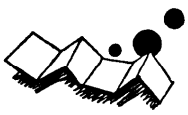
解　説



Pascal とそれ以降の言語設計†

川合　慧††　石畑　清††

1. はじめに

Pascal の特長のひとつは，その言語仕様が，概念整理と実用性という両面においてすぐれたものであることである．どれほど美しい理論体系を持った言語であっても，その具体化が非常に困難であるならば，実際の場では受け入れられない．この逆の場合には，言語が提供する概念の貧困さによって，プログラマに実りの少ない単純作業を強いることになる．

本解説では，まず Pascal の言語仕様を概観し，言語としての全体像を明らかにしたあと，Pascal 以降に設計された諸言語による言語機能の増強および補強について論ずる．Pascal を原点とする言語設計のひとつの流れの中で明らかにされてきた，プログラム言語に対する数々の要求仕様を明確化するのが，本論の目的である．

2. Pascal の言語仕様

本章では，Pascal が高い評価を勝ち得た理由であるその言語仕様について概説する．紙数の関係で，データ型以外の項目については，その内容と特徴を述べるにとどめた．

2.1 データ型

データ型の概念は，計算機の中で取り扱われる情報を，その具体表現によってではなく，プログラマがそれに与えた意味によって把握する手段として，Algol 60 や FORTRAN に導入された．その後，C. A. R. Hoare[3] は構造化による一段と高水準なデータ型の概念を提唱したが，これを現実のプログラム言語の中で具体化したのが Pascal である．Hoare の提唱した型の概念は集合や写像などの数学的概念を基礎としていたが，Pascal には具体化の技法が許す限りそれが取り入れられている．ただし，プログラムの実行効率を落すようなもの，たとえば実行時にしか表現の定まらないデータ型などは省かれ，それを（プログラマが）実現するための，より低水準の道具が導入された．以下に，その各項目を示す．

（1） 数え上げ (enumeration)

それに属するすべての値の列挙によって定義される型である．ユーザが指定した名前そのものが値となる．値には構造がなく，書かれた順に順序づけされる．

type *honor*=(*Ten*, *Jack*, *Queen*, *King*, *Ace*)

これらの値に対しては，直前・直後の値を求めたり，相互比較や（たとえば *honor* 型の）変数への代入をしたりすることができる．別々に宣言された数え上げ型はすべて異なり，それらに属する値の混用は，翻訳時に誤りとして検出される．数え上げ型使用の実際的な利点の1つはこの‘型検査による誤り検出’であり，整数値で値の意味を表現しておく（たとえば *Ten*=10, *King*=13 等々）プログラミングより優れている．

（2） 部分範囲 (subrange)

上下限を指定してその間のすべての値の列挙のかわりとする，一種の数え上げ型である．もととなる型は順序づけられたもの（数え上げ型や整数型）であり，範囲内のすべての値の列挙が可能なもの（たとえば実数型は不可）である必要がある．

type *studentage*=18..25;
　　cardtrio=*Jack*..*King*

部分範囲型は主に配列の添字型として使用されるが，実行時に行われる範囲チェックの機能を利用して防衛的プログラミングの一助とする用法もある．

（3） 配列 (array)

配列は，要素数有限の添字 (index) 型から要素型への写像として定式化される．たとえば ‘*a*[5]=3.14’ は配列 *a* が添字型の値5を要素型の値 3.14 へ写像することを示す．配列値は，すべての添字値に対する写像値を書き並べることで定義される．

† Language designs of Pascal and its successors by Satoru KAWAI and Kiyoshi ISHIHATA (Department of Information Science, Faculty of Science, University of Tokyo).

†† 東京大学理学部情報科学科

添字型は，部分範囲型や数え上げ型のような要素数が有限で順序づけられたものに限る．また，添字型も'翻訳時に表現が確定'されてしまうので，実行時になってから上下限値の定まる動的配列は許されない．

要素型は一般の型でよく，複雑な構造を持つ値も配列の要素となり得る．配列の要素は添字づけによって選択される．配列値全体の（配列変数への）代入が可能である．

```
type value=array [honor] of 0..4;
    matrix=array [1..4] of
                array [0..10] of real
```

（4） レコード (record)

レコード型はいくつかの型の直積であり，その値は要素型の値を並置したものである．選択指定のために，各要素には名前がつけられている．

```
type eval=record card: honor;
                val: −3..5 end
```

上例では *card* と *val* とが要素選択用の名前である．要素型は一般の型でよい．また，レコード値全体の（レコード変数への）代入が可能である．

可変部 (variant part) はレコード型の要素の1つで，複数の型のうちからひとつを選択できる．選択は，同じレコードに属するタグと呼ばれる要素の値，あるいは動的にレコード変数を作り出す時点での指定による．たとえば

```
record case rank: honor of
      Jack: (j: array [1..4] of char);
      Queen: (q: −100..100) end
```

という型では，*rank* という名の要素の値が *Jack* であれば，第2要素は配列型で名前は *j*, *Queen* であれば，部分範囲型で名前は *q* となる．可変部は，複数の型の直和型に対応する概念であるが，これを独立した構造化法としている言語（たとえば Algol 68）とは異なり，型選択用の要素値 (*rank*) の制御はユーザの責任となっている．

（5） 集合 (set)

もとの型の値全体が作る集合の部分集合を値とする型．全体集合の実現が容易であるもののみが，もとの型として許されている．合併や共通部分等の集合演算を単一の機械命令で実行することが意図されている．

```
type hand=set of honor;
    change=set of (y1, y5, y10, y50, y100)
```

Hoare の提案にあった，要素数，全体集合，要素の順次取出しなどの演算は Pascal では省かれている．

（6） ポインタ (pointer)

ほかの型の値への参照を値とする型で，列や再帰的な型の実現，および効率的なデータ表現の目的で導入された．再帰的な型とは自分自身を構造要素の一部に含む型である．たとえば，2分木の部分木はふたたび2分木であり，次のように定義できる．

```
type tree=record case kind: Boolean of
          true: (left: tree;
                 right: tree);
          false: ( ) end
```

可変部は再帰を終結させるためのものである．ところが Pascal では，上例のような直接的な再帰を許さず，より低レベルな概念であるポインタ型とその特殊な値 **nil** による記述法を採用している．上例は

```
type tree=record left: ↑tree;
                right: ↑tree end
```

となり，前例での *kind=false* の場合の値を **nil** で表わしている．実際のプログラミングでは，種々のデータ構造を構築する手段としてポインタが使われる．Pascal のポインタは，参照先の型が定まっていることと，標準手続き *new* によって実行時に作られたデータしか参照しないことが，その特徴となっている．

（7） ファイル (file)

順編成ファイルを簡単にモデル化した型で，一般の型を要素とするバイナリ入出力と，行という概念を持つテキスト入出力とをサポートしている．

```
type master=file of record name: char;
                age: 0..150 end
```

2.2 制御構造

Pascal における実行の制御構造は，順次・選択・反復の3種に明確に分類される．また，選択と反復については複数の機能ごとに専用の構造を用意している．このように整理された制御構造は，実際のプログラミングに際しての確かなよりどころとしての役目を果している．

（1） 順次実行

'書かれている順に'実行される複数の文をまとめる括弧として，Algol 60 から **begin** と **end** とを受け継いでいる．ただし，宣言局所化の機能は除かれた．

```
begin t:=x; x:=y; y:=t end
```

（2） 選択実行

Algol 60 のIF 文のほかに，3通り以上に分岐できる CASE 文が導入された．たとえば

```
case mycard of
```

Jack: …;
Queen: …;
King: … **end**

では，*mycard* (*cardtrio* 型) の値によって3個の選択肢のうちのひとつが選ばれ，実行される．

(3) 反復実行

これには，終了の制御を論理条件で行う WHILE 文と REPEAT 文 (条件判定の場所が異なる)，および制御変数で行う FOR 文とがある．

while $x>0$ **do** $x:=a[x]$;
repeat *read* (n) **until** $n=0$;
for $i:=-20$ **to** 40 **do** $a[i]:=0$;

FOR 文の制御変数 (上例の i) の変化のきざみは，整数では±1 (−1は **downto** で示す)，数え上げ型では直後 (あるいは直前) の要素に限られており，Algol 60 の複雑な変化指定はおろか，FORTRAN の1以外のきざみ指定と比べても，かなりの機能縮小となっている．

for *card*:=*Ace* **downto** *Ten* **do**…

(4) GOTO 文

以上の制御構造と手続き呼出しだけではうまく記述できない場合もかなり存在する．それで，最も低水準な制御である GOTO 文が生き残っている．行く先であるラベルは整数に限られ，しかも前もって宣言しておかねばならない．制御構造の外から内への飛込みが禁止されているのは言うまでもない．

2.3 宣　言

プログラマが使用する各種の要素，たとえば変数や手続きなどを使用する前に宣言しておくというのは，Algol 60 以来の伝統である．これは，書き誤りによるエラーの防止や翻訳手続きの効率化などの面で，欠かせない方式となっている．ただし，Pascal における宣言 (や定義) は，実行されることがない．すなわち，すべての宣言情報は翻訳時に決定される．言いかえれば，翻訳時に不定であるものは Pascal では宣言できない．これは，実行効率や翻訳処理系の効率と，動的な宣言の重要性との兼合いから決められた．

Pascal における宣言と定義は，主プログラムおよび手続きの先頭部分で行う．しかも，宣言対象の種類により順番が決められている．その順番は，

ラベル，定数，型，変数，手続きと関数

である．このような，宣言の場所と順番については，制限が強すぎるという意見も多い．

データ型に名前をつけて宣言できることは，Pascal の大きな特徴のひとつである．この機能は，データ型という概念をより一層明確にするのに役立つと共に，ポインタ型の助けを借りて再帰的データ構造を構成することを可能とした．

2.4 手続きと関数

Algol 60 で採用されていた名前渡し引数の機能は，非常に強力なものではあったが効率的な実現は困難であった．Pascal の手続きでは，ほかのいくつかの方法でこれを代用する方式をとっている．

(1) 値引数 (value parameter)

値引数は手続き側に用意された変数で，呼出し時に計算される実引数の値を初期値として受け取る．手続きの本体中では，ふつうの変数として振舞う．

(2) 変数引数 (variable —)

呼出し時に指定された変数を手続き側からアクセス可能とする機構．番地渡しの手法で実現されることが多い．値引数による大量のデータ複写を避けたり，複雑な構造データを結果として返したりするのにも役立つ．

(3) 手続き/関数引数 (procedure/function —)

呼出し時に指定された手続きまたは関数を，呼び出された側で使用する機構．'やりたい事' を渡す引数と思えばよい．

引数としては，ごく少数の例外を除けば，Pascal で使用できるあらゆる型のデータや変数が許される．これは，手続きや複雑なデータ型による抽象度の高いプログラミングを行う上で，特に有用な点である．

procedure *compose* (*x*: *tree*; **var** *y*: *honor*;
procedure *append*); …

この例では，x が *tree* 型の値引数，y は *honor* 型の変数 (**var** が示す) 引数，*append* は手続き引数である．このような '頭書き' は，手続き (関数) 宣言の頭部に置かれ，その後に本体の記述が続く．

関数が結果として返すデータの型は，構造のない型とポインタ型とに制限されている．引数の場合と比べて非常に強いこの制限により，構造データの受け渡しに変数引数が多用されることになり，プログラムの信頼性や読みやすさによくない影響を与えることも多い．

Algol 60 では議論の多かった own 機構，すなわち呼出し時に作られた変数を次回の呼出しまで保存しておく機構は，Pascal には取り入れられなかった．

2.5 そ の 他

(1) 名前の有効範囲の制御

ある名前の有効範囲は，それが宣言された手続き・関数の中だけに制限される．また，内部的な手続き・関数の中での同じ名前の宣言は許されるが，その場合は，外側での宣言内容は内部では無効となる．このような入れ子構造による有効範囲の制御は，Algol 60からの継承である．ただし，Pascal では，入れ子の単位が手続きと関数のみに制限された．

有効範囲制御の例外はレコード型の選択名である．レコード型の選択名は，その型の中で一意的であればよく，外側に同名の変数があったり，その型の要素型が同一名を選択名として持つレコード型であったりしてもよい．これは，レコード型の要素選択に特別な構文形式が使用されているためである．

（2） 記憶域の使用

Pascal では，制御の入れ子構造を実現するスタックと，それとはまったく別の領域に確保されるヒープとによって，プログラムの実行を管理する．ヒープ中のデータは，ポインタを通じてユーザが管理する．ヒープについてのゴミ集めの機能に関しては，Pascal は何も言及していない．個々のデータに関しては，そのサイズが動的に変化することがないことが，翻訳処理の上で重要な点である．可変部つきのレコードの場合も，データを確保する時点で複数のサイズのうちのひとつが選ばれるだけであり，確保された後で伸び縮みするわけではない．絶対番地の使用などの低水準機能は，Pascal にはない．ふつうは絶対番地によって具体化されるポインタ型については，等値の演算のみが定義されている．

3. Pascal 以降の言語設計

Pascal は，その簡潔に整理された概念設計と高い実行効率に裏づけされた実用性とによって，プログラム言語の歴史に一時代を画した．実用的に使えるAlgol系言語の最初のものとも言うことができるPascal の出現は，プログラム言語およびプログラミング手法の両者に，飛躍的な発展を促すこととなった．

Pascal の発表以後も，研究の進展にともなって数数のプログラム言語が設計・開発されたが，多くのものが Pascal に範をとっている．そして，発表当初から指摘されている Pascal の不備や欠点などに対し，さまざまな改良がなされてきた．また，その後の言語研究によって形成されてきた諸機能，たとえば並行処理や例外処理の機能などが言語に組み入れられるようになってきている．

Pascal 系の言語の代表的なものは次のようなものである．

Concurrent-Pascal[1] は，Pascal の仕様に並行処理を導入したもので，高水準言語による OS 記述の目的で開発された．SOLO と呼ばれる OS が，実際にこの言語によって作成された．

Modula[16] も同様の目的で設計された言語であるが，比較的小型の計算機システムを対象とし，Pascal の仕様をかなり縮小したうえで，多重プログラミング用の機能を導入している．この言語の使用経験などをもとにして後に開発されたのが Modula-2[17] である．Modula-2 の設計思想は Modula のそれとは大分異なったものとなっており，並行処理がコルーチン機能で代用されているほか，Pascal の機能がほとんど復元され，拡張も行われている．

Euclid[9] の設計目的はほかの言語とかなり異なっており，プログラムの検証可能性が重要視されている．そのため，自動的な検証をやりにくくするもの，たとえば関数の副作用などは，言語仕様中で禁止されている．

Mesa[13] は大規模ソフトウェアを書くための実用的な言語として開発された．実際の場で必要とされる多くの機能を含み，システム開発を支援するための専用の制御言語をも持っている．

Lis[6] も Mesa と同種の目的を持っている．多人数による共同開発を意識した言語仕様となっており，ライブラリの管理などの機能も含んでいる．

Ada[14] は米国国防省用の言語で，1980年7月に最終仕様が決定した．総合的な言語を目ざしており，大型から超小型までのすべての計算機システムが対象となっている．これまでに示した言語の仕様のほとんどを含む強大な仕様となっている．

Pascal 系の言語とはあまり言い難いが，抽象データ型という概念を中心として設計された言語として Clu[11] がある．言語仕様の多くの項目で，Pascal 系の言語と共通の目的が見出される．

4. 言語仕様の最近の動向

この章では，最近の言語，特に Pascal 系の言語の仕様を，3章で述べた各言語を中心に解説する．**4.1** では Pascal から継承された仕様，**4.2** では Pascal 以後の新しい言語仕様を中心に述べる．言語仕様に関する議論を目的とし，個々の言語の紹介は意図していない．

4.1 Pascal の仕様に対する批判と改良

4.1.1 データ型

データ型の基本の考え方は各言語とも Pascal のものを継承している．しかし，細部に注目するとそれぞれの設計方針を反映して種々の変化が見られる．パラメータ付きの型，抽象データ型など型に関する新しい話題も多い．

一般に式の演算においては型の一致が要求されるが，この判定に2つの流儀がある．1つは型の名前による方法，もう1つは型の構造による方法である．

```
type a=array [1..10] of integer;
     b=array [1..10] of integer;
```

a と b は，前者によれば異なる型，後者によれば同じ型である．Pascal はいずれの流儀をとるか不明確であったが，最近は名前による判定法を採用する言語が多い．これは型の偶然の一致を排除し，より厳密なチェックを行おうとする言語設計上の基本的態度にもとづくものである．またこのことは型に名前をつけて使うことを奨励することにもなる．特に Ada では名前を持たない型はほとんど使えないようになっている．

このことと関連して Ada には，ある型の属性をすべて受け継ぎながら，これとは異なる型を作る derived type の機能がある．可能な操作等を共用することによって不必要な複雑化を避け，気楽に新しい型を導入できるようにするのが目的である．異なる目的には異なる型を使うという方針にそったものと言えよう．

Pascal では配列の上下限など型に関する情報は，翻訳時に定めるのを原則とした．このことは実用的なプログラムを作る立場から柔軟性に欠けるという批判を受けた．たとえば，行列演算ルーチンを作ることを考えれば，少なくとも手続きの引数には任意の長さの配列を渡し得ることが望ましい．

最近の言語では配列の大きさ等を実行時に定めるものがほとんどで，記述の柔軟性が向上している．これにともなって型に関する情報 (attribute) を値として返す機能が使えるようになっている．

さらに Lis, Mesa では array descriptor の概念を導入してより自由な配列操作を可能にした．array descriptor は配列の実体を間接に示す一種のレコードで，配列の上下限やアドレスからなる．これの各要素を適当に操作することによって自由な配列操作が可能となるが，ポインタを介しているため誤りを検出しにくい弱点がある．Clu の配列は実行時に伸縮できる．これはほかの言語とは全く異なるメモリ管理にもとづくもので同列に比較はできない．

型の持つチェック機能を保ちながら柔軟性をねらう新しい考え方がパラメータ付きの型である．これは配列の上下限，可変部つきレコードのタグ等を型に対するパラメータと考えるものである．型の宣言は型の骨格のみを定めたものとなり，変数宣言等で型を使う際，実パラメータを与えて型の実体を決定する．演算は骨格としての型に対して定義されるため，パラメータの異なる変数等を同一のプログラムで処理できる．手続きに引数として渡すときは，実引数のパラメータが仮引数に自動的に受け継がれるようになっている．

パラメータ付きの型は Euclid で導入され，Ada に受け継がれた．Ada ではパラメータが不定のものを型，それに実パラメータを与えて固定したものを副型と呼んで区別している．Ada ではパラメータとは称していないが記法上からも共通点が見られる．

可変部つきレコードは多くの言語でそのまま採用されている．ただ可変部中の要素への不当な参照を防ぐためのチェックが厳しくなっている．特に Euclid, Clu ではレコードの可変部は特別の CASE 文の中でしか参照できない．この CASE 文はタグの値によって分岐するものである．各分岐肢ではタグの値が固定されるため，可変部中の要素の参照の正当性を翻訳時にチェックできる．

ポインタの扱いには2つの流儀が見られる．Lis, Mesa のようにシステム記述を志向した言語では，ポインタをアドレスと同一視し，通常の変数をもポインタで指せるようになっている．一方 Euclid 等では動的変数の割り当て場所を型単位にユーザが指定できるようにし，ポインタ使用の危険を軽減しようとしている．このことは動的変数の割り当て，消去をユーザが制御できるようにしようとする傾向にもつながっている．

4.1.2 制御構造

制御構造は言語設計において必ずしも最重要な分野とは言えないが，最も目立つ分野であるため議論が多いようである．最近の制御構造の傾向を箇条書きにしてみよう．

・制御文の終わりを示す閉じ括弧，たとえば **if** に対する **fi** や **end if** を要求する言語が多い．これは構文上のあいまいさをなくし，不要な **begin-end** を追放する意味から当然の方向であろう．

・IF 文において **else if**…**else if** のような形が多く用いられるため，**elsif** ないし **elseif** をキーワー

ドとして組み込むことが多い．

・CASE 文で選択子の値がどのラベルとも一致しなかった場合の処理を others 句で書けるようにすることが多い．Mesa では CASE 文のラベルに単なる定数だけでなく比較式の一部を書けるようになっている．

繰り返しからの脱出の判定を Pascal ではループの先頭 (WHILE文) または最後 (REPEAT 文) で行う．これは制御の流れの把握を容易にするエレガントな方法であるが，この2つだけではうまく表現できないケースも多い．このため最近では EXIT 文によってループ中の任意の場所から脱出できるようにした言語が多い．特に Euclid は繰り返しを無限ループの構文に限り，脱出の判定を EXIT 文のみによっている．

サーチなどの繰り返しの場合，ループ脱出後の後処理が脱出の種類によって違うことがある．これをあらためて IF 文で判定するのは2度手間となる．これを避けるため Mesa, Clu ではループに付随して後処理部が設けられるようになっている．後処理部は CASE 文のような形式をとり，脱出の種類によって場合分けされる．

FOR 文は WHILE 文や REPEAT 文に比べて，繰り返しの制御に関する操作が繰り返しの本体と区別して書けるところに特色がある．たとえば

```
p:=a[i];
while p<>nil do
  begin
    .....
    p:=p↑.cdr
  end
```

において，$p:=a[i]$ と $p:=p\uparrow.cdr$ は繰り返しの制御に関する操作で，繰り返しの本体中で行われていることとは区別して考える必要がある．FOR 文はこのような繰り返し制御を制御文の中に取りこんでしまった点が使いやすさの1つの要因であろう．この考えをもとにして，FOR 文を一般の繰り返しにも適用できるようにしようとする試みが iterator である．iterator は繰り返しの制御に関する操作を1つのモジュール (**4.1.3** 参照) にまとめたものである．たとえば上の例をEuclid では次のように書く．

```
module listtraverse (initialvalue: ptr)
    exports (value, stop, next)
    var value: ptr; var stop: Boolean
  procedure next=
      begin value:=value↑.cdr
        stop:=(value=nil) end next
  initially begin value:=initialvalue;
        stop:=(value=nil) end
end listtraverse
```

繰り返しの本体は iterator を参照することによって簡単に書くことができる．

```
for p in listtraverse (a[i])
    loop……end loop
```

制御に関する操作が1つにまとまって読みやすくなっているのが見てとれよう．iterator は一連操作の抽象としての手続きと同様の意味で，繰り返しに関する抽象化と言われることがある．

4.1.3 宣 言

Pascal 流の宣言機構はほとんどの言語でそのまま採用されている．ただし，型，変数，手続きの順に宣言すること，定数の値が翻訳時に定まること，などの制限は除かれる傾向にある．また Pascal が宣言を手続き単位に限ってしまったことが反省され，Algol 流のブロックを復活させている言語が多い．

Pascal では Algol 60 と違って，宣言の効果がテキスト上宣言より前の位置には及ばない．この制限された有効範囲規則は，処理系の作りやすさを優先させた結果であるが，処理系の作りやすい言語は人間にとっても理解しやすいという主張や，処理系の効率重視などの理由から，最近の Pascal 系言語のほとんどに採用されている．

最近の言語では，宣言や名前の使用に関する規則をより厳密に定め，不用意な誤りをできるだけ翻訳時に検出しようとする傾向が見られる．逆に言えばプログラマが意図をより明確に表現することを要求されているとも言える．たとえば Lis や Clu では宣言された名前を入れ子の内側で再宣言することが禁じられている．またさらに極端な例として Euclid ではグローバルな名前を直接参照できない．ブロック内で使われるグローバルな名前はあらかじめ import 宣言で指定しておく必要がある．

このような例からもわかるように最近の言語では，Algol 流のブロック構造による名前の有効範囲規則に対する反省が見られる．Algol 流の欠点を一言で言えば，名前の有効範囲が広すぎることである．たとえば

```
var hiddenrandom: integer;
    .....
procedure random (var value: integer);
```

```
    begin ····· end ;
begin
    {initialize hiddenrandom}
    ·····
end
```

という例において，変数 *hiddenrandom* は手続き *random* と初期化の部分のみで使われるが，ブロック構造のみによる有効範囲規則ではグローバルに宣言せざるを得ない．このため関連する部分がテキスト中で分散してしまい，プログラムの読みやすさの点で好ましくない．また *hiddenrandom* のような変数をプログラム中の不必要な部分から参照可能にしておくことは，不用意な誤りのもととなる．

このような問題を解決するためにモジュール化の考え方が採用されている．モジュールはプログラム中の論理的に関連する部分を一まとめにしたものである．モジュール内で宣言された名前は，export 宣言されたもの以外，外からは参照できない．こうすることによって局所的な情報の参照の範囲が限られ，プログラムの読みやすさ，誤り検出の両面で好都合となる．同じ例を Modula のモジュールを使って書いてみよう．

```
module randomnumber ;
    export random ;
    var hiddenrandom : integer ;
    procedure random (var value : integer) ;
        begin ····· end random ;
begin (*initialize hiddenrandom*)
                        end randomnumber ;
```

モジュールは名前の有効範囲を制限するだけで，手続きのようにプログラム実行の単位となるわけではない．モジュールの実行部は，それを含む手続きが起動されたときに実行される．

上例のようなモジュールの場合，モジュールの外から参照できる名前の実体の宣言は，モジュール全体に分散して書かれる．このことは，モジュールの利用者にモジュール全体を読むことを強いることにつながり，モジュール内の細部を隠すという本来の目的に矛盾することにもなる．

このため，モジュールを仕様部と実現部に分けることがある．仕様部には外に見せる名前の仕様，たとえば手続きの引数を書く．型や手続きの実体は実現部に書き，この部分は外からの参照を許さない．モジュール外のプログラムは仕様部のみを参照して開発することができる．例として Ada におけるモジュール（パッケージ）を示す．

```
package randomnumber is
    procedure random (value : out integer) ;
end randomnumber ;
  ·····
package body randomnumber is
    hiddenrandom : integer ;
    procedure random (value : out integer) is
        begin ····· end random ;
  begin -- initialize
  end randomnumber ;
```

前半が仕様部，後半が実現部である．仕様部と実現部を分割することは，トップダウンプログラミングや分割翻訳にも適しており，Lis, Modula-2, Mesa の各言語でも採用されている．

4.1.4 手続き，関数

手続きでは，引数の渡し方に変化を見ることができる．Modula-2 では Pascal 同様の値引数と変数引数，Euclid 等では定数引数と変数引数を採用している．定数引数は初期の Pascal にもあった考え方で，手続き内で引数に代入することを許さない．Lis や Ada では引数によるデータのやりとりの方向を強調して，in, in out, out の 3 種類を用意している．Mesa は変数のアドレスをポインタとして扱えるため，値引数のみとしており Algol 68 の考え方に近い．各言語とも引数機構をコピー渡し，番地渡しのいずれで実現するかについては規定しないのが原則であるが，Ada では構造を持たない型についてはコピー渡しとすることが定められている．

手続きを引数とするための Pascal の記法は未成熟であるため後続の言語では採用されていない．Lis, Mesa では手続きを 1 つの型ととらえるため，普通の引数と同様の手順で渡すことができる．

Mesa, Ada では実引数をキーワード方式で書くことを許している．この方法は従来ジョブ制御言語で使われていたもので，次のような形をとる．

```
search (table=>v_table, item=>p)
```

この方法によると引数名が明示されるため，プログラムの文書性が向上する．引数の順序は任意であり，省略された引数に対する標準値を手続き宣言の際に指定できるようになっている．

手続きに関する最近の話題として別名と関数の副作用の問題がある．別名とは 1 つの変数の実体を 2 つの異なる名前を通して参照することで，次の例が代表的

なケースである.

```
procedure p (var a, b: integer);
    begin …… end;
begin p (x, x) end
```

手続き p の中で変数 x は a, b 2つの名前で参照される.

別名，副作用ともプログラムの理解を妨げ，正しさの証明を困難にするばかりでなく，処理系の作り方によってプログラムの結果が変わるという欠点をもつ．近年プログラムの検証の立場からこれらの問題が重視されるようになった.

ここにとり上げた言語のうちで，これらを言語仕様の中で禁止した最初の言語が Euclid である．Euclid の文法書は別名をエラーとして検出することをコンパイラに要求している．Ada も暫定版では禁止していたが，改訂版ではこれらを使ったプログラムの結果を保証しないとする立場に後退してしまった．コンパイラの技術的問題や目的プログラムの効率を考慮した現実的な妥協と言えよう.

4.2 Pascal 以後の新しい言語仕様

4.2.1 並行処理

OS などのシステム記述用に高級言語を使う機会が増えており，並行処理の機能を言語レベルで提供する傾向が強まっている[18]．ここでとり上げた言語の中では，Concurrent Pascal, Modula, Mesa, Ada が並行処理機能を持つ.

並行処理において中心となる問題は，タスク間の通信，同期，およびそれにともなう相互排除の実現方法である.

Ada 以外の各言語はこのための道具としてモニタ[4]を採用している．モニタはタスク間で共有される変数とこれを操作する手続きをモジュールとしてまとめたものである．モニタ内には1つのタスクのみが入ることを許され，これが終わるまでほかのタスクは待たされる.

モニタによる方法では，共有変数を媒介としてタスク間通信を行う．これに対して Ada では引数を通してタスク間で直接通信を行い，ランデブと呼ぶ方法によって同期をとるのが原則である．各タスクには通信を受けるためのエントリが用意される．ほかのタスクからのエントリ呼び出しとこのタスクにおける受け付けの実行が一致した時通信が行われる．一方のみが実行されたときは他方が実行されるまで待ちの状態に入る.

タスクの生成，消滅の記述方法は，プログラムテキストからタスクの個数が静的に決まるもの，タスクの生成を指定する文を持つものなどさまざまである．Ada ではタスクを1つのデータ型と考え，タスク型の変数の割り付けによってタスクを生成させるという考え方をとっている.

4.2.2 分割翻訳

大規模プログラム開発の際，役割分担やプログラム管理を円滑にするため，1つのプログラムを分割して開発することが必要である．このためプログラムの各部分を単独に翻訳する機能が言語および処理系に要求される.

FORTRAN など初期の言語でも副プログラム単位に翻訳することができた．ただこの場合，引数の対応などプログラム単位間にまたがる論理的誤りを検出することができず，プログラムの信頼性の1つの障害となっていた．ここで述べる分割翻訳はこれと異なり，翻訳単位間にまたがる文法チェックを完全に行う方式のことである．分割翻訳したときとプログラム全体を1度に翻訳したときの文法チェックは厳密さにおいて何ら変わるところはない.

分割翻訳は Lis, Modula-2, Clu, Mesa, Ada で採用されている．各言語の仕様には共通点が多いため，ここでは Ada についてのみ紹介することにする.

Ada における翻訳の単位は副プログラムまたはパッケージである．Ada の分割翻訳はトップダウンプログラミングの支援とライブラリの構築の2つの目標のもとに設計されている.

トップダウンプログラミングでは，上位のプログラムは下位のプログラムの仕様のみを参照して開発し，下位のプログラムはこれとは独立に開発する．このことを強調するために下位のプログラムの本体を分割して翻訳するのがトップダウン方式の分割翻訳である.

```
procedure main is
    procedure p (x: in integer) is separate;
begin
    ……
end main;
```

separate と指定された副プログラムやパッケージは，引数などの仕様のみが書かれる．それぞれの本体はこのプログラムとは独立に開発，翻訳される．これらの独立に翻訳されたプログラムは，論理的には主プログラムの一部であり，ほかのプログラムからは参照できない.

一方ライブラリは共通部品の蓄積をねらういわばボトムアップ的な考え方である．ライブラリ中のプログラムは，副プログラム，パッケージとも仕様部と実現部に分けて翻訳され登録される．ユーザプログラムでは次のような形式でライブラリを参照する．

```
with real_operations, gaussian_elimination;
procedure user_program (……) is
    ……
```

with 節で指定されるのがライブラリ名である．

分割翻訳の際，型などのチェックを厳密に行うため環境ファイルを使用する．このファイルはシンボルテーブルなど翻訳や整合性チェックに必要な情報を含むものである．各単位の翻訳の後，オブジェクトコードのほかに環境ファイルが作られ，この単位を参照するほかの単位の翻訳の際に使われる．

このことからわかるように，各単位の翻訳の順序には一定の制限が加えられる．一般に参照されるプログラムを先に翻訳することが必要である．またプログラムを修正，再翻訳したときも同様の条件でほかの単位の再翻訳が必要になる．

4.2.3 例外処理

例外処理とはプログラムの実行中に発生する例外的な事象に対する処理を記述するための機能である．例外が発生した場合，所定の例外処理部に自動的に処理が移るのが通常の形態である．

例外的な事象の定義は明確でないが，要約すれば通常のプログラムの流れとは異なる処理を必要とする事象ということができる．この中にはオーバフローや配列範囲外参照などのエラーのほかに，稀ではあるが十分予想される事象，たとえばファイル終わり検出などを含む．また制御文の一種として，プログラマが定義した例外を陽に発生させる機能を持たせることも多い．この意味で例外処理は単なるエラー処理にとどまらず，制御の流れの特殊な記述形式の1つと言うことができる．

例外処理はこれまでの言語ではほとんどとり上げられていない．わずかに PL/I の ON 条件が相当するが，例外処理部の登録が動的に行われるため，GOTO 文と同様に無秩序なプログラム構成を招きやすかった．これに対して，例外処理部の担当範囲がプログラムテキストから静的に定まるのが最近の例外処理の傾向である．ここでは Ada の例外処理を中心に最近の例外処理機能について解説する．

次にあげるのは Ada における例外処理の例である．

```
1  procedure linear_equation (……) is
2      ……
3      singular : exception;
4  begin
5      ……
6      if v < eps then raise singular; end if;
7      ……
8  exception
9      when singular | numeric_error=>
10         put ("matrix is singular");
11     when others=>
12         put("fatal error"); raise error;
13 end linear_equation;
```

この例において，*singular* はユーザが定義した例外名，*numeric_error* はオーバフローなどの演算エラーによる例外である．RAISE 文はユーザ定義の例外を発生させる．8行目の **exception** 以下が例外処理部である．

例外処理部は副プログラムやブロックなどのプログラム単位ごとに書くことができる．プログラム単位の中で例外事象が発生すると，処理は中断され制御は対応する例外処理部に移される．例外処理部は CASE 文と同様の形式をとっており，例外の名前によって場合分けされ適当な処理が行われる．それが終わった後，プログラム単位の呼び出し点に戻って正常の実行が続けられる．いったん例外処理部に入った後，例外の発生した地点に制御が戻ることはない．

プログラム単位中で起こり得るすべての例外の処理を対応する例外処理部で記述する必要はない．処理が記述されていない例外が発生すると，そのプログラム単位の実行は打ち切られ，このプログラムを呼び出した点で同じ例外が発生する．このことを例外の伝播と言う．このことは例外処理を不必要に複雑にしないために重要である．各例外はそれぞれ適当なレベルでのみ処理すればよいため，プログラムの論理的階層構造を保つことができる．

Ada のほかに例外処理を持つ言語は Clu および Mesa である．Clu の例外処理は Ada とほぼ同じ仕様となっているが，例外名に引数を持たせることができる点が異なっている．Mesa では例外発生地点に制御を戻すなどの複雑な制御が可能である．

例外処理機能を言語の中に含めることの効用をまとめると次のようなことがあげられる．

1) プログラムの信頼性を向上させる．局所的なエラーに対して適当な処置を行うことによって，プログラム全体が動作不能になることを防ぐことができる．

2) エラー処理の通常の方法，たとえば手続き呼び出しごとにフラグを調べる方法に比べて，いちいち条件を判定する必要がなくなる分だけ効率が向上する．

3) 例外処理をまとめて記述することによりプログラムの発生しうる例外条件が容易に読みとれるようになる．これによってプログラムの文書性が向上する．

4.2.4 抽象データ型，generic

ここではより強力な言語をめざす2つの言語仕様，抽象データ型と generic について述べる．

（1） 抽象データ型

そもそも高級言語は，物理的実体を隠して，より抽象化された世界で考えることを可能にするのが大きな目的である．データ型の考え方も，メモリ上の配置を意識することなしに論理的操作のみを考えるというのが動機となっている．この考えをさらに進めて，より高度の抽象化をねらったのが抽象データ型である．

通常，ある型がどのように構成されているか，たとえば配列であるかどうかということはプログラム作成の際，常に意識されている．しかし，型をデータ構造の抽象と考えるとこの方法には難点がある．たとえば，列を配列で実現するかリスト構造で実現するかの区別より，列の生成，連結等どのような操作が可能であるかの知識の方がプログラム作成上重要である．このような見地から，データの構成法とそれに対する基本操作を合わせて定義し，これを型とする考え方がある．これが抽象データ型である．

たとえば Clu による複素数型の定義は次のようになる．

```
complex=cluster is add, ……
    rep=record[x,y: real]
    add=proc (a, b: cvt) returns (cvt)
        return (……)
    ……
end complex
```

1行目で宣言された型名と演算の名前のみが外側から参照可能である．2行目の **rep** 以下で指定される型の構成法を参照することはできない．記法はほかの言語のモジュールとよく似ているが，型として，たとえば変数の宣言に用いられる点が異なる．

ユーザは型の内部構造を意識することなく，型によって与えられた基本操作を用いてプログラムを書くことができる．この意味でデータの構成法と無関係な，データの抽象的な性質をもとにプログラミングを行うことができる．これが抽象データ型と呼ばれるゆえんである．

抽象データ型の考え方は，**4.1.3** で述べたモジュール化とも関連を持つ．Ada では型とそれに対する演算を1つのパッケージにまとめる技法が推奨される．この場合，パッケージの外で得られる情報は型の名前とそれに対する演算のみであり，抽象データ型に近い効果を得ることができる．

（2） generic

generic は型や手続きをパラメータとするプログラムを書くための機能である．Ada における例を見てみよう．

```
generic
    type elem is private,
package stack is
    procedure push (e: in elem);
    procedure pop (e: out elem);
end stack;
```

型 *elem* が *stack* に対するパラメータである．generic な手続きやパッケージは置き換え用のテキストとしての意味しか持たず，通常のもののように呼び出したりすることはできない．これに実パラメータを与えて初めて通常の手続きやパッケージとなる．たとえば

```
package integer_stack is stack (integer);
```

元の定義の中のパラメータは実パラメータにテキスト上で置き換えられる．

スタック操作のようにいろいろな型について共通な仕事は少なくないが，従来の言語では実際にあらゆる型に適用可能なルーチンを書くことはできない．型の制約に矛盾するためである．このことはプログラムの重複を招くことに通じ，プログラマに無用の負担をかけることになる．このような欠点を除き，型の概念を生かしながら汎用のルーチンを書くことが generic の主な目的である．

generic はマクロの技法を使って翻訳される．このため処理系の負担は軽く，目的プログラムの効率をそこなわない．generic に限らず，最近の言語設計では処理系による効率のよい処理が常に念頭におかれている．

4.2.5 低水準記述機能

ここでとり上げている言語はすべていわゆる高級言語であるが，Clu を除く各言語が機械に密着した低

水準の記述を可能にしていることは注目に値する．システム記述等の用途が増え，効率や計算機の機能の活用，記述の柔軟性等の要求から，機械語レベルの記述が必要とされていることの現われと言えよう．

各言語に採用されている低水準記述機能を列挙すると次のようになる．

1) 型の表現の指定．データを主記憶中でどのような形式で表現するかをワード，ビットを単位として指定する．

2) 変数を指定された絶対番地に割り当てる機能．

3) 型整合性のチェックをはずし，ある型のデータをそのビット表現のままで別の型として扱う機能．

4) ほかの言語で書かれた手続きを参照するための機能．

5) 機械語命令を記述する機能．

また手続きの in line 展開やデータの圧縮など最適化に関する指定を可能にしている言語が多い．

これらの機能はプログラムのポータビリティをそこねるなど，高級言語の目的と矛盾する点を含む．このため各言語とも，特殊なモジュール内でのみ使用可能とするなど，乱用を防ぎ，使用されている部分を局所化するための工夫がされている．

5. おわりに

本解説では，Pascal において導入され確立されたプログラム言語の諸概念と，その後開発された Pascal 系' 諸言語によるそれらの洗練と改良，および，言語に対する機能要求としてその後新設されてきた諸項目のそれぞれについて概観した．プログラム言語の仕様や実用性については，本論で示した概念やその背景をぬきにしては論ずることができない．

紙数の関係で，多くの分野においてはその主な項目を示すことしかできなかった．以下に掲げる参考文献が，さらに詳しい項目理解の助けとなれば幸いである．

参考文献

1) Brinch Hansen, P.: Concurrent Pascal report, CIT-IS-TR 17, Information Science, California Institute of Technology (1975).

2) Goodenough, J.B.: Exception handling: issues and a proposed notation, CACM Vol. 18, No. 12 (1975).

3) Hoare, C. A. R.: Notes on data structuring, in Structured programming, Academic Press (1972).

4) Hoare, C.A.R.: Monitors: an operating system structuring concept, CACM, Vol. 17, No. 10 (1974).

5) Hoare, C. A. R.: Communicating sequential processes, CACM, Vol. 21, No. 8 (1978).

6) Ichbiah, J. D. et al.: The system implementation language Lis, Technical Report 4549 E1/EN, Compagnie Internationale pour l' Informatique (1974).

7) Ichbiah, J. D. et al.: Rationale for the design of the Adap rogramming language, SIGPLAN Notices, Vol. 14, No. 6, Part B (1979).

8) Jensen, K. and Wirth, N.: Pascal: user manual and report, Springer (1974).

9) Lampson, B. W. et al.: Report on the programming language Euclid, SIGPLAN Notices, Vol. 12, No. 2 (1977).

10) Liskov, B. et al.: Abstraction mechanisms in Clu, CACM, Vol. 20, No. 8 (1977).

11) Liskov, B. et al.: Clu reference manual, MIT computation structures group memo 161 (1978).

12) Liskov, B. and Snyder, A.: Exception handling in Clu, IEEE Transactions on Software Engineering, Vol. SE-5, No. 6 (1979).

13) Mitchell, J. G. et al.: Mesa language manual version 5.0, CSL-79-3, Xerox Palo Alto Research Center (1979).

14) Reference manual for the Ada programming language, U. S. Department of Defense (1980).

15) Wirth, N.: The programming language Pascal, Acta Informatica, Vol. 1, No. 1 (1971).

16) Wirth, N.: Modula: a language for modular multiprogramming, Software-Practice and Experience, Vol. 7, No. 1 (1977).

17) Wirth, N.: Modula-2, Berichte Nr. 32, Institut für Informatik, Eidgenössische Technische Hochschule (1980).

18) 古谷立美：高級言語による並列処理の記述，情報処理，Vol. 21, No. 9 (1980).

(昭和55年11月7日受付)