW'	指定されたファイルが存在してはいけません．新たに作成されます．
'SCRATCH'	名前なし一時ファイル (unnamed temporary file) が作成されます．このファイルはプログラムの終了時に自動的に削除されます．
'UNKNOWN'	このときの効果はシステムに依存します．しかし，普通は指定されたファイルが存在すればその既存ファイルが用いられ，存在しないときには新たにファイルが作成されます．

ディフォルト値は 'UNKNOWN' ですが，'UNKNOWN' の定義が非常に曖昧なため，STATUS=キーワードを省略することはお勧めできません．

FILE=*character-expression* （ファイル指定子）はファイル名を指定するもので，文字式 *character-expression* 中の後続の空白文字は無視されます．使用可能なファイル名の形式はシステムに依存します．シ

ステムによっては完全なファイルの属性として，装置名 (device) やユーザ名 (user name)，ディレクトリのパス名 (directory path)，ファイル形式 (file type)，バージョン番号 (version number) などの情報が含まれることがあり，これらの区切りるために様々な区切り文字が必要になることもあります．可搬的なプログラムでは，多様なオペレーティングシステムで使えるように短く，かつ，簡単な名前を用いるべきでしょう．ファイル指定子 (FILE= specifier) にはユーザが実行時にファイル名を与えられるように，文字型変数（あるいは配列要素）を使うことができます．ファイル名のディフォルト値はなく，名前なしファイルを用いる STATUS='SCRATCH' の場合を除いて必ず一つ指定しなければなりません．

ACCESS=*character-expression*（ACCESS= 指定子）はアクセスの方法を指定します．値は次のどちらかです．

'SEQUENTIAL'　逐次アクセスを指定，ディフォルト値です．
'DIRECT'　直接アクセスを指定，この場合には RECL= キーワードも必要です．

FORM=*character-expression*（FORM= 指定子）は記録形式 (record format) を指定します．値は次のどちらかです．

'FORMATTED'　書式付き．逐次アクセスの場合のディフォルト値です．
'UNFORMATTED'　書式なし．直接アクセスの場合のディフォルト値です．

RECL=*integer-expression*（RECL= 指定子）は記録長 (record length) を整数式 (integer expression) で指定します．これは直接アクセスの場合には必ず与えなければならず，かつ，それ以外の時には指定してはいけません．記録長の単位は，書式付きファイルの場合には文字数でカウントされますが，書式なしファイルの場合にはシステムに依存します（多くの場合，数値記録単位 (numeric storage unit) が用いられます）．

BLANK=*character-expression*（BLANK= 指定子）は数値の入力欄中の途中や後の部分の空白文字の取り扱いを指定します（BN 編集記述子や BZ 編集記述子で陽に指定される場合はそちらの指定が使われます）．

'NULL'　空白文字をヌル，すなわち無視する．ディフォルト値です．
'ZERO'　空白文字を 0 とする．

ディフォルト値はすべての場合に対して合理的な選択になっていると思います．

IOSTAT=*integer-variable*（IOSTAT= 指定子）は整数式 (integer variable)（または配列要素）に OPEN 文を実行したときの入出力状態コード (I/O status code) を返します．もしエラーが生じなかった場合は 0，それ以外の場合にはシステムに依存する正値を返します．

ERR=*label* (ERR= 指定子）はエラーが生じたときに制御を移す，同一プログラム単位中の実行文の文番号 (label) を指定します．

10.14　CLOSE 文

CLOSE 文 (CLOSE statement) はファイルをクローズし，接続を解除します．接続を解除した後はその装置 (unit) やファイル（存在するならば）は再び自由に接続することができます．もし，CLOSE 文に指定された装置がファイルに接続されていなかった場合には，何の影響も与えません．CLOSE 文の一般的な形式は次のようなものです．

CLOSE(*control-list*)

ここで，制御並び (control list) には次の項目を含めることができます．

UNIT=*integer-expression* （UNIT=指定子）はクローズする装置番号を整数式 (integer expression) で指定します (オープン文の場合と同じです)．

STATUS=*character-expression*（STATUS=指定子）はクローズするときの状態を文字式 (character expression) で指定します．文字式は次の値のどちらかです．'KEEP' はクローズしたファイルをそのまま保存します．'DELETE' はクローズした後ファイルを削除します．ファイルが STATUS='SCRATCH' の指定でオープンされた場合を除いてディフォルト値は STATUS='KEEP' です．ファイルが STATUS='SCRATCH' の指定でオープンされた場合にはファイルのクローズ後に必ず削除され，STATUS='KEEP' は指定できません．

IOSTAT=*integer-variable* （IOSTAT=指定子）と ERR=*label* （IOSTAT=指定子）のどちらも OPEN 文の場合と同様に使うことができます（ただし，CLOSE 文ではあまり問題になるようなことは生じないでしょう）．

10.15 INQUIRE 文

INQUIRE 文は次のように二通りの形式で使うことができます．

INQUIRE(UNIT= *integer-expression, inquire-list*)

INQUIRE(FILE= *character-expression, inquire-list*)

最初の形式は装置番号で調べる形式で，装置 (unit) に関する情報と，ファイルに接続されている場合にはファイルに関する情報を返します．ファイルに接続されていない場合にはほとんどの引数は不定値 (undefined) あるいは'UNKNOWN' の値が返されます．

二番目の形式はファイル名で調べる形式で，名前付きファイル，すなわち，Fortran プログラムがオープンできるファイルが存在するかどうか調べます．文字式 (*character-expression*) 中の後続の空白文字は無視され，可能なファイル名の形式は OPEN 文と同じくシステムに依存します．ファイルが存在し，かつ，装置に接続されている場合には非常に多くの情報が得られます．

inquire-list には以下の項目を指定することができます．これらの項目は ERR=label を除き，指定された変数（あるいは配列要素）に値を代入することで情報を返します．代入には通常の代入文の規則が適用されるため，文字型の項目は必要な長さがあればよく，必要に応じて宣言された長さまで後続の空白文字が補われます．

IOSTAT=*integer-variable* と ERR=*label* は OPEN 文や CLOSE 文の場合と同じように使います．これらの項目は INQUIRE 文を実行したときのエラーを検出するもので，問い合わせの対象となるファイルや装置の状態を示すものではないことに注意してください．

EXIST=*logical-variable* は指定された装置やファイルが存在するときに論理型変数 (*logical-variable*) あるいは配列要素に.TRUE. を，存在しないときには.FALSE. を返します．「装置 (unit) が存在する」ということは指定された装置番号がシステムで許された範囲にあるということです．また，「ファイルが存在する」とは OPEN 文でオープンできるということです．例えばファイルにパスワードが必要な場合や他の人が利用中の場合など，単にオペレーティングシステムがファイルの利用を禁じている場合にもファイルが存在しないとされることがあります．

OPENED=*logical-variable* は指定された装置（あるいはファイル）が現在接続されている場合，論理型変数 *logical-variable* に.TRUE. を返します．

NUMBER=*integer-variable* は接続されているファイルの装置番号を整数型変数 *integer-variable* に返します．接続されていない場合には不定値になります．

NAME=*character-variable* は文字型変数 *character-variable* にファイル名を返します．ファイルに名前がない場合には不定値になります．ファイル名を用いた INQUIRE 文では（装置名 (device name) やディレクトリパスが加えられることがあるため）FILE=で指定した名前と異なる名前が返される場合があります．しかし，返されたファイル名は必ず OPEN 文でオープンできる形式になっています．

NAMED=*logical-variable* は指定されたファイルに名前があるとき，論理型変数 *logical-variable* に .TRUE. を返します．

ACCESS=*character-variable* は記録のアクセス方法を文字型変数 *character-variable* に返します．ファイルが接続されている場合には'SEQUENTIAL' あるいは'DIRECT' のどちらか，接続されていない場合には不定値になります．

SEQUENTIAL=*character-variable* はファイルが逐次アクセスでオープンできる場合'YES' を，逐次アクセスでオープンできない場合'NO' を文字型変数 *character-variable* に返します．それ以外の場合には'UNKNOWN' になります．

DIRECT=*character-variable* はファイルが直接アクセスでオープンできる場合'YES' を，直接アクセスでオープンできない場合'NO' を文字型変数 *character-variable* に返します．それ以外の場合には 'UNKNOWN' になります．

FORM=*character-variable* はファイルが書式付きで接続されている場合に'FORMATTED'，書式なしで接続されている場合に'UNFORMATTED' が文字型変数 *character-variable* に返します．接続されていない場合には不定値になります．

FORMATTED=*character-variable* は書式付き入出力が可能な場合に'YES' を，できない場合に'NO' を文字型変数 *character-variable* に返します．それ以外の場合[44]'UNKNOWN' になります．

UNFORMATTED=character-variable は書式なし入出力が可能な場合に'YES' を，できない場合に'NO' を文字型変数 *character-variable* に返します．それ以外の場合'UNKNOWN' になります．

RECL=*integer-variable* は直接アクセスとしてファイルが接続されている場合に記録長を整数型変数 *integer-variable* に返します．それ以外の場合には不定値になります．記録長の単位は書式付きのファイルに対しては文字数ですが，書式なしのファイルでは単位はシステムに依存します．

NEXTREC=*integer-variable* はファイルが直接アクセスとして接続されている場合，直前に入力あるいは出力した記録番号より 1 大きい番号を整数型変数 *integer-variable* に返します．ファイルが直接アクセスとして接続されているが，全く入出力が行われていない場合 1 が返されます．ファイルが直接アクセスとして接続されていない場合不定値になります．

BLANK=*character-variable* はファイルが書式付きとして接続されている場合に，数値中や後続の空白文字の扱いに応じて'NULL' か'BLANK' が返されます．それ以外の場合には不定値になります．

10.16　READ 文と WRITE 文

READ 文はファイル上の一つあるいは複数の記録から変数や配列要素などの項目並びに情報を読み込みます．WRITE 文は変数や配列，式などから構成される項目並びの情報を出力し，一つあるいは複数の記録をファイルに生成します．一つの READ 文あるいは WRITE 文によって書式なしファイルに対し

[44]訳注：システムが判断できない場合

ては一つの記録しか転送できませんが，内部ファイルも含め書式付きファイルには書式指定によって複数の記録を転送することができます.

この二つの文は同じ形式で書かれます.

READ(*control-list*) *data-list*

WRITE(*control-list*) *data-list*

制御並び (control list) には装置識別子 (unit identifier) を必ず含まなければなりません．そのほかの項目はなくてもかまいません．データ並び (data list) もなくてもかまいません．この場合一つの記録（書式指定によっては複数の記録）が転送されます.

READ 文の場合，*data-list* には変数，配列要素，文字部分列，全ての型の配列が可能です．添字のない配列名は配列全体を意味し，手続きに渡された擬寸法仮配列 (assumed-size dummy array) の場合には添字のない表記は使えません（配列の大きさがわからないため）．並びには DO 並びも使えます.

WRITE 文の場合には READ 文で使用可能な項目に加えて，全ての型の式が使えます．ほかの全ての入出力文と同様に，式中にはほかの入出力文の実行を含むことはできません[45].

10.16.1 装置識別子

書式識別子 (unit identifier) は次の形式のどれかで書かれます.

UNIT= *integer-expression*　整数式 *integer-expression* の値は 0 以上で， 有効な入出力装置を指すものでなければなりません.

UNIT=*　システムであらかじめ接続済みの標準入力，標準出力の装置を指定します.

UNIT=*internal file*　内部ファイル (internal-file) は文字型の変数，配列要素，部分列，配列です．10.3 節を参照してください.

装置識別子が制御並びの先頭に書かれる場合には UNIT=のキーワードは不要です.

10.16.2 書式識別子

書式識別子 (format identifier) は書式付きファイル（あるいは内部ファイル）にアクセスするときには必ず指定しなければならず，それ以外の場合には指定できません．次の形式が可能です.

FMT=*label*　文番号 (label) は同一プログラム単位中の FORMAT 文の文番号です.

FMT=*format*　書式 (*format*) は書式指定（10.6 節参照）を表す文字式あるいは文字型配列です.

FMT=*　並びによる書式 (list-directed format) を指定する場合 (10.10 節参照).

最初に装置識別子がキーワードなしで書かれ，その次に書式識別子が書かれる場合には，キーワード FMT= も省略可能です.

10.16.3 記録番号

記録番号識別子 (record number identifier) は直接アクセスファイルに対しては必ず指定しなければならず，それ以外の場合には指定できません．次の形式で指定します.

REC=*integer-expression*

記録番号は 1 以上で，READ 文のときには記録の存在する範囲でなければなりません.

[45]訳注：式中に外部関数が含まれるとき注意.

誤り識別子，ファイル終了識別子，状態識別子

これらの識別子はどのような組み合わせでも指定することができますが，ファイル終了識別子 (END=*label*) は逐次ファイルあるいは内部ファイルの読み込みの場合のみ意味を持ちます．詳細は 10.5 節を参照してください．

END=*label*
ERR=*label*
IOSTAT=*integer-variable*

10.16.4　DO 型並び

配列のすべての要素を入出力するには，配列名に添字を付けずに転送並びに書くのが最も簡単で効率的な方法です．多次元配列の場合には記憶順 (storage sequence) に要素が転送され，これは最初の添字が最も速く変化する順です．

DO 型並びを用いると要素を選んだり，標準と異なる順序で要素を転送することができます．DO 型並びの規則は通常の DO ループと同様ですが，DO 型並びは転送並びの 1 項目を構成するもので，DO〜CONTINUE ではなく丸括弧で囲んでおきます．例えば，次のように書きます．

```
      READ(UNIT=*, FMT=*) (ARRAY(I), I= IMIN, IMAX)
      WRITE(UNIT=*, FMT=15) (M, X(M), Y(M), M=1,101,5)
15    FORMAT(1X, I6, 2F12.3)
```

DO 型並びを用いると多次元配列を転置 (transpose) した形式で出力することができます．次の例は配列 X(100,5) を横に 5 要素，縦に 100 行で出力するものです．

```
      WRITE(UNIT=*, FMT=5) (I,I=1,5),
     $                     ((L,X(L,I),I=1,5),L=1,100)
5     FORMAT(1X,'LINE', 5I10, 100(/,1X,I4, 5F10.2))
```

この例では，最初の DO 型並びが 5 カラムの見出しを出力し，次の二重の DO 型並びが行番号と 5 つの要素を出力します．丸括弧はそれぞれ対応がとれていなければならず，また，内側の DO 型並びは外側の DO 型並びに含まれる 1 項目であるため，内側の DO 型並びの右括弧の後にカンマが必要なことに注意してください．

DO 型並びの形式は次のようになります．

（ *data-list, loop-variable = start, limit, step* ）

ここで，初期値 *start*，終了値 *limit*，増分 *step* には通常の DO ループと全く同じ規則が適用されます．ループ変数 *loop-variable* （通常は整数型）はデータ並び *data-list* 中で利用可能であり，データ並びにはさらに DO 型並びを含めることができます．

DO 型並びの実行中に誤り状態 (error condition) やファイル終了状態 (end-of-file condition) が生じた場合，入出力終了後のループ変数の値は不定値になります．すなわち，個数の定まっていないデータを入力し，かつ，実際の入力データ数を把握しておく必要がある場合には，通常の DO ループを使う必要があります．

10.17　REWIND 文と BACKSPACE 文

これらのファイル位置づけ文 (file-positioning statement) は外部逐次ファイルのみに使うことができますが，ほとんどのシステムではディスクやテープなどの周辺機器に記録されるファイルに限っているはずです．

REWIND 文はファイルの先頭に現在位置を戻します．従って，REWIND 文の次の READ 文は最初の記録を読み込みます．一方，REWIND 文の後に WRITE 文を用いると，ファイル中の既存の記録はすべて捨てられます．すでに先頭に戻されているファイルに REWIND 文を実行しても何も行われません．接続されているが存在はしていない装置（例えば，あらかじめ接続されている出力ファイル（標準出力ファイル））に REWIND 文を実行するとファイルが生成されます．

BACKSPACE 文はファイルの現在位置を 1 記録戻します．このため，同じ記録を再読込したり上書きしたりすることができます．現在位置が先頭にあるファイルに BACKSPACE 文を実行しても何も起きませんが，存在しないファイルに BACKSPACE 文を実行するとエラーになります．並びによる書式によって書かれた記録に BACKSPACE 文を実行するのも正しくありません（生成される記録の数がシステムに依存するため）．いくつかのオペレーティングシステムでは BACKSPACE 文の実装が難しく，ファイルの巻き戻し (rewind) と適当な記録へ移動する機能しか持たないものもあります．プログラム中にバッファ(buffer) を設けることで BACKSPACE 文の使用を避けることができる場合があります．書式付きファイルの場合には内部ファイルに対する READ 文と WRITE 文を使います．

一般形は次のように互いに似た形式になります．

REWIND(*control-list*)

BACKSPACE(*control-list*)

ここで，制御並び *control-list* には次の項目が指定できます．

UNIT=*integer-expression*

IOSTAT=*integer-variable*

ERR=*label*

装置識別子 UNIT=は必須ですが，その他は省略可能です．装置識別子だけを指定する場合には次の省略形が使用可能です (Fortran66 との互換性のため)．

REWIND *integer-expression*

BACKSPACE *integer-expression*

ここで，*integer-expression* は装置番号です．

11 DATA 文

DATA 文 (DATA statement) は変数や配列要素に初期値 (initial value) を与えるのに使います．DATA 文は非実行文ですが，主プログラム単位中ではプログラムの最初に置かれた代入文と全く同じ働きをします．従って，主プログラム単位内では，次の代入文

```
LINES = 625
FREQ  = 50.0
NAME  = 'PAL'
```

の代わりに，次の DATA 文

```
DATA LINES/625/, FREQ/50.0/, NAME/'PAL'/
```

を使うことができます．

この方法は，特に配列を初期化するときに便利で，プログラムがメモリにロードされたときに一度だけ初期化するので効率的でもあります．

これに対して，手続き (procedure) 中では，項目の値が手続きの実行中に変化する場合に，両者に違いが生じます．DATA 文はプログラムの実行の開始時に一度だけ値をセットするのに対して，代入文では手続きが実行されるたびに値を代入します．

DATA 文と PARAMETER 文の区別も大切です．DATA 文は単に変数（や配列）の初期値を指定するもので，プログラムの実行時に変えることができます．PARAMETER 文は定数の値を指定するもので，コン

パイルし直さない限り値を変えることはできません．ただし，定数の配列は Fortran ではサポートされていないため，通常の配列に DATA 文を使って初期化することになります（実行時に値を変えないように注意しましょう）．

11.1 確定値と不定値

プログラム開始時のそれぞれの変数や配列要素の値は，DATA 文を用いて初期化しない限り不定値 (undefined) になります．不定の変数はその値を確定値 (defined) にするような方法で用いられる実行文のみで使うことができます．具体的には，次の方法で用いることで値が確定されます．

- 代入文の左辺
- DO 文の制御変数（control variable・ループ変数）
- READ 文の入力並び（input list）
- WRITE 文で使われる内部ファイルのファイル識別子
- 入出力文 (I/O statement) 中の状態識別子 (I/O status identifier)
- INQUIRE 文中で，ファイルあるいは装置番号以外で用いられる場合
- 対応する仮引数が手続きの終了までに確定値になる手続きの呼び出し

不定値の変数はこれ以外の使い方をしてはいけません．不注意に不定値の変数を使うとすぐに誤りが生じ，場合によっては原因を見つけ出すのが困難になります．このため，初期化が必要な項目をすべて把握し，DATA 文で適切に初期化しておくことが大切です．

最近のオペレーティングシステムの多くが，前のジョブの残したデータを勝手に読み出すことができないようにするため，プログラムが使うメモリ領域をクリアします．オペレーティングシステムによっては数値としてはあり得ないビットパターンで埋めておいてくれるものもあります．これによると，不定値の変数が式中で使われると必ず実行時エラーになるため，診断機能として非常に有用です．それ以外のシステムでは単に 0 で初期化するだけです．この場合，不定値の変数が使用されているのを見つけ出すのは難しく，プログラムをほかのシステムに移植して明らかになることもあります．初期化されていない変数や配列要素は 0 になるということを当てにすると，オペレーティングシステムを変えたときに正しく動作するかどうかは運次第ということになります．

11.2 変数の初期化

最も簡単な DATA 文の形式は，次のようにスラッシュ「/」で囲った初期値を変数名の後に書いて並べたものです．

```
DOUBLE PRECISION EPOCH
LOGICAL OPENED
CHARACTER INFILE*20
DATA EPOCH/1950.0D0/, OPENED/.TRUE./, INFILE/'B:OBS.DAT'/
```

DATA 文は宣言文の後に書かなければならないことに注意してください．別の書き方は，次のように最初に名前をすべて書いてから初期値を並べて書く方法です．

```
DATA EPOCH, OPENED, INFILE / 1950.0D0, .TRUE., 'B:OBS.DAT'/
```

初期化する項目の数が多いときに，複数の DATA 文を用いるか，一つの DATA 文で継続行を使うかは好みの問題です．もちろん，同じ名前が二度現れるのは間違いです．

文字型変数は,（もしその方が便利ならば）次のように部分列の表記法を用いて分けて初期化することができます．

```
      CHARACTER*52 LETTER
      DATA LETTER(1:26)/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/,
     $     LETTER(27:) /'abcdefghijklmnopqrstuvwxyz'/
```

文字定数の長さと文字型変数の長さが異なる場合には，代入文と同じように打ち切られたり，空白文字が補われたりします．DATA 文中の算術型の項目には代入文の型変換の規則も適用されます．

11.3　配列の初期化

DATA 文を用いて配列を初期化するにはいくつか方法がありますが，どの方法も DO ループで代入するよりも簡単で効率的です．最も必要性の高いのは配列全体を初期化することでしょう．この場合，配列名に添字を付ける必要はありません．複数の要素を同じ値で初期化するときには，次のように定数値の前にその個数 (repeat count) を書くことができます．

```
      REAL FLUX(1000)
      DATA FLUX / 512*0.0, 488*-1.0 /
```

全体の個数は配列の総要素数と一致していなければなりません．定数値は配列要素の記憶される順 (normal storage sequence) に割り当てられ，この順序は最初の添字が最も速く変化する順です．

定数値には名前付き定数 (named constant) は使えますが，整数式 (constant expression) は使えません．個数 (repeat count) は整数の直定数 (literal constant) あるいは名前付き定数です．添字の範囲がパラメータで与えられている多次元配列を初期化するには，次の例のように総要素数を保持する別の整定数が必要です．

```
      PARAMETER (NX = 800, NY = 360, NTOTAL = NX * NY)
      DOUBLE PRECISION SCREEN(NX,NY), ZERO
      PARAMETER (ZERO = 0.0D0)
      DATA SCREEN / NTOTAL * ZERO /
```

数個の要素だけを初期化する場合には，要素を個別に並べて初期化することができます．

```
      REAL SPARSE(50,50)
      DATA SPARSE(1,1), SPARSE(50,50) / 1.0, 99.99999 /
```

三番目の方法は最も複雑な方法で，DO 並び (implied-DO loop) を用いる方法です．この方法は入出力文で使われる DO 並びとよく似ています．

```
      INTEGER ODD(10)
      DATA (ODD(I),I=1,10,2)/ 5 * 43/
      DATA (ODD(I),I=2,10,2)/ 5 * 0 /
```

この例では奇数番号の要素を 43，偶数番号の要素を 0 に初期化します．ループの制御変数（この例では I）のスコープ（有効範囲）は，これが使われている DATA 文の外には及びません．このため，どの整数変数もほかに一切の影響を与えずに DATA 文のループの制御変数として使うことができます．また，これらの文によって変数 I の値を定めることもできません．

多次元配列の一部を初期化する場合，DO 並びをネストすると簡単になることがあります．

```
      DOUBLE PRECISION FIELD(5,5)
      DATA ((FIELD(I,J),I=1,J), J=1,5) / 15 * -1.0D0 /
```

この例では正方行列 FIELD の上三角 (upper triangle) [46]だけを初期化しています.

11.4 手続きの中の DATA 文

手続き (procedure) 中の DATA 文は代入文では代わりにならない機能を担っています．ファイルをオープンしたり，連続的な手続きの呼び出しを通して値をたしあわせていく変数や配列を初期化する際など，手続きの最初の呼び出しのときだけ初期化を実行したい場合がよくあります.

手続きの次の呼び出しまで情報を局所変数や局所配列に保存しておきたい場合には SAVE 文も用いる必要があります（詳しくは 9.11 節を参照）．実際，手続き中の DATA 文で初期化されている変数や配列要素は，その値が不変の場合を除いて SAVE 文で指定しておく必要があります.

次の例では，この手続きの最初の呼び出しでデータファイルをオープンし，論理型変数 OPENED を用いてファイルの状態を保存するものです.

```
      SUBROUTINE LOOKUP(INDEX, RECORD)
      INTEGER INDEX
      REAL  RECORD
      LOGICAL OPENED
      SAVE OPENED
      DATA OPENED / .FALSE. /
*On first call OPENED is false so open the file.
*最初の呼び出しでは OPENED は偽なので，ファイルをオープンする.
      IF(.NOT. OPENED) THEN
           OPEN(UNIT=57, FILE='HIDDEN.DAT', STATUS='OLD',
     $           ACCESS='DIRECT', RECL=100)
           OPENED = .TRUE.
      END IF
      READ(UNIT=57, REC=INDEX) RECORD
      END
```

この例では，簡単のために入出力装置番号は直定数 (literal constant) にしています．手続きの独立性を高めるには，装置番号を手続きの引数とするか，INQUIRE 文を使ったコードを付加して，適当な未使用の装置番号を自動的に取得するようにした方がよいでしょう.

当然のことながら，この手続きの中で，最後の呼び出しであることを知ることはできませんので，ファイルをクローズすることはできませんが，Fortran ではプログラムの終了時にすべてのファイルを自動的にクローズするためファイルのクローズを厳密に実行する必要はありません.

仮引数の変数や配列，関数名と同じ名前を持つ変数を DATA 文で初期化することはできませんので注意してください.

11.5 一般的な規則

DATA 文の一般的な形式は次のように書けます.

DATA *nlist* / *clist* /, *nlist* / *clist* /, ...

ここで，*nlist* は変数名，配列名，部分列名，DO 並びを項目とする並び，*clist* は直定数，名前付き定数，

[46]訳注：I が行，J が列を指すとすると下三角です.

あるいは個数 (repeat count) とアスタリスクを直前に付した直定数や名前付き定数です．個数 (repeat count) もまた符号なし整定数あるいは名前付き定数です.

最初の名前以外の名前の前のカンマは付けなくてもかまいません．DO 並びの形式は次のように書けます.

（ *dlist, intvar = start, limit, step* ）

ここで，*dlist* は DO 並びや配列要素の並び，*intvar* はループ制御変数と呼ばれる整数型変数，*start*，*limit*，*step* はすべてのオペランドが整数型である整数式，あるいはその外側の DO 並びのループ制御変数です.

DATA 文は無名共通ブロック (blank common block) 中の項目を初期化することはできません．名前付き共通ブロック (named common block) に含まれる項目のみ，BLOCK DATA プログラム単位中で初期化することができます（12.4 節を参照）.

DATA 文はそれぞれのプログラム単位中ですべての宣言文の後に書かなければいけませんが，実行文や文関数定義文が DATA 文の間に入っても構いません．ただし，すべての DATA 文を実行文の前に置く慣習に従っておくのが最良の方法でしょう.

12　共通ブロック

共通ブロック (COMMON block) とは，一つ以上のプログラム単位から直接アクセスできる名前付きのメモリ領域に確保された変数や配列の並びのことです．共通ブロックは主に，あるプログラム単位から別のプログラム単位にデータを渡すために使われ，引数並びを用いてデータを渡す方法の代わりとして，あるいは両者を併用することができます.

共通ブロックは複数のプログラム単位でメモリの領域を共有することで大規模なプログラムを小規模なコンピュータで稼働させる目的にも使われることがあります．しかし，このプログラミング手法は勧められるものではなく，仮想メモリ (virtual memory) 機能を持つオペレーティングシステムの増加とともに使われなくなるでしょう.

共通ブロックの名前は外部名 (external name) であり，手続き名など，同一実行プログラム中のほかの大域名 (global name) と異なっていなければなりません．共通ブロックに割り当てられた変数や配列は通常の方法では初期化できず，この目的のために設けられた BLOCK DATA プログラム単位 (BLOCK DATA program unit) を用いる必要があります.

12.1　共通ブロックの使用

普通，あるプログラム単位から別のプログラム単位へデータを渡すには手続き (procedure) の引数並びを介して行うのが最良の方法です．この方法によって手続きの独立性 (modularity and independence) が最大限に保たれます．一方，この方法は多数のデータを互いに共有するパッケージを構成する手続きにはあまり向いていません．この場合，引数並びが長くなり煩わしい上に，非効率的でもあります．複数の手続きで構成されるパッケージが汎用的な利用を目指しているものならば，外部とのインターフェースは可能な限り簡単にしておくべきです.

このためには，パッケージの外部とのデータの受け渡しだけを引数並びで行い，パッケージ内の他の手続きとのデータの受け渡しは共通ブロックを用いて行うとよいでしょう．このようにすることで，ユーザがパッケージの内部に関わらないで済ますことができます.

たとえば，プロッタ (pen-plotter) を扱うパッケージでは，次のような簡単な手続きの呼び出しの方がよいでしょう[47].

[47]訳注：標準の Fortran にはこのような描画機能は一切ありません．calcomp，GKS などのパッケージが有名です．ユーザがパッケージを準備する必要があります.

CALL PLOPEN	plotter 装置の初期化.
CALL SCALE(F)	スケールファクタを F にする.
CALL MOVE(X,Y)	ペンを (X,Y) の位置に移動する.
CALL DRAW(X,Y)	ペンの現在位置から (X,Y) まで線を引く.

明らかにこれらの手続き同士で，ペンの現在位置やスケールファクタなどを互いにやりとりする必要があります．これに必要な共通ブロックの定義はたとえば次のようになるでしょう.

```
COMMON /PLOT/ OPENED, ORIGIN(2), PSCALE, NUMPEN
LOGICAL OPENED
INTEGER NUMPEN
REAL PSCALE, ORIGIN
SAVE /PLOT/
```

これらの宣言文はパッケージ内のすべての手続きで宣言しておく必要があります.

12.1.1 共通ブロック名

それぞれのプログラム単位では COMMON 文 (COMMON statement) に書かれた名前を引用しておくことで，それに含まれる項目にアクセスすることができるようになります．共通ブロック名はスラッシュ「/」で囲み，その名前は COMMON 文と SAVE 文のみに使うことができます．共通ブロック自体には型はなく，その名前はすべてのプログラム単位の名前と区別できるものでなければなりません．さらに，その共通ブロックにアクセスするすべてのプログラム単位中の局所名とも区別できる名前でなければいけません．それぞれのプログラム単位中で使える共通ブロックの数に制限はありません．また，名前なし共通ブロック (blank common block, un-named common block) もあり独自の特長があります．これについては 12.2 節で説明します.

共通ブロックに含まれる変数や配列自体には大局的な属性はありません．これらの変数や配列は共通ブロックに書かれた位置のみによって，ほかのプログラム単位と同じ名前を持つ共通ブロック中の項目と関連づけられています．従って，あるプログラム単位で

```
COMMON /OBTUSE/ X(3)
```

と宣言され，別のプログラム単位で

```
COMMON /OBTUSE/ A, B, C
```

と宣言されている場合，両者で型が同じであると仮定すると，X(1) は A，X(2) は B，X(3) は C に対応付けられます.

COMMON 文によって一つのメモリ領域に異なった名前（別名・aliase）を付けていることになります．全体の長さが同じならば，データの型は一致している必要はありませんが，一般的には，両者で同じ型を持っている場合にある手続きから他の手続きへデータを受け渡すことができます．型が一致しない場合，ある項目が確定値になっても，COMMON 文の同じ位置にあっても異った型を持つ項目はすべて不定値になります．この規則には一つ例外があり，複素数型は実部と虚部の二つの実数値と直接関連づけられているため，一つの複素数型の変数（あるいは配列要素）から二つの実数型へ（あるいはその逆へ）はデータを受け渡すことができます.

以上のように，通常の場合，対応する項目は同じ型を持つようにする必要がありますが，さらに同じ英字名を用いると混乱を最小限に抑えることができます．これには INCLUDE 文を用いると簡単です（お使いのシステムでサポートされているならば）．インクルードファイル (include file) には COMMON 文だけでなく，関連する項目の型宣言と必要に応じて SAVE 文を書いておきます．もちろん，COMMON 文の定義を変更したときには，それを参照するすべてのプログラム単位を再コンパイルする必要があります.

12.1.2 配列の宣言

配列の添字の上下限は COMMON 文の中で宣言することも，型宣言文あるいは DIMENSION 文で宣言することもできますが，どちらか一方で宣言しなければいけません．従って，

```
COMMON /DEMO/ ARRAY(5000)
DOUBLE PRECISION ARRAY
```

は

```
COMMON /DEMO/ ARRAY
DOUBLE PRECISION ARRAY(5000)
```

と，さらに

```
COMMON /DEMO/ ARRAY
DOUBLE PRECISION ARRAY
DIMENSION ARRAY(5000)
```

とも全く同じ意味になります．ただし，最後の書き方は冗長なのでわざわざ使う理由はありません．

12.1.3 型

通常の型に関する規則が共通ブロックに含まれる変数や配列にも適用されます．暗黙の型の規則に従っている場合には型宣言は不要ですが，特に IMPLICIT 文で暗黙の型の規則を変更している場合には型宣言を行っておく方がよいでしょう．型宣言文は COMMON 文の前でも後ろでも構いません．同様に，文字型変数の長さの指定も型宣言文で行う必要があります．この宣言は COMMON 文で行うことはできません．

12.1.4 記憶単位

5.1 節で説明したように，共通ブロックの長さは記憶単位 (storage unit) を用いてカウントされます．すなわち，整数型，実数型，論理型の項目は 1 数値記憶単位 (numeric storage unit)，複素数型と倍精度実数型は 2 数値記憶単位を占めます．可搬性を最大限考慮して，文字記憶単位と数値記憶単位の関係は規定しないものとされています．この理由から，文字型と非文字型の項目を一つの共通ブロックに混在させることはできなくなっています．

このため，ある種のデータ構造を保持するために一つを文字型の項目，もう一つを非文字型の項目として，二つの共通ブロックを用いることがしばしばあります．もし，最初の例で描画パッケージに描画のタイトルが必要ならば，次のような別の共通ブロックが必要になるでしょう．

```
COMMON /PLOTC/ TITLE
CHARACTER TITLE*40
SAVE /PLOTC/
```

互いに関連している共通ブロックの名前には，関連する文字型と非文字型の項目がそれぞれ保持されていることがわかる名前を用いるとよいでしょう．

名前付き共通ブロックの長さは，それを参照するすべてのプログラム単位で同じになっていなければなりません．確実に同じ長さに合わせるには，共通ブロックの内容を全く同じにするのが一番簡単です．ただし，プログラム単位間のデータの受け渡しを行わず，かつ，全体の長さが同じであるならば，型を必ずあわせなければならないというわけではありませんし，同じ順で列挙されていなければ

いけないというわけでもありません．次の二つの共通ブロックの長さはどちらも 2000 数値記憶単位になっています[48]．

```
COMMON /SAME/ G(1000)
DOUBLE PRECISION G

COMMON /SAME/ A, B, C, R(1997)
REAL A, R
LOGICAL B
INTEGER C
```

共通ブロックの項目は連続したメモリ位置に記憶されます．残念ながら，コンピュータによっては倍精度実数型と複素数型の項目を偶数番目のメモリ位置に限って記憶するものがあり，型を混在させた共通ブロックをうまく扱えない場合があります．このような欠点を持つコンピュータでは，倍精度実数型と複素数型のすべての項目を共通ブロックの最初に持ってくると，たいていの場合問題を回避できます．

12.1.5 SAVE 文と共通ブロック

制御が手続きから呼び出し側に返されたときには，局所変数や局所配列と同じように共通ブロックの項目が不定値になることがあります．ただし，共通ブロックが無名共通ブロックの場合や，共通ブロック（種類は問わない）が現在の手続きの呼び出しの階層の中でこの手続きよりも上位に位置するプログラム単位にも COMMON 文の宣言がなされている場合には，このようなことは起こりません．主プログラムは常にこのような呼び出しの階層の最上位にありますから，主プログラムで宣言された共通ブロックには，この理由で不定値になることはありません．そのほかの場合には SAVE 文を使うのが賢明です．

SAVE 文で指定できるのは共通ブロック名だけで，共通ブロックの個別の項目を指定することはできません．従って，例えば次のような指定になります．

```
SAVE /SAME/, /DEMO/
```

共通ブロックがすべてのプログラム単位で SAVE 文の宣言がなされている場合にはデータは常に保持されます．このため，INCLUDE 文を使うならば，COMMON 文や型宣言文といっしょに SAVE 文もインクルードされるようにしておくべきです．この場合，主プログラムでも共通ブロックの宣言をする場合には，SAVE 文が主プログラムにもインクルードされてしまいますが，この SAVE 文には何の働きもなく，害はありません．

12.1.6 制限

手続きの仮引数は共通ブロックに含めることはできず，また，関数では関数名と同じ名前を持つ変数も共通ブロックに含めることはできません．手続きの呼び出しで実引数になる項目は，二重定義の可能性があるため共通ブロックに使うには制限があります．

例えば，手続きが次のように定義されているとします．

```
SUBROUTINE SILLY(ARG)
COMMON /BLOCK/ COM
```

[48]訳注：この二つの共通ブロックはそれぞれ別のプログラム単位にあることを前提にしています．型が違うためデータの受け渡しはできないので，局所変数のメモリの節約が目的とした例です．

そして，この共通ブロックを使っている呼び出し側のプログラム単位では共通ブロックの項目が実引数になっているものとします.

```
      PROGRAM DUMMY
      COMMON /BLOCK/ VALUE
*...
      CALL SILLY(VALUE)
```

この場合，サブルーチン SILLY では ARG と COM は同じ項目 VALUE に関連づけられます．このため，どちらかに値を代入するとエラーになります.

12.2　無名共通ブロック

共通ブロックは，複数のプログラム単位に同じメモリ領域を共有させることでプログラム全体のメモリ使用量を減らす目的にも使われます．この方法は難しく，リスクが伴いますので，他に方法がない場合を除いて，使わない方がよいでしょう.

ほとんどの Fortran システムは静的 (static) なメモリの割り当てを行います．それぞれのプログラム単位は局所変数や配列用に独立したメモリ領域を割り当てられています．複数の手続きがそれぞれ大規模な配列をその中で使う場合には，プログラムが占有するメモリ領域が非常に大きくなってしまいます．ある手続きが別の手続きを呼ぶ階層的な形式ではなく，一群の手続きが順に呼び出されていく場合には，これらの手続きの記憶領域を共有させて，総メモリ使用量を減らすことが可能です．それぞれの手続きの局所配列の領域として同じ共通ブロックを用いるようにします.

名前付き共通ブロックは参照するプログラム単位で同じ長さを持っていなければなりません．このため，名前付き共通ブロックを使う場合にはどの手続きで最も多くのメモリを必要とするかを調べて，他の手続きではその長さまで項目を加えて同じ長さに合わせておかなければなりません．もう一つの方法は特殊な形式である無名共通ブロック (blank common block, un-named common block) を用いる方法で，これにはプログラム単位ごとに異なる長さで宣言することができるという便利な特徴があります.

例えば，あるプログラム単位で次のように指定していても

```
      COMMON // DUMMY(10000)
```

別のプログラム単位では

```
      COMPLEX SERIES(512,512), SLICE(512), EXPECT(1024)
      COMMON // SERIES, SLICE, EXPECT
```

のように指定できます.

無名共通ブロックには，ほかにも次のような二つの特徴があります．一つ目は，DATA 文で初期化できないことで，BLOCK DATA プログラム単位でもできません (これは一時的な記憶領域として使う場合には．あまり大きな問題ではありません)．二番目は，無名共通ブロックに含まれる項目は手続きの終了後に不定値になることがないことです．このため，無名共通ブロックに対して SAVE 文の指定はできません.

12.3　COMMON 文

プログラム単位にはいくつでも COMMON 文を書くことができます．また，一つの COMMON 文には複数の共通ブロックを定義することができます．COMMON 文は宣言文であり，次のような形式で書かれます.

COMMON / *name* / *list-of-items* , / *name* / *list-of-items* ...

name は共通ブロック名 (common block name) として定義され，大域的な通用範囲 (global scope) を持ちます．Fortran 規格では文関数[49]や局所変数，局所配列と同じ名前を使うことが許されていますが，名前付き定数や組込関数と同じ名前を使うことはできません．項目の並び *list-of-items* には変数や配列の名前を列挙します．配列名には寸法の指定をつけることができますが，それぞれの配列の寸法の指定はプログラム単位で一度だけでなければなりません．二番目以降の共通ブロック名の前のカンマ「,」は省略することができます．

普通，無名共通ブロックは一組のスラッシュ「/」で指定します（間の空白文字は無視されます）が，COMMON 文の最初のブロックであるときにはこれを省略することができます．

共通ブロックの内容は同一プログラム単位中にあるそのすべての定義をつなげたものになります．従って，

```
      COMMON /ONE/ A, B, C, /TWO/ ALPHA, BETA, GAMMA
      COMMON /TWO/ DELTA
```

は二つの共通ブロックを定義し，/ONE/ には 3 つの項目が，/TWO/ には 4 つの項目 (ALPHA〜DELTA) が含まれます．

手続き中では，仮引数となっている変数や，ならびに，関数名と同じ名前の変数は COMMON 文に含めることはできません．

12.4 初期値設定副プログラム (BLOCK DATA program unit)

初期値設定副プログラム (BLOCK DATA program unit) は名前付き共通ブロックに含まれる変数や配列に初期値を与える必要がある場合に用いられる特殊なプログラム単位です．初期値設定副プログラムは BLOCK DATA 文で始まり，END 文で終わります．その間に必要な宣言文と DATA 文を書きます．コメント行も書くことができます．初期値設定副プログラムは実行されるものではなく，手続きではありません．

次の例は，12.1 節で例として用いた描画パッケージの共通ブロック中の項目を初期化するもので，例えば，ペンの現在位置の初期値を原点に，スケーリングファクタの初期値を 1 にしています．

```
      BLOCK DATA SETPLT
*SETPLT initialises the values used in the plotting package.
*SETPLT は描画パッケージの変数を初期化します.
      COMMON /PLOT/ OPENED, ORIGIN(2), PSCALE, NUMPEN
      LOGICAL OPENED
      INTEGER NUMPEN
      REAL PSCALE, ORIGIN
      SAVE /PLOT/
      DATA OPENED/.FALSE./, ORIGIN/2*0.0/, PSCALE/1.0/
      DATA NUMPEN/-1/
      END
```

初期値設定副プログラムには，任意個数の名前付き共通ブロックの初期値を与えることができます（無名共通ブロックは初期化できません）．それぞれの共通ブロックに含まれる項目はすべて書かなければなりませんが，すべての初期値をここで与える必要はありません．複数の BLOCK DATA プログラム単位を用いることができますが，共通ブロックは二度以上現れてはなりません．

Fortran66 との互換性のため，プログラムを通して一つだけ無名初期値設定副プログラムを置くことができます．

[49]訳注：原文では intrinsic function になっています．

12.5　初期値設定副プログラムのリンク

プログラムをリンク (link) しているときに，何らかの理由で手続きを含んだオブジェクトモジュールがリンク作業から漏れた場合，ほぼ確実にエラーになります．しかし，リンカ (linker) に何らかの指定を行わない限り，初期値設定副プログラムが欠落した場合にはエラーになりません．生成されたプログラムは初期値設定副プログラムがなくても実行することもできますが，得られる結果は正しくないでしょう．

このリスクは次のようにすると簡単に回避することができます．それは，BLOCK DATA プログラム単位の名前を，それに含まれる共通ブロックを用いるいくつかのプログラム単位の中で EXTERNAL 文を用いて指定しておくことです．すべてのプログラム単位中で行っても害はありません．これによって，これらのプログラム単位が使われたときに，リンク時に BLOCK DATA プログラム単位が参照されるようになります．この方法では，共通ブロックの定義を初期値設定副プログラムを含むそれぞれのプログラム単位に INCLUDE 文を用いて取り込んでいる場合に多少問題になります．それは自己参照になるためで，この問題を避けるため，インクルードファイルにはこの EXTERNAL 文を書くべきではありません．

このように多少ややこしくなりますが，この方法は簡単で有用な予防措置です．また，この方法によって，他の種類のプログラム単位と同じように，初期値設定副プログラムをオブジェクトライブラリに含め，必要なときに自動的に初期値設定副プログラムを抜き出すことができるようになります．

13　使われなくなった機能と使用を避けた方がよい機能

本章で述べる機能は完全に使われなくなっていたり，構造化されたプログラムに使うには適さないような欠点があるため，新しく作成するプログラムには使用すべきではありません．本章の説明は既存のプログラムを読んだり，更新したりする必要のあるプログラマのためのものです．

13.1　非文字型要素への文字型の記憶

文字型が考え出される以前には，算術型の変数や配列に文字を記憶することができましたが，その操作はきわめて限られたものでした．それぞれの項目に記憶できる文字数はシステムによって異なっていました．これに関連した問題として，多くのシステムが未だに算術型の要素に A 形編集記述子を使うことを許していることが挙げられます．これは，データ転送並びと編集記述子の不一致の誤りをそのまま見過ごしてしまう原因になります．

13.2　算術 IF 文

算術 IF 文は次のような形式で書かれる実行文です．

IF(*arithmetic-expression*) *label1* , *label2* , *label3*

これによって 3 箇所への分岐を行うことができます（2 つの文番号を同じにすると 2 箇所への分岐になります）．算術式 *arithmetic-expression* には整数型，実数型，倍精度実数型の式を使うことができます．この式の値が負のときに文番号 *label1*，0 のとき文番号 *label2*，正のときに文番号 *label3* の文に制御が移ります．

13.3　文番号代入文と割り当て形 GO TO 文

この 2 つの実行文は一緒に使われるのが普通です．文番号代入文 (ASSIGN 文) は文番号を整数型の変数へ代入します．一度代入を行うと変数の内容は算術型の値でなくなります．文番号が実行文に付けられた文番号であるとき，その変数は割り当て形 GO TO 文 (assigned GO TO statement) で使うことができます．FORMAT 文の文番号であるときには，READ 文と WRITE 文で使うことができます．これらの文の一般的な形式は次のように書かれます．

ASSIGN *label* TO *integer-variable*

GO TO *integer-variable*, (*label*, *label*, ..., *label*)

割り当て形 GO TO 文のカンマ，ならびに，丸括弧で囲まれた文番号の並び全体は省略可能です．

割り当て形 GO TO 文によって，プログラム単位の内部に置かれた「サブルーチン」のようにプログラム単位のある部分へ，あるいは，一部分からの接続が可能になります[50]．しかし，この目的に対して割り当て形 GO TO 文は便利でもなく，十分なものでもありません．

13.4　PAUSE 文

PAUSE 文は，何らかの方法でユーザ（システムによってはオペレータ）が再開できるような形でプログラムを一時停止させる実行文です．一般的な形式は STOP 文のものと同じで，例えば次のように使います．

```
PAUSE 'NOW MOUNT THE NEXT TAPE'
```

あるいは

```
PAUSE  54321
```

PAUSE 文は WRITE 文一つと READ 文一つで置き換えることができ，この方が柔軟で，かつ，システムへの依存性を低くすることができます．

13.5　選択戻り

選択戻り (alternate RETURN) はサブルーチン内（外部関数では不可）で使われ，CALL 文の終了後に文番号が振られた複数の文に制御を移せるようにするものです．これを使うためには，それぞれ，先頭にアスタリスクを付けた文番号の並びが CALL 文での引数に含まれていなければなりません．与える文番号は，CALL 文が終了した後に呼び出し側のプログラム単位中で実行を開始する箇所に付した文番号です．例えば，

```
CALL BAD(X, Y, Z, *150, *220, *390)
```

に対して，対応する SUBROUTINE 文にはそれぞれの選択戻り指定子 (label specification) に対する仮引数として次のようにアスタリスクがなければいけません．

```
SUBROUTINE BAD(A, B, C, *, *, *)
```

戻る場所は RETURN 文に付けられた整数式の値によって変わります．例えば，

```
RETURN 2
```

[50] 訳注：???

によって，二番目の文番号（この例では 220）が付けられた文から実行が再開されます．もし，RETURN 文に付けられた整数式の値が 1〜n（n は引数中の文番号の個数）の範囲にない場合，あるいは，通常の RETURN 文が実行された場合には，通常のように CALL 文の直後から実行が再開されます．

この方法はエラー処理に使うことができますが，二段以上の呼び出し側のプログラム単位に情報を伝えることができず，柔軟性に欠けます．整数型の引数を用いて，状態値 (status value) を呼び出し側のプログラム単位に戻し，呼び出し側のプログラム単位でこの値に基づいて IF 文 (計算型 GO TO 文も可）を用いて処理するのがよい方法です．

13.6 ENTRY 文

ENTRY 文は外部関数やサブルーチンに別の入口 (entry point) を設けるために使います．ENTRY 文は非実行文で，SUBROUTINE 文と同じ形式で書かれます．ENTRY 文は手続き (procedure) のどこにでも置くことができますが，ENTRY 文の仮引数の宣言文は他の宣言文と同様に宣言文の正しい位置に置かなければなりません．

主要な入口が SUBROUTINE 文であるとき[51]，ENTRY 文で付加された入口 (alternative entry point) はサブルーチンと全く同じ方法で呼ぶことができます．主要な入口が FUNCTION 文であるとき[52]には，ENTRY 文で付加された入口はすべて関数として使うことができます．また，主要な入口が文字型の関数である場合，ENTRY 文で付加された入口はすべて文字型でなければなりません．

ENTRY 文で付加された入口は他と異なった仮引数を持つことができますが，呼び出し側のプログラム単位に仮引数で値を返す場合，プログラマの責任でこのルーチンの終了までに間違いなく値を確定させておかなければなりません．

必然的に ENTRY 文に関する規則は複雑なため頻繁に間違えます．分割した複数の手続きを用い，必要な情報は共通ブロックで共有する方がよいでしょう．少なくともこの方が不満は少ないと思います．

13.7 EQUIVALENCE 文

EQUIVALENCE 文は二つ以上の項目（変数あるいは配列）を互いに関連付ける，すなわち，同じメモリに割り当てるための宣言文です．文字型の項目は文字型の項目とのみ関連付けることができます．その他の型の項目は互いに一致している必要はありません．ただし，共通ブロックと同じように，データの受け渡しは互いに型が一致する場合のみ許されています．なお，二つの実数型と関連付けられた複素数型の場合は例外です．

EQUIVALENCE 文を用いて配列の特別な要素の別名 (alias) として単純な（添字のない）変数名を付けたり，文字型定数を同じ長さの文字型配列に関連付けたりするには何の危険性もありません．例えば，

```
CHARACTER STRING*80, ARRAY(80)*1
EQUIVALENCE (STRING, ARRAY)
```

としておくと，文字列中の 1 文字を STRING(K:K) としてアクセスする代わりに，ARRAY(K) のように多少簡略化された記法でアクセスすることができます．

EQUIVALENCE 文の一般形は次のようになります．

EQUIVALENCE (v, v, ... v), (v, v, ... v), ...

ここで，それぞれの v は変数，配列，配列要素，部分列のどれかです．手続きの仮引数（あるいは，外部関数中のその関数名と同じ名前の変数）はこれに含めることはできません．添字のない配列名は配列の最初の要素を指します．直接的・間接的を問わず，同じ配列の二つ以上の要素を関連付けたり，その他の記憶順の規則に反する関連付けは許されません．共通ブロックに含まれる変数や配列も

[51] 訳注：ENTRY 文がサブルーチンで用いられた場合
[52] 訳注：ENTRY 文が外部関数で用いられた場合

EQUIVALENCE 文で指定することができますが，これによって関連付けられたすべての項目が共通ブロックに含まれることになります．これによって共通ブロックの内容を前向きに拡張することができますが，共通ブロックの規則から後向きに拡張することはできません[53]．

EQUIVALENCE 文には正当な使用法があるのですが，特殊な目的のために Fortran の規則に反した方法で使われているのが常です．このようなことをしたプログラムにはほとんど可搬性がありません．

13.8　組込関数の個別名

組込関数名を手続きの呼び出しの実引数に用いる場合のみ，総称名 (generic name) に変えて個別名 (specific name) を用いるべきです．またこのとき，その個別名は INTRINSIC 文で宣言されていなければなりません．次の組込関数はこの方法で使うことができず，これらの関数の個別名は完全に使われなくなったものです．

使われなくなった個別名	望ましい総称名
IFIX, IDINT	INT
FLOAT, SNGL	REAL
MAX0, AMAX1, DMAX1	MAX
AMAX0, MAX1	MAX *
MIN0, AMIN1, DMIN1	MIN
AMIN0, MIN1	MIN *

*印を付けた AMAX0，MAX1，AMIN0，MIN1 関数は引数と異なった戻り値を持つため，MAX，MIN 関数で置き換えるためには適当な型変換関数を用いる必要があります．

13.9　PRINT 文と簡略形の READ 文

PRINT 文はシステムであらかじめ接続されている標準出力 (standard pre-connected output file) に書式付き，あるいは，並びによる書式で出力するものです．このため，次の 2 つは全く同じ意味を持ちます．

```
READ fmt, data-list
READ(*, fmt) data-list
```

PRINT 文はその???が???のため，機能が限られており，間違えやすいものです．

同様に簡略化された READ 文 (simplified form of READ statement) もあり，次の 2 つは同じ意味になります．

```
READ fmt, data-list
READ(*, fmt) data-list
```

13.10　END FILE 文

END FILE 文は次のように REWIND 文や BACKSPACE 文と同じ形式を持ちます．

```
END FILE(UNIT=unit, ERR=label, IOSTAT=int-var)
END FILE unit
```

[53]訳注：詳しい説明は浦昭二編，FORTRAN77 入門 (改訂版)・培風館，p.305-306 を参照．

この文は連続的に入力する際にファイルの終端を検出する機能がある逐次アクセスファイルに対して，特殊なファイル終端 (“end-of-file”) の記録を追加するためのものです．このファイル終端の記録以降には何も出力できませんので，次の操作は CLOSE 文，REWIND 文，BACKSPACE 文のどれかになります．

現在のほとんどすべてのシステムでこの機能を使わなくても入力ファイルの終端を検出できるため，この文は不要になっています．Fortran の規格ではファイルの終端記録は実際の記録なので，ファイルの終端を検出した後に新たな記録を追加するのに先だって，明示的に BACKSPACE 文を実行する必要があります．この考えは少し不自然で，すべてのシステムで正しく実装されているわけではありません．これは Fortran の規格から意図的に離れた方が可搬性を向上することができるまれな例です．

13.11　使われなくなった書式記述子

データ記述子 Dw.d は入力では Ew.d とまったく同じ，出力では指数部分の E が D になることを除いて同じです．実数型と倍精度実数型はどちらも D，E，F，G で同じように読み込むことができます．

書式記述子 nHstring は ’string’ とまったく同じです（ここで，n は文字列の長さを与える符号なし整定数です）．書式付き WRITE 文に用い，出力記録に指定した文字列がそのままコピーされます．nH の形式はアポストロフィー「’」を出力する際に二つ並べて書く必要がありませんが，正確に文字数を数えておく必要があります．

14　Fortran 規格に対する共通の拡張機能

Fortran77 の公式規格 (ANSI X3.9-1978) が制定される前から，さまざまなソフトウェア開発会社が独自の拡張機能を加え始めていました．米国国防省は 1988 年に国防省との契約に必要となる Fortran の拡張機能を提示した MIL-STD-1753 と呼ばれる文書を発表しました．後にこの条件は連邦政府の他の分野の調達にも用いられるようになったため，現在この拡張機能はほぼすべてのシステムで提供されており，可搬性の低下を心配することなく使うことができます．

14.1　MIL-STD-1753 拡張

14.1.1　IMPLICIT NONE 文

この文は暗黙の型がないことを宣言するもので，すべての名前が付けられた項目（変数，配列，定数，外部関数）はその型を型宣言文で陽に宣言しなければなりません．この文はすべての宣言文の前で宣言しなければならず，同一プログラム単位では，他の IMPLICIT 文を使うことはできません．多くのプログラミングの初心者はそれぞれの名前を使う前に型宣言を行うことを無駄な作業と考えます．しかし，変数名のスペルミスが非常に簡単に発見できるメリットはとても大きく，ほとんどの熟練プログラマは型宣言の価値を認識しています．

14.1.2　INCLUDE 文

INCLUDE 文はソースコードの一部を書いた別のファイルを指定するための文です．このファイルは，例えば，共通ブロックやそれに関連する宣言文や共通に使われる定数（例えば π）など，複数のプログラム単位に共通する宣言文を書くのに非常によく使われます．当然のことながらファイル名の形式はシステムに依存します．可搬的なプログラムを書く場合にはほとんどのオペレーティングシステムで使えるように，単純な名前を用いるのが賢明です．例えば，単に

```
INCLUDE 'trig.inc'
```

としておき，ファイル trig.inc(TRIG.INC かもしれません) に

```
REAL PI, TWOPI, RTOD
PARAMETER (PI = 3.14159265, TWOPI=2.0*PI, RTOD=PI/180.0)
```

のように書いておきます．もしこのような定数が一カ所だけで定義されていると，内容をチェックするのが非常に簡単になります．同様に，共通ブロックが一カ所で定義されていれば，プログラムを通して一貫性を保証することができます．

14.1.3 END DO を用いた DO ループ

Fortran77 の規格で IF 文が END IF 文で組になっているのに対し，DO 文が END DO 文で対になっていないのは欠点と思います．この拡張は非常に多くのシステムで備えられており，それぞれのループで異なる行番号を決めていく必要がなくなるため便利です．例えば，次のように使います．

```
DO J = 1,NPTS
   SUM   = SUM + X(I)
   SUMSQ = SUMSQ + X(I)**2
END DO
```

繰り返し部分を明示するため，DO 文と END DO 文の間の行を字下げしておくのがよいでしょう．DO〜END DO を用いたプログラムでは，ほとんどの行番号が GO TO 文の飛び先に付けられることになるため，通常の実行の流れが変更されることをプログラマに示唆することができます．文番号を使った DO ループを用いたプログラムではこのような例外的な処理が目に付くことはほとんどないでしょう．

A 組込関数一覧

この表はそれぞれの関数の引数の個数と使用できる型を示したものです[54]．型の記号は I=整数型 (Integer)，R=実数型 (Real)，D=倍精度実数型 (Double precision)，X=複素数型 (Complex)，C=文字型 (Character)，L=論理型 (Logical) を表し，*は引数の型と同じ型の結果を返すことを表します．複数の引数が必要な場合には，すべて同じ型でなければならないことに注意してください．

R = ABS(X)	複素数の絶対値（すなわち，二つの成分の二乗和の平方根）を求めます.
* = ACOS(RD)	逆余弦関数（アークコサイン・arc-cosine），結果は 0〜$+\pi$ の範囲です.
R = AIMAG(X)	複素数の虚部を取り出します．実部を取り出すには REAL を用います.
* = AINT(RD)	切り捨て †
* = ANINT(RD)	最も近い整数に丸めます.
* = ASIN(RD)	逆正弦関数（アークサイン・arc-sine），結果は 0〜$+\pi$ の範囲です †.
* = ATAN2(RD,RD)	arg_1/arg_2 の逆正接関数（アークタンジェント・arc-tangent）を求めます．$-\pi$〜$+\pi$ の範囲の正しい象限で解が求められます．二つの引数がともに 0 のときにエラーになります.
C = CHAR(I)	システムの文字コード表の N 番目の文字を返します.
X = CMPLX(IRDX,IRD)	複素数型に変換します．二番目の引数は省略可能です.
X = CONJG(X)	複素数の複素共役 (complex conjugate) を求めます.
* = COS(RDX)	余弦関数 (cosine)，角度はラジアン単位.
* = COSH(RD)	双曲線余弦関数 †.
D = DBLE(IRDX)	倍精度実数型へ変換します.
* = DIM(IRD,IRD)	arg_1 と arg_2 の正の差，すなわち，$arg_1 > arg_2$ のとき (arg_1 - arg_2)，それ以外のとき 0 を返します（arg_1，arg_1 はそれぞれ 1 番目，2 番目の引数）.
D = DPROD(R,R)	二つの実数型の積を倍精度実数型で求めます.
* = EXP(RDX)	指数値 (exponential) を求めます．自然対数 (natural logarithm) の逆関数です.
I = ICHAR(C)	文字列の最初の文字のシステムの文字コード表上の位置を返します.
I = INDEX(C,C)	最初の引数の文字列から二番目の引数の文字列を探し，その位置を返します．見つからなかった場合には 0 を返します.
I = INT(IRDX)	切り捨て (truncation) により整数型に変換します.
I = LEN(C)	文字型引数の長さを返します.
L = LGE(C,C)	ASCII 文字コードを用いた文字比較を行います．$arg_1 >= arg_2$ なら真を返します.
L = LGT(C,C)	ASCII 文字コードを用いた文字比較を行います．$arg_1 > arg_2$ なら真を返します.
L = LLE(C,C)	ASCII 文字コードを用いた文字比較を行います．$arg_1 <= arg_2$ なら真を返します.
L = LLT(C,C)	ASCII 文字コードを用いた文字比較を行います．$arg_1 < arg_2$ なら真を返します.
* = LOG(RDX)	底が e の対数です (e=2.718...).
* = LOG10(RD)	底が 10 の対数です.

[54] 訳注:† 印の関数は原著にないものです.

* = MAX(IRD,IRD,...)	最大の引数の値を返します.
* = MIN(IRD,IRD,...)	最小の引数の値を返します.
* = MOD(IRD,IRD)	arg_1 modulo arg_2，すなわち，arg_1 を arg_2 で割った商を返します.
I = NINT(RD)	四捨五入整数化 †
R = REAL(IRDX)	実数型へ変換します.
* = SIGN(IRD,IRD)	符号の変換を行います．arg_2 が負のとき結果は $-arg_1$，arg_2 が 0，あるいは正のとき結果は arg_1 になります.
* = SIN(RDX)	正弦関数 (sine)，角度はラジアン単位.
* = SINH(RD)	双曲線正弦関数 †.
* = SQRT(RDX)	平方根.
* = TAN(RD)	正接関数 (tangent)，角度はラジアン単位.
* = TANH(RD)	双曲線正接関数 †.

B　組込関数の個別名

個別名 (specific name) は他の手続きの実引数として使う場合には依然として必要です．この場合，個別名を INTRINSIC 文で宣言しておく必要もあります．次の表は Fortran77 で依然として有用な個別名を示したものです．ここにない組込関数は総称名を持たないか，実引数として渡すことができないかのどちらかです.

総称名	個別名			
	整数型	実数型	倍精度実数型	複素数型
ABS	IABS	ABS	DABS	CABS
ACOS		ACOS	DACOS	
AINT		AINT	DINT	
ANINT		ANINT	DNINT	
ASIN		ASIN	DASIN	
ATAN		ATAN	DATAN	
ATAN2		ATAN2	DATAN2	
COS		COS	DCOS	CCOS
COSH		COSH	DCOSH	
DIM	IDIM	DIM	DDIM	
EXP		EXP	DEXP	CEXP
LOG		ALOG	DLOG	CLOG
LOG10		ALOG10	DLOG10	
MOD	MOD	AMOD	DMOD	
NINT		NINT	IDNINT	
SIGN	ISIGN	SIGN	DSIGN	
SIN		SIN	DSIN	CSIN
SINH		SINH	DSINH	
SQRT		SQRT	DSQRT	CSQRT
TAN		TAN	DTAN	
TANH		TANH	DTANH	

C GNU Free Documentation Licence

C.0 PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

C.1 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an

otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

C.2 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

C.3 COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

C.4 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

C.5 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

C.6 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

C.7 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

C.8 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

C.9 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

C.10 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

D Acknowkedgements

I am grateful to many people who have let me know of mistakes in my text. These include: Paul Youngman, Mikhail Titov, Myron Calhoun, David Simpson, Rory Yorke, Alexandr Lomovtsev, John Girash, Rob Scott, and Jan Wennstrom. Apologies to those whose contributions I have forgotten to record.

Index

本書は WINDOWS 用の g77(GNU Fortran77) と一緒に配布されていた prof77.ps を訳したもので，原著は講義用に配布した CDROM にも収録しています．この訳はまだ不完全なものですので原著と併用で講義の予習復習に活用して頂けると幸いです．

最近では Fortran90, Fortran95 などより新しい規格が出ていますが，Fortran77 には通常の数値計算には十分な機能が備わっています．複素数型がサポートされているなど他の言語にはあまりないような特長もあり，Fortran77 は依然として数値計算によく使われています．また，新しい規格の Fortran でも Fortran77 で書かれたプログラムがそのまま動くこと，フリーコンパイラ g77 が使えること，Fortran77 では文の種類が少なく覚えやすいことなど，依然として練習用としても Fortran77 には利点があるように思います．

訳に際しては

浦昭二編，FORTRAN77 入門 (改訂版)・培風館

`http://www.komazawa-u.ac.jp/~kazov/Nis/fortran/words.txt`

を参照しました．前者は Fortran77 に関して広くカバーしていて，辞書的に使えるのでとても便利な教科書です．

和訳に関する誤りやその他のご指摘は， `prof77_ja@kiban.civil.saitama-u.ac.jp` まで頂けると幸いです．その際， `mail_on_prof77_ja` の文字列を件名に含めるようにしてください（迷惑メールのフィルタに使います）．

埼玉大学理工学研究科・講師・茂木秀則