The University of Osaka
Institutional Knowledge Archive

Title	フォートラン・プログラミングにおけるバッグ（誤り）とデバッグ（修正）
Author(s)	磯本, 征雄
Citation	大阪大学大型計算機センターニュース. 1974, 13, p. 1-15
Version Type	VoR
URL	https://hdl.handle.net/11094/65236
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

大阪大学大型計算機センター・ニュース（No. 13）1974－5

プログラム・ノート

フォートラン・プログラミングにおける バッグ(誤り)とデバッグ(修正)

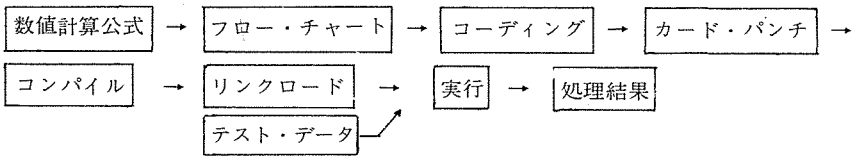
研究開発部　磯　本　征　雄

1．はじめに

フォートランは，科学技術計算のための標準プログラミング言語である。そして科学技術計算公式を容易にプログラム化できるため広範な研究分野で利用されている。ところが“プログラム化しやすい事”が即座に“プログラムを短期間に作れる事”にはならない。実際にプログラムを作成してみると，実に様々の誤りを犯すものである。しかもその原因すら発見できずに困惑してしまう事もよくある。これらはプログラマー個人の経験で培われた勘によって解決されているのが現状である。

一方，これらバッグ（誤り）及びデバッグ（誤りの修正）についての知識及び手法が，プログラミングの知識及び手法の一部分として文献[1), 2), 3)]でも取扱われ，またすでに研究グループ（プログラム相談の機械化）さえ生れている。この反面，現在の状況においても，バッグ及びデバッグについてはっきりとした意識をもってプログラムを作っている利用者は少いように思われる。このような理由により，プログラミングの際の参考になるように，バッグ及びデバッグについて解説を試みた。

プログラム作成は多くの場合，



の順に進められる。バッグがなければ，これでプログラムは完成である。ところがある段階でしばしば誤りを犯し，その時点ではそれに気づく事もなく（例え気づいても偶然の場合が多い）次の段階へ進む。そして誤りに気づくのは後の段階に至ってである。それは，計算機より出力されたメッセージか又はテスト・データによる出力結果が予測された結果と違っている事により知らされる。このようにバッグの存在が発見されるプログラム上の位置やプログラム作成段階について確認される事は偶然の場合（又は利用者）が多い。

さらに悪い事には，バッグの存在を知っても，それが直ちにバッグの場所に結びつかず修正に手間取る事がある。この解説はこれらバッグの発見とデバッグについての指針の一助となる事を願って書かれたものである。なお参考までに記しておくが，統計[1)]によれば「バッグの原

＊ 本稿で使われる用語については日本工業標準JIS FORTRAN C6201－1972，JIS 情報処理用語 C6230－1970，日本規格協会を参照されたい。

因」は表１に示すように，圧倒的に利用者プログラム自体の中にあったという事である。したがって不信な事が発生した場合，まずもって利用者プログラムの中味を充分に調べなければならないだろう。

表１　Percentage of system failures due to

Hardware	1%
Soft ware	2%
Operating	5%
Programing	90%
Systems	2%

文献(3)

２．フォートラン文とバッグ

フォートラン・プログラムは主として表２第１列に示した文からなっている。これらの文は相互に関係し合っている。例えば，文番号付きの文と GO TO文のように両者１対になってその機能を果たす強い関係にある文もあれば，副プログラム文と STOP文のように相互にほとんど影響し合う事のない弱い関係にしかない文もある。まず強い関係にある文に注目する。表２の第２列目にそれらの間に実線を結んで示した。これらの文は単独でプログラム中に現われる事はなく，常に対になっている。(但し副プログラム文と EXTERNAL 文のように主従の関係にあるものもあるがこの点は無視し，EXTERNAL 文があれば必ず副プログラム文があり，その逆が必ずしも真でない場合も含まれている。) フォートランは四則演算が中心であるため常に配列や変数の値が重要であるが，これについてはその影響の範囲が余りに広いので表からはずした。

強い関係にある文は，その一方にバッグがあった場合に必ずその影響が他方の文におよぶ。そしてバッグの悪い影響は，特に強い関係にある文を通して次々に伝播され，拡散されてゆく。その影響が計算機・機能に対して破局的状態になった時点（場所）でバッグがある事が発見される。(バッグをもつ文自体で，バッグの存在が発見されるのはむしろ幸運である。) 相互に必ずしも強い関係にある文とは言えないが，時に重大なバッグとなり得るものでしかも見落されがちな関係にある文について点線で結んだ。例えば副プログラム相互間でデータを転送するために用いた READ 文と WRITE 文においては，一方に誤りがあれば，当然，他方の結果は信用できないものとなる。このようにバッグのある文と，バッグのために悪い影響が表面にあらわれる場所の間に距離がある事を知っておかなければならない。

文はそれぞれに固有の機能と効果をもっている。したがってバッグをもつ文はその文の機能と効果に符合した影響を次の文へ伝播する。例えば表２第３例に示した副プログラムの結合に関する文（表中○印）にバッグがあれば，この時の誤りは結合処理時に発見される事が多い。以下同様に「変数・配列の（数）値を変える」，「実行順序の流れを変える」，「変数・配例の（数）値の転送」及び「変数・配列の（数）値の影響を直接受ける」等の項目について表２の対応する欄の各々の文に○印で示した。各文中にバッグがあった場合，上記の各項目に符合する内容に見合った性質の影響が生ずる。

表２　文の相互関係と特徴

文			強い関係をもつ文	(1)	(2)	(3)	(4)	(5)
実行文	代入文	算術代入文			○			○
		論理代入文			○			○
		ASSIGN文			○			○
	制御文	単純GO TO文				○		
		割当GO TO文				○		○
		計算型GO TO文				○		○
		算術IF文				○		○
		論理IF文						○
		CALL文		○	○		○	
		RETURN文		○		○		
		CONTINUE文（又は文番号付の文）						
		STOP文とPAUSE文						
		DO文			○	○		○
	入出力文	READ文			○		○	○
		WRITE文					○	
		補助入出力文						
非実行文	宣言文	DIMENSION文					○	○
		COMMON文		○			○	○
		EQUIVALENCE文					○	
		EXTERNAL文		○				
		型宣言文						
		DATA文			○		○	○
		FORMAT文						
		文関数定義文						
		副プログラム文		○			○	○

(1)　副プログラムの結合に関するもの
(2)　変数・配列の(数)値を変える。
(3)　実行順序の流れを変える。
(4)　変数・配列の(数)値の記憶と転送
(5)　変数・配列の影響を受ける

理解を容易にするために一例をあげる。今CALL文でバッグが発見されたと仮定する。CALL文は副プログラム文と強い関係をもつ。したがってこのCALL文に対応する副プログラム文についても誤りの有無を同時にしらべてみなければならない。またCALL文は，副プログラムの結合に関するものであると同時に，実行の前後で配列・変数の(数)値を変える事，実行の流れを変える事（＆1000がある場合）及び変数・配例の(数)値の転送をおこなう等の点で次の文へ，あるいは引用されたサブプログラムへ影響する。したがって(数)値がおかしい場合，あるいは処理時間が意外に短かいので流れがおかしいのではないかと思われる場合のいずれにお

いても，CALL 文は一度しらべてみる必要のある文である。このようにバッグとその症状の 間には，文の性質に依存した関連性もある。

3．バッグの特徴と発見時点

フォートラン・プログラムは，コンパイレーション→リンクロード→実行→結果出力の順に処理が進む。そしてバッグは各段階ごとに異なる方法で診断される。したがって各段階ごとに，各々の処理内容に対応したエラー・メッセージが出力される。ところが，計算機による診断では発見し得ないバッグもある。これらは，適切な例題を用いて実際の実行結果からその所在をしらべるしかない。ここではジョブ処理のバッグ発見段階とバッグの特徴についてのべる。

コンパイル時

この時，フォートランに対する文法診断がおこなわれる。診断は主プログラムと副プログラム単位に独立におこなう。診断の結果，出力されるエラー・メッセージは多岐多様にわたり，また計算機システムにより出力書式は異る。紙面の制約のためここでは省略するが文献 4）に具体的に説明されているので参照するとよい。この段階でのエラーは主として文単位で出力されるので，バッグそのものが示されている事がほとんどである。したがってエラー・メッセージの意味を正しく理解するならば修正は比較的容易である。しかしフォートラン文法の特徴から次のような困難な例があり得るので示しておく。

a）1つのバッグが他の正しい文までも誤りであるとしてしまう。これは図1の例の状態で起る。GO TO 20は正しい文である。20 X=X=1.0は，20 X=1.0の誤りである。ところが，この2つの文は密接に結びついているために(表2参照)，後者が誤りであればGO TO 20文の行先き20がなくなり，これもエラーとして出力される。但し，修正されるべきは 20 X=X=1.0 のみである。

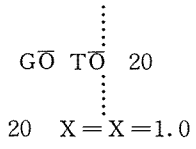


図1

このように密接に関連した強い関係をもつ文の間では一方の文の誤りが見かけ上さらに別の文をも誤りであるが如くに多くのエラー・メッセージを出力するのでバッグの位置をあいまいにしてしまう事がある。

b）1つのバッグが他のもう一つのバッグをおおいかくす事がある。図2において DIMENSION 文の中の A(2,4(は誤りであり A(2,4) と修正されなければならない。コンパイル時にはこの文は無視される。この結果 X=A(I, J, K) の中の配列Aは外部関数として処理される。ところがAは実際には2次元配列である。したがって DIMENSION 文が誤りである場合に

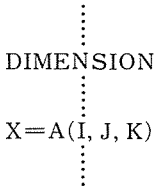


図2

はエラー・メッセージは出力されないが，それが修正された段階で改めてエラー・メッセージが出力される。（これはDIMENSION文を忘れた場合も同じである。）このようにバッグが表面にあらわれず，他のバッグのためにかくされてしまうことがあるので注意が必要である。

c）2つのバッグが相互に打消し合って表面にあらわれない事がある。図3において第2行は最後の）が落ちている。第3行はカラムのまちがいである。ところが，これら2つの文は個々にはバッグであるが，偶然に第3行目7カラム目以後が第2行目の欠落部分としてつながり第3行目は第2行目の継続行として処理されてしまう。したがってエラーは全く出力されない。このようにコンパイル時においてさえ文法違反が検出されないこともある。

```
Col 5 6 7
      DIMENSION A(10,10), B(10), C(10)
                  ⋮
      X=A(I,J)*B(I
Y=B(I)+C(I)
                  ⋮
```

図3

以上はコンパイル時に発見されるべきエラー及び当然発見されるべきはずのエラーが，計算機の性質上見落されるものについて示した。しかし，これらのバッグはまだ比較的発見されやすくまた容易に修正され得るものである。

リンク・ロード時

この段階では副プログラム間の結合処理がなされる。したがってエラーは表2第3列目の○印の文又はその周辺の密接な関係をもつ文において検出される。

例えばCALL文中の仮引数の個数と，これに対応する副プログラム中の実引数の個数の不一致等に対して出力されるエラー・メッセージはこの典型である。その他，このリンク・ロード時に検出されるバッグ及びエラー・メッセージの種類については文献5）を参照するとよい。いずれの場合においても，出力されたエラー・メッセージとバッグとの間の関係が直接に示されているとは限らない。以下にエラー・メッセージの文面だけからはバッグの所在を推測する事の困難な例を示す。図4及び図5の例ではいずれも

$$\text{LLK17I} * * * \text{ UNRESOLVED ENTPYES } \left\{ \begin{matrix} \text{A} \\ \text{SUB1} \end{matrix} \right\}$$

がリンクロード時に出力される。図4に対してはサブプログラムAがない事を示す。図5に対してはサブプログラムSUB1がない事を示す。ところが図4においては配列Aを定義する事を忘れたため，配列Aは外部関数とみなされたものである。したがってプログラマーの意図と計算機の出力したメッセージの間に不一致がみられる。図5の例ではサブルーチンSUB1の 副

```
C  MAIN PROGRAM
   (宣言文なし)
      ⋮
   Y=A(I)
      ⋮
   STOP
   END
```

図4　主プログラムのみから成るプログラムの例

```
      ⋮                    SUBROUTINE SVB1
CALL  SUB1                       ⋮
      ⋮
RETURN                       RETURN
END                          END
```

図5　CALL文又は副プログラム文における誤りの例

プログラム文においてSUB1を誤ってSVB1とミスパンチしてしまったものである。しかしエラー・メッセージにおいてはCALL文におけるSUB1が存在しない事になっており，単純なミスのため気のつきにくいSVB1における誤りについては全くふれられていない。もちろん見掛け上の誤りはCALL SUB1の文である。図4及び図5のいずれの例においても，誤りの場所（フォートラン・ステートメント・番号）が示されていないので，バッグの追跡は困難をともなう。

このようにリンク・ロード時のエラー・メッセージはプログラマーが意図しているようなプログラム構造と同じ方法で，エラー・チェックがなされているとは限らない。したがってエラー・メッセージの解訳には，プログラムの構造を念頭においてメッセージの文面だけでなく，それに関係した近傍についても同時にしらべなければならない。(表2参照)

実行時

この段階のエラーは，①宣言文等に対する約束違反，②計算機の性能を超えた計算・処理及び③計算機・機能の使い間違い等にに関するものである。これらのバッグは，いくつかのステップを通過した後に現われる場合が多いので，バッグ存在の発見場所とバッグ自体（修正されるべき文）を区別して考えた方がよい。バッグの存在の発見場所は計算機システムにより内部文番号で示される。したがってその時に示された内部文番号を出発点として，バッグの本当の位置をしらべなければならない。出力されるエラー・メッセージについては文献4）を見るとよい。

① 宣言文等に対する約束違反

```
      ⋮
DIMENSION A(3,3)
      ⋮
I=5
J=4
A(I,J)=X*Y
      ⋮
```

図6

宣言文は，変数の性質や配列の大きさを定める。したがってこの定められた約束に違反した場合にはいかなる事態を引き起こすかは予測できない。例えば図6に見るように配列A(3,3)に対してA(5,4)を代入文で定義しようとすれば，プログラムの他の部分を破壊して

```
ADDRESS ERR=10
```

が出力される。そしてプログラムは実行不可能となる。配列添字に零が入れられ A(0, 0) となった場合も同様にプログラムを破壊する事がある。この時のプログラムの壊され方はプログラムの構造によるので予測できない。

その他，これに類する“約束違反”のために生ずる誤りとしては次のようなものがあげられる。(文献2)

a）値を代入していない変数や配列の使用

算術代入文，READ文，あるいはDATA文によって変数や配列に値を入れておかないでそれらを式や文の中で使用したとき，英字名のつづりをまちがえたために値の代入されていない変数を使用したとみなされる場合も多い。

b）引用の誤り

基本外部関数や副プログラムの引用において，引数の型や個数が定義と異なっているとき，あるいは実引数の値が定められた範囲を越えているとき。

c）添字式の値の誤り

配列要素名を用いたときに，その添字式の値がゼロ以下であったり，配列宣言子で定義した寸法を越えてしまうとき。

d）入力データの誤り

穿孔の誤りなどによって，数値のデータの中に英字があったり，整数型のデータの中に小数点がついていたり，数値が大きすぎたりする場合，あるいは，データ・カードの書式仕様と合っていないために，READ文の実行中にデータを読みつくしてしまう場合。

e）欄記述子の指定の誤り

WRITE文によって変数の値を印刷するときに，変数の値が大きすぎてしまって，指定した欄に結果がおさまらないとき。

f）入出力並びの誤り

入出力並びに書いた変数名や配列要素名の型とそれに対応する欄記述子が矛盾しているとき。

② 計算機の性能を超えた計算・処理

ある数を零で割る事は数学的に無意味である。また根号の中味が負になっている実数計算も一般には無意味である。したがってこのような時

ZERO-DIVIDE IN……
NEGATIVE ARGUMENT TO SQRT

等のエラー・メッセージが出力される。

一方，計算機で扱える数字の有効桁数，最大値及び最小値は有限である。したがって例え数学として意味があっても計算機の性能の限界を超える場合には処理不可能となり，エラー・メッセージが出力される。このエラーの典型的例は，

OVER FLOW IN……

である。

さらに計算機は大きさの限られた機械である。したがって入出力等の機能においては，例えばライン・プリンターに見るように，1行当りの印刷文字数は133文字まで，と限定される。この限界を超える場合には

WRITE-FILE 06 INSUFFICIENT RECORD LENGTH

のエラー・メッセージが出力される。

③ 計算機・機能の使い間違い

計算機には，磁気テープ装置，カード・リーダー，ラインプリンター等の周辺機器があり，それぞれにUNIT番号がつけられている。そしてこれらの機器のいずれかを使用する際には必ず対応するUNIT番号を指定しなければならない。例えばカード・リーダーはUNIT番号5でラインプリンターは6である。この時何のことわりもなく

READ(6, 100) X

を実行しようとすれば，物理的に不可能となりエラー・メッセージが出力される。但しREAD(6, 100) 文はフォートラン文法に違反していないのでコンパイル時にはエラーとはならない。

④ その他

計算機より出力されるメッセージとしてそのほかに

CPU-TIME LIMIT
SPU-LIMIT

等がある。これらは各々制限時間に達した場合，出力カード枚数の制限に達した等の運用上の制約によるものである。バッチ・システムではたいていこの様な制限があるので注意を要する。

このようにフォートラン文法として正しくても実際に実行した場合に様々の制約や限界のために実行不可能な状態に直面する。これが実行時のエラーである。したがって，プログラム作成にあたっては自分の使用する計算機システムの機能・性能および運用規則を予め知っておかなければならない。

実行終了後

FORTRANプロセサーにはエラー・チェック機能が備えられている事は上記の通りである。しかし，これらチェック機能によってすべてを検出できるわけではない。したがってプログラ

ム実行後の結果が正しいという保障は，「エラーメッセージがない」というだけでは十分でない。厳密に検討された何組かの異った条件でのテスト・データを用いて，そのテスト結果の正しい事を確認する事が不可欠である。これらテスト・データはプログラムの持つ特性のすべてを検討できるものでなければならない。ここで問題になるのは，最終結果が予期されているものとは違った場合である。我々の手元には処理結果の出力シートと処理時間（ACCOUNT LIST）のみがのこる。したがって以後のバッグ追跡はこの時点からはじめなければならない。

バッグとの関連において問題になるのは出力書式の異常，出力数値の異常及び処理所要時間の異常の３点であろう。**出力書式の異常**については，対応するFORMAT文の書式を検討すればバッグの場所の見当はつく。**出力数値の異常**と**処理所要時間の異常**とは多くの場合相互に密接に関連している。しかし異常の状況については，ほとんど無限の可能性があり得ると思う。ここでは，一般的に通用するであろうと思われる状況について表３に大きく分類してみた。表の中には個々の場合の考え得る可能性について記しておいたので参考にして下さい。

表３　実行終了後に発見されるバッグの特徴

実行終了後に発見されるバッグは，出力数値の異常と処理所要時間の異常の２面から見ることができ，そして各々の異常に対して個々に症状の内容が存在する。ここではこれら２者を異なる次元のものであると考えて別々にその特徴を整理した。しかし現実にはこれら両者を同時に吟味しなければならない。この意味において実際のバッグに対しては以下に示す２つの表を同時に見ることが必要である。（ここに示した例は必ずしも一般的な説明を意図して書いたものではない。これまでの筆者の経験の中で特に強く印象にのこった事やごく常識的でありながら案外見落されるのではないかと思われるものを示した。したがってこれ以外の可能性があり得る事は否定できない。）

処理所要時間の異常

異　常　の　特　徴		
所要時間が短かすぎる	所要時間は妥当	所要時間が長すぎる
GO TO 文や CALL 文等の誤りにより実行の流れが短絡してしまい，必要な部分の計算を実行していない。	実行の流れは正しいが副プログラム文の引数や CALL 文の引数及び COMMON 文等の異りのため変数の転送に異常がある。 変数の使用誤り等。 この部分はむしろ出力数値の異常を重点にバッグをさがすほうが良い。	DO 文の誤り，GO TO 文の誤りなどのため流れが乱れて多数回同じ部分を実行したり不要なプログラム部分の実行を行うために誤りとなる。

出力数値の異常

異常の特徴	注目すべき事項
特徴のある数値 (例えば，すべてゼロであったり入力データと全く同一である等)	流れの乱れのため必要な計算の実行がなされなかったり，副プログラム間のデータの授受がうまくいっていない。未定義の定数や変数を使っている場合も多い。
数値の桁*が大きく違う	四則演算の演算子を間違えている事がある。 (例えば Y＝X＋100.0 を Y＝X＊100.0 と間違える等である) あるいは流れの乱れのために多数回計算が実行され累積された結果を出力している。
正確な値に対して定数倍又は定数差だけ違う（但し出力が多数ある場合のみ）	入力データの誤り又はプログラム中の定数定義の誤り等が原因する事が多い。特にプログラム中で定数倍したり定数だけ差引きされる部分があれば，当然その前後の異常の有無をしらべると良い。混合演算があれば再吟味してみる事。
数値の桁*は同程度だが値が全く違う	これは最も特徴の不明瞭なバッグであるためデバッグは困難である。定数・変数の定義や使用場所の誤り，実行手順の誤り等多岐にわたってその可能性を均等にもつために，あらゆる状況を相定してしらべなければならない。次の例はよくおこる誤りであり又発見の困難なものである （例）過失によるカードの入れ替り 誤り　　正 ⋮　　⋮ X＝X＋2.0＊A　　A＝0.5＊X/B A＝0.5＊X/B　→　X＝X＋2.0＊A Y＝X＊A　（訂正）　Y＝X＊A 左右で結果は違ってくる
精度が悪い	数値計算公式とプログラム間のわずかの不一致が原因している事が多いまた計算過程での桁落誤差に原因する事もあるので再吟味が必要。また例えば連立一次方程式の場合のように，与えた問題が悪条件であるために使用しているプログラムではより高精度の結果を得る事が本質的に不可能な場合もある。

（注）＊ ここで桁といっているのは有効桁数の事ではない。仮数部に対する指数部の意味であり数値の大きさのことである。

4．デバッグの手法

前節においてバッグ及びバッグが原因で生ずる様々の現象について述べた。バッグはその場所と正体さえつかめば，それを修正する事はそれほど困難ではない。そこで本節では，バッグを未然に防ぐための手段及び，もしバッグがある事がわかった場合のその場所と正体を追跡する手法について述べる。これらについては文献 1）にくわしく述べられており，また文献 2)，3）にも初心者向けの解説がなされている。ここでは文献 2）を参考にしながらデバッグの手法についてまとめてみる。

計算機に通す前の注意事項

バッグは1度生ずると非常に発見の困難なものもある。また，計算機を有効に利用する上からも未然に誤りを防ぐのが常識といえる。したがってプログラムを設計・コーディングしてパンチし，計算機に通すまでの段階でぜひともおこなっておかなければならない事項についてここで述べる。

(1) 入出力の書式の設計

入出力の書式はプログラムをつくる最初の段階ではっきりさせておくと良い。これによりテスト・データの誤りや，出力結果の混乱等を避け，不要な失敗を防ぐ事ができる。入力については，読み込むデータの順序と各データに必要なけた数を決め，カード上の配置を考える。あとで穿孔したカードを検査するときのために，データとデータの間には空白を入れて読みやすい形にしておく必要がある。出力についても同様で，印刷の結果が見やすくなるように配慮しなければいけない。そのために，計算結果だけではなく，見出しや簡単な説明をつけたり，空行を入れたりするとよい。

(2) コーディング

流れ図(FLOW Chart)を見ながらそれをフォートランの文で記述してゆく作業（コーディング）の際には，文の表現上の重大な誤りのほかに，ミス・パンチに類する些細な誤りがある。この両者は計算機の側から見た場合には全く同程度の誤りなので，どちらの場合についても軽視すべきでない。後者の誤りを防ぐためには，プログラムを書く際にコーディング用紙を用いてきれいに書く習慣をつけた方がよい。このとき，0とO，1と/とI，2とZ，PとDなどは他人が見ても区別がつくようにはっきりとわかりやすく書いておく必要がある。

次に文の表現上の誤りの防止としては主に次の注意事項をあげることができる。

① FORTRANになれないうちは，文の凝った使い方を工夫するよりも処理の内容をすなおに記述して，わかりやすいプログラムを書くように心がける。つまり，難しい文を使わず，たとえステップ数が長くなっても平易で且つあとでデバッグしやすいプログラムを作ることが大切なことである。効率の良いプログラムを意識しすぎて背伸びする事は禁物である。

② プログラムが見やすくなるように，処理のブロックごとに注釈行を入れる。また，プログラムだけを見てもその内容がわかるように，注釈行を用いてそのブロックでの処理についての説明をつける。

③ 変数名や配列名などの英字名は，問題で扱っているのと同じものを用いるようにし，英字名だけを見てもそれがどのようなものであるかがわかるようにする。

④ 文の番号には，その大きさに意味をもたせるようにするとよい。たとえば，READ文で指定するFORMAT文には1000番から，WRITE文で指定するFORMAT文には2000番から，

各処理のブロックは100番から10きざみの番号を使うようにする。

⑤ 読み込んだデータは印刷するようにする。このようにしておくと，データ・カードの穿孔の誤りや読み込み用に用いた書式仕様の誤りを見つけるのが容易になる。

⑥ 入力データが正の整数とか，絶対値が1.よりも小さいというように特定の性質をもつような場合にだけプログラムが正しく働くようにつくってあることがある。このようなプログラムでは，データを読み込んだら必ず検査を行ない，もし必要な条件を満たしていなかったらメッセージを印刷して，そのあと適切な処置をとるようにしておくことが望ましい。このことは，入力データにかぎらず，計算の途中の結果についてもいえる。

⑦ ＩＦ文で実数どうしの比較をするときには注意しなければならない。例えばIF(X.EQ.0.0)において，もし X＝Y＊Z－2.0 によりXが与えられたものであるならば，計算機の有効桁数が有限であるために常にXにはまるめの誤差がくり込まれ X＝0.0の状態が再現される可能性は少い。このような場合には IF(ABS(X).LT.1.0E－10) 等とすべきである。

(3) 机上デバッグ

プログラムが書けたら，流れ図とコーディング用紙をつき合わせて文法的な誤り，あるいは論理的な誤りを調べる。いくら注意してコーディングしても，何かしらの誤りを犯しているものである。以下に，犯しやすい文法的な誤りを列挙するので，それらについて自分のプログラムを見なおしてみるとよい。

a．算術式あるいは論理式中の型の異なった演算

b．配列宣言子の書きわすれ

c．文の番号の定義のしわすれ

d．文の番号の多重定義

e．つづりの誤り（たとえば，DIMENSION文を「DEMENSION…」と書いたために，コンパイラが文の種類の識別ができなくなってしまう）

つづりの誤りとしては， READ とか FORMAT のように文の種類を示す働きをもつ語のつづりの誤りのほかに，文法上必要なかっこやコンマを書き落したり，文の中で不必要なところに余計な文字を書いてしまったり，英字名のつづりをまちがえたりするものも含まれる。

f．GO TO文，算術IF文で指定したつぎの実行文に文の番号がつけられていない。

g．DOの範囲内で，そのDOのパラメタの値をかえようとする。

h．左右のかっこが正しくつり合っていない。

i．H変換や文字定数 $n\mathrm{H}h_1h_2\cdots h_n$ において，n と文字列 $h_1h_2\cdots h_n$ の長さとが一致していない。

j．DO文のパラメタの書き方が正しくない。

k．添字式の書き方が正しくない

上に示したような文法的な誤りは実際に計算機に通してみれば，コンパイラが診断をしてくれるが，このような誤りは計算機に通すまでもなく，自分でよく調べて，それを検出するよう心がけなければならない。

文法的な誤りを調べたら，つぎはプログラムの論理を確かめるために，プログラムを机の上で追跡し，はじめに考えたアルゴリズムのとおりに働くかどうかを見る。特別なデータを用意しておいて，最初から文を１つずつ追跡し，変数の配列の値の変化を調べるとよい。

論理的な誤りで犯しやすいのは，流れ図の判断の箱をコーディングするときに，IF文の使い方をまちがえて判断後の処理を逆にしてしまうことと，変数や配列要素に値を代入するのを忘すれて，それをあとの式の中で使うことがあげられる。

デバッグを容易にすための方法

小さなプログラムをつくるときには，以上に述べた点に注意して作業をすれば，あとは実際に計算機に通してみて，その結果でどこが悪いかを判断することはできる。しかし，複雑で大きなプログラムになると簡単には誤りがわからないことが多い。デバッグの作業はプログラムを計算機にかけてから始まるものであるが，デバッグについてはコーディングあるいはそれ以前の段階から考慮しておかなければならない。プログラムをデバッグする単位としては，１つのプログラムにおける各処理のブロックとプログラム単位とが考えられる。それらが正しい処理をしているかどうかを調べるには，プログラムの要所にWRITE文を挿入して必要な中間結果を印刷するようにし，あとから処理の流れが追跡できるようにしておくとよい。もし，処理の流れが複雑ならば，変数や配列の値だけでなく，どの処理に移ったかがわかるように各処理のはじめで特定のメッセージを印刷するようにするとよい。

WRITE文とFORMAT文をいちいち挿入するのが面倒ならば，各型の変数または配列に対して，それらの値を印刷するサブルーチン副プログラムを用意しておいて，WRITE文の代りにCALL文を挿入する。自分で工夫をしてデバッグ用の副プログラムをつくっておくと，他のプログラムのデバッグにもそのまま使えるので便利である。このとき，整数型の引数を１つ追加し，デバッグ用の副プログラムが呼ばれたときに，その引数の値によって印刷をしたり，しなかったりするようにしておくとよい。こうしておくと，実際の処理のときには，主プログラムでその引数に印刷が抑制されるような値を代入するだけでよく，CALL文をとり除かなくてもすむ。

なお，フォートラン700においてはデバッグ行と次の９つのデバッグ文がある。

DEBUG文	FLOW文	TRACE文
TRANSFER文	DISPLAY文	TERMINATE文
SUBCHK文	SUBTRACE文	DOCHK文

これらを適切に利用する事により，デバッグを容易にすることができる。

デバッグ

穿孔したプログラムを計算機にかけて正しい結果が得られなかったならば，デバッグの作業に入る。エラー・メッセージとして，印刷して指摘される誤りには，コンパイル時に検出されるものと実行時に検出されるものがあることは前節でのべたとうりである。またエラー・メッセージが例え出力されていなくても単にそれだけでは“プログラムは正しい”とは言えない。そしてたいていの場合何らかの誤りが潜んでいる。

これらの誤りも大半は不注意によるものである。結果が印刷されているときには，それが正しいかどうかを調べる。このとき，誤差についてもよく検討しなければならない。もし結果の誤差が理論上の誤差よりもずっと大きければ，データ，プログラムの論理，およびアルゴリズムとその記述方法をよく調べなければいけない。一般に，正しい結果が得られなかったときは，つぎの順序でプログラムと実行結果を見てゆくとよい。

(1) 入力データが正しく読み込まれていることを確認する。このことを忘すれてプログラムばかりを見ている人がいるけれども，データが間違っていたのでは，いくらプログラムを見なおしても誤りは発見できない。

(2) プログラムのリストと実行結果とをつき合わせる。プログ ラムを追いながら，実行結果を見るとき，計算の途中の結果を確認する。こうすることによって，どこまでプログラムが正しく働いていたかがわかる。もし予期していなかった結果が印刷されていたら，それ以前の正しく働いていたところから，誤った結果を印刷した文までの部分をよく調べる。このような方法をとることによって，プログラムの論理的な誤りを見つけることができる。

(3) もし，プログラムに誤りがないと判断したときには，おかしいと思われる部分を流れ図の上で調べるのがよい。同じ論理を追うにしても流れ図の方が見やすく，わかりやすい。流れ図が正しければ，流れ図とプログラムとをつき合わせてみる。

(4) 1組のデータについての処理が正しくできていたら，別のデータについて処理を行ない，いままで実行されなかった部分についても正しく処理ができるかどうかを調べる。

プログラムが主プログラムと多数の副プログラムからなり，しかも副プログラムの引用が複雑になっていることがある。このようなときには，最初から実行の順を追ってプログラム全体を調べてゆくのは能率的ではない。その場合には，まず個々の副プログラムについてのデバッグをする。そのために，各副プログラムを引用するための簡単な主プログラムとテスト・データを用意する必要がある。各副プログラムを単独に引用したとき正しく働くことが確認できたら，実際の主プログラムを使った総合テストにはいる。このようにすると，あとは主プログラム自身と副プログラム間の変数や配列の受け渡しや，多重の引用による影響を中心にデバッグをすればよいので，作業がらくになる。

5．まとめ

バッグ及びデバッグは，よく例えられているように，人体における病気とその治療に相当する。現在，これらについての系統だった知識や手法はない。多くの経験の中から得られた勘にたよっている。したがってプログラム作成者は，個々のプログラムにおいて最も確からしいバッグの状況を推測し最適の手法でバッグを追跡するしかない。ここでは，バッグに対するこれまでの経験を通してデバッグの際に役立つであるうと思われる事項を稚拙ながら整理してみた。説明の不充分な所も少くなくないと思われるが，この解説がデバッグの際の様々な状況の推測及び最適手法の工夫に役立てば幸である。

参考文献

1）A. R. Brown and W. A. Sampson; *Program Debugging,* Macdonald・London and American Elsevier New York, (1973).

2）浦　昭二；FORTRAN入門・改訂版，培風館（1972).

3）小林竜一；はじめて学ぶ人のために　Basic Fortran，培風館

4）NEACシリーズ2200　FORTRAN 700プログラミング説明書，日本電気株式会社

5）NEACシリーズ2200，操作法説明書，日本電気株式会社