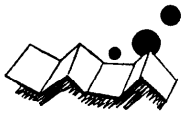
解　説



Ada の概要†

和　田　英　一††

1. はじめに

図-1 は Ada[1]*で書いたパスカル三角形のプログラムである．**図-2** は Pascal で書いた同じプログラムである．まずこのふたつを比較しながら，Ada のプログラムがどうなっているか説明する．

Ada のプログラムはコンパイル単位 (compilation unit) という形をとる．それは副プログラム (subprogram，手続きと関数のこと．ここでは手続き本体) の前に with クローズ (with clause) や use クローズ (use clause) がついたものである．今の場合は「**with** TEXT_IO；」が with クローズ，「**use** TEXT_IO；」が use クローズである．Pascal のプログラムでは，標準の定数，型 (この例では *integer*)，変数 (*output*)，手続き (*write*, *writeln*) や関数はすべて，プログラムの外側で定義，宣言してあることになっているが，Ada では標準の型，手続きなどはいくつかのパッケージ (package) に分かれて宣言してあり，コンパイル単位をコンパイルしようとすると，すぐ外側に無条件でつけ加わるのは STANDARD という名のパッケージだけで，ここではその中で宣言してある型名 INTEGER や整数の演算は使うことができる．しかしその中にはテキスト入出力の手続き類はなく，それらは TEXT_IO という名の別のパッケージに求めなくてはならない．例題の with クローズはこのパッケージを取り込むことを指示する．パッケージ TEXT_IO の中で宣言した手続き，たとえば PUT を PASCAL_TRIANGLE の手続き本体の中から使うには TEXT_IO.PUT のようにピリオドによる選択形 (selection) を書けばよいがこれはわずらわしい．例題の use クローズはパッケージで宣言した名前を，選択形にせず，じかに書けるようにするものである．これにより PASCAL_TRIANGLE の中で，SET_COL, PUT, NEW_LINE がそのままの形で書けるようになった．

```
with TEXT_IO; use TEXT_IO;
procedure PASCAL_TRIANGLE is
H: constant INTEGER: =10; -- total height
W: constant INTEGER: =4;  -- unit width
function C(N, I: in INTEGER) return INTEGER is
begin if I=0 or I=N then
  return 1;
else
  return C(N-1, I-1)+C(N-1, I);
end if;
end C;
begin for I in 0.. H loop
    SET_COL (W*(H-I)/2);
    for J in 0.. I loop
    PUT (C (I, J), W);
    end loop;
    NEW_LINE;
  end loop;
end PASCAL_TRIANGLE;
```

図-1　パスカル三角形のプログラム (Ada)

```
program pascaltriangle (output);
const h=10; w=4; (*h: total height, w: unit width*)
var i, j: integer;
function c (n,i=integer): integer;
  begin if (i=0) or (i=n) then c:=1
    else c:=c (n-1, i-1)+c (n-1, i) end;
begin for i: =0 to h do
  begin write ( ' ' : w* (h-i) div 2);
    for j: =0 to i do write (c(i, j): w);
    writeln end end.
```

図-2　パスカル三角形のプログラム (Pascal)

Ada では今の with クローズ，use クローズで見たように，予約語 (reserved word) は小文字の太字 (または下線つき)，一般の名前は大文字の細字 (または下線なし) で表わすのが文法書の習慣だが，利用者が宣言する一般の名前には 62 種の予約語と同じ綴りのものは使ってはならず，大文字と対応する小文字は大文字と同一文字として扱うことになっている．なお一般の名前には途中に連続しない下線を入れてよい．「**procedure** PASCAL_TRIANGLE」から「**end** PASCAL_TRIANGL；」までが手続き本体である．もし

† An Overview of Ada by Eiiti WADA (Department of Mathematical Engineering, University of Tokyo).
†† 東京大学工学部計数工学科
* Ada 成立の背景については文献 2) 参照

手続きに仮パラメタ (formal parameter) があれば，手続きの名前と **is** の間に書く．その下の2行が，この手続きに局所的なオブジェクト宣言 (object declaration) で，たとえばHは整数型の定数で値は10という宣言である．オブジェクト宣言で **constant** のあるものが定数，ないものが変数で，変数の場合：＝と式があれば，それは初期値を表わす．このプログラムには変数のオブジェクト宣言はない．－－から行末までは注釈である．

「**function** C (N, I: **in** INTEGER) **return** INTEGER **is**」から6行先の「**end** C；」までが，この手続きに局所的な関数Cの本体 (body) である．Pascal のプログラムではこれに対応するのが関数宣言であるが，Ada ではこれは関数宣言とはいわない．関数宣言をしようと思ったら，宣言が並んでいるうちに（本体より前に），もう一度

function C (N, I: **in** INTEGER) **return** INTEGER；と書いておくと，これが関数宣言になる．Ada では関数にも手続きにも宣言と本体があって，宣言だけまとめてはじめに書き，そのあとに本体を並べる．関数宣言や手続き宣言は Pascal の forward 宣言みたいな機能だが，本体の方にも仮パラメタや結果の型をくり返さなければならないところが Pascal と異なる．Pascal で forward 宣言がいらないような場合には関数や手続き宣言はなくてもよく，その場合は本体が宣言の機能を果す．

関数本体の先頭で，仮パラメタと結果型を示す．仮パラメタの **in** はモード (mode) というもので，データが関数に入るだけ，関数から出るだけ，または出入り両方などを **in**, **out**, **in out** で示すことになっている．**in** の仮パラメタは本体の中では定数である．**in out** だと変数，また **in out** と **out** の実パラメタ (actual parameter) には結果が代入できるから変数の必要があるけれども，関数には **in** モードのパラメタしか使えない．また **in** モードなら **in** は省略できる．

この関数Cには局所的な宣言はない．実行すべき文の列（実行文部）が **begin** につづいてすぐに始まる．この場合は文は if 文がひとつ．if 文は「**end if**；」でしめくくることになっているので，**then** のあとにも **else** のあとにも文の列 (sequence of statements) を書くことができる．with クローズや手続き本体などもそうであったが Ada は多くの構文単位がセミコロンでおわる．文も同様であり，**then** のあとに書く文も今は **return** 1；がひとつ．空文がある Pascal では **else** の前にセミコロンを書くと致命的な構文エラーだが Ada では反対に必要である．

関数の結果の値を返すには Pascal のように関数名に代入するという気持ちの悪いことはせず，Lisp 流に **return** につづけて返す値を書く．値なしの **return**；は手続きの途中から帰るのに使う．**else** のあとのreturn 文に見るように，関数や手続きの再帰的使用はもちろん許される．本体のおわりは **end** と，そのあとにつけたければ関数名をつけ，セミコロンとなる．

手続きの実行内容は局所的な宣言と宣言に付随した本体を呈示したあとの **begin** と **end** 間の文の列で示す．今の場合は loop 文がひとつである．この例では loop 文が入れ子になっているから注意を要するけれど，どちらの loop 文も Pascal の for 文に相当するものになっている．Pascal の while 文も Ada では loop 文の一種である．for 文相当の loop 文は **loop** の前に「**for** 制御変数 **in** 出発値..最終値」を書くのだが，ここでも Pascal とちがってループパラメタ (loop parameter) とよぶ制御変数は **for** のところで自動的に宣言したことになる．したがって図-1には，図-2 のような *i*, *j* の変数宣言は書いてない．ループパラメタを，出発値から最終値まで順々にふやしながら **loop** と「**end loop**；」間をくりかえし実行するけれども，その間ではパラメタは定数ということになっている．またループの終りでループパラメタの有効範囲は消失することになっている．

そういう loop 文でIを0からHまで変えつつパスカル三角形を印刷するプログラムなのだが，まず各行の左端から W*(H－I)/2 だけの空白をおき，そのあとに二項係数を並べようとする．図-2 のプログラムでは *write* の手続きでスペースを必要な幅だけ出力しているが，図-1 の方は，TEXT_IO の手続き SET_COL で印刷位置をセットする．そのあと内部の loop 文でJを0からIまで変えつつ C(I, J) を計算し，幅Wで出力する．それにはやはり TEXT_IO の手続き PUT を使う．Jによるループが終ったあとは NEW_LINE；で改行する．C(I, J) は関数Cが実パラメタを必要としたからカッコ内に実パラメタを書いたのだが，関数にもともと仮パラメタがないか，あってもデフォールト (default) な値があって，実パラメタを必要としない場合でも，関数の呼出しには () だけは書かなければならない．だが手続きの場合には実パラメタが不要なら NEW_LINE；の例で見たよ

うに，（）は書かなくてよい．NEW_LINE；でIによるループのくりかえし部分がおわり，「**end loop**；」でしめくくる．さき程の **begin** に対応する **end** と，念のためにつけた PASCAL_TRIANGLE とセミコロンで手続き本体，したがってコンパイル単位がおわる．

Ada のプログラムはこの例から見る限り，Pascal のそれとあまりちがわないけれども，やはり設計年代に 10 年のへだたりがあるだけのことはあって，いろいろな新機軸が用意してある．以下にそれらの主なものを説明しよう．

2. 式 と 値

Ada は型あり言語 (typed language) である．そこで式と，その式を評価して得られる値には型がある．式の構成要素は定数，変数，関数呼出し，演算子による演算など，従来のプログラム言語にみられたもののほか，値の構造体ともいうべきアグリゲート (aggregate)，型の性質（配列の上限や下限など）を調べたりするアトリビュート (attribute)，ヒープ (heap) 領域にある型の場所をとって，そこへのポインタを返すアロケータ (allocator) など，面白いものができた．

型の考え方は，ひと口でいえば Pascal のそれのようではあるが，Pascal で分かりにくい部分範囲型や，上下限を固定しなければならなかった配列型などは，考え方を整理しようとした跡がみられる．すなわち型を基底型 (basetype) と部分型 (subtype) の 2 種類の概念に分けた．基底型は演算の対象としての分類，部分型はとりうる値としての分類と思ってよい．たとえば INTEGER 型のデータは整数演算の対象となりうるからこれは基底型である．INTEGER（−128..＋127）型のデータは整数演算の対象となりうるけれども，とりうる値は −128 から ＋127 までに制限される制限つきの部分型である．もちろん INTEGER 型だって無制限に大きい値がとれるわけではなく INTEGER'FIRST から INTEGER'LAST の範囲の値しかとれないから，基底型であると同時に制限なしの部分型である．これに対して INTEGER（−128..＋127）は制限つきの部分型である．Pascal では部分範囲型を整数型と同じように一人前の型としたから，整数型の演算は整数型の部分範囲型のデータにも使えるだろうかという疑問を起こさせたりもしたが，その点は解決した．

一方配列の方はどうしたかというと，添字の型に制限なしの型をもつような配列を，制限なしの配列型として認める．しかしそんな無制限な場所はとれないから，実際に使うときは添字の制限を何らかの型でつけるということにした．つまり

type MAT **is array** (INTEGER **range** ＜＞,
INTEGER **range** ＜＞) **of** INTEGER;

という制限なしの二次元配列型 MAT が宣言でき（**array** とそれからうしろを配列の型定義という），

BOARD : MAT (0..7, 0..7);

のように型名に添字の制限 (index constraint) をつけたオブジェクト宣言をして，二次元配列 MAT の部分型としての変数 BOARD をとることができるようになった．

Ada の型の考え方には上にのべた基底型と部分型のほかに導出型 (derived type) がある．型あり言語では型が同じとはどういうことかという根本的な問題がある．Ada では型定義を書くと，それはどんなに前に書いた型定義とそっくりでも，厳然と別の型になる．したがって前の型を仮パラメタにもっている手続きや関数を新しい型に流用して使おうとしても，それはだめである．プログラム作成技術上別の型にはしたいけれど，すでに宣言した手続きや関数はぜひ共用したいというときには，はじめの型を親型 (parent type) とする導出型を宣言する．そうすると親型とは相互に代入は不可能だが演算，手続き，関数などが共用できるもうひとつの型が出現する．たとえば

type MAT 1 **is new** MAT;

BOARD1 : MAT1 (0..7, 0..7);

と書くと MAT1 は MAT からの導出型となり，BOARD 1 は MAT1 の部分型であり，BOARD とは型がちがうので BOARD1 に BOARD を代入することはできない（代入は演算とはいわない！）

型についての基本的な考え方は以上の通りだが，型を分類すると次のようになる．

- 型
  - スカラ型 (scalar)
    - 離散型 (discrete)
      - 数えあげ型 (enumeration)
      - 整数型 (integer)
    - 実数型 (real)
  - 合成型 (composite)
    - 配列型 (array)
    - レコード型 (record)
  - アクセス型 (access)
  - プライベート型 (private)
  - タスク型 (task)

数えあげ型は Pascal のそれとほとんど同じで，BOOLEAN 型も CHARACTER 型も標準の数えあげ型である．Pascal とくらべて異るのは，数えあげ型の型定義で，とり得る値として並べる定数 (enumeration literal) が，名前のほか，文字型定数 (character literal) でもよいこと，またほかの数えあげ型の定数名を使ってもよいこと（したがってこの定数名はふたつの型に属することになる．このようにひとつの名前が同時に複数の意味をもつことを Ada ではオーバロード (overload) という．）などである．

```
type SYMBOL is (ID, NUM, ADD, SUB, MULT,
  DIV, L_PAR, R_PAR);
type OP_CLASS is (ADD, MULT);
```

のように ADD を SYMBOL と OP_CLASS の両方の数えあげ型の定数名として使うことができる．Pascal でコンパイラを書こうとすると，シンボル名としても，加減演算子のようなオペレータの組の名としても *add* が使いたくて困るようなことがあったが，それは Ada では解決した．

次は整数型．Ada の整数型は恐らく初心者にとって驚きのひとつとなるであろう．文法書には整数の型定義として範囲の制限 (range constraint) **range** 1..10 のような形が示してある．そこで

```
type INDEX is range 1..10;
```

のような型宣言は当然書きたくなる．一方範囲があまりよくわからないものは，オブジェクト宣言で

```
I: INTEGER;
```

とし，範囲のわかっているものは，

```
J: INDEX;
```

とする．変数 I の値が充分 INDEX の範囲内にあるからと思って，J に I を代入しようとするとこれがだめなのである．反対に I に J を入れるのもだめ．なぜかというと，整数の型定義をしたのにも拘らず，INDEX は INTEGER の導出型であって，INTEGER とは型が違うのである．文法書には明確にそう規定してあるが，これはきっと多くのプログラマが間違えるにちがいない．INDEX を上のようなつもりで使うには

```
subtype INDEX is INTEGER range 1..10;
```

としなければならない．

整数の定数，1234 のようなものは INTEGER とか INDEX のような型名をもった型に属するのではなくユニバーサル整数 (universal integer) という特別な型で，どの整数型とも一緒に使うことができる．

実数についても事情はほぼ同様である．ただ実数には，相対精度に基づいた浮動小数点型 (floating point type) のものと絶対精度に基づいた固定小数点型 (fixed point type) のものの 2 種類がある．それらは精度の制限 (accuracy constraint) を，**digits** 式で書くか **delta** 式 で書くかで指定する．これまでのプログラム言語では，実数の精度については大体が，"in the sense of numerical analysis" の程度のことを規定して，詳細は処理系にまかせていたけれども，Ada では各型各精度の実数値のとり得る値を，2 進法の計算機の値に照らして正確に記述した．

Pascal で構造型とよぶものは Ada は合成型であるが，配列とレコードしかない．

配列についてははじめの方にのべたので，ここでは ⅰ）配列型の代入は，要素の添字を平行移動しても，対応する要素があるなら許される，ⅱ）添字は丸い( ) の中に書くということだけをのべて先にすすもう．

レコード型はずい分面白いことになった．

ⅰ）各要素にはデフォールトの初期値を書いてよい．したがって，この型のデータを作ったときに，積極的に初期値を入れなければ，自動的にデフォールトの値になる．たとえば二進木のセルは

```
record L_LINK, R_LINK: LINK: =null;
       DATA: INTEGER: =0;
end record;
```

のようなレコード型の定義が書ける．

ⅱ）Pascal のレコードのタグフィールドのように，レコードの構造をきめる要素は，ディスクリミナント (discriminant) といって，まとめて前におき，型宣言で宣言する型名の仮パラメタのような働きをする．たとえば，加減乗除の OPERATOR か文字列の変数名からなる算術式のノードを表わす型は，

```
type MODE is (OP, VAR);
type NODE (M: MODE);
type LINK is access NODE;
type NODE (M: MODE) is
record case M is
  when OP=>
     L_LINK, R_LINK: LINK;
     OPR: OPERATOR;
  when VAR=>
     VAR: STRING;
  end case;
end record;
```

と書く．ここで NODE の型宣言のところにあるM：MODE がディスクリミナントであり，NODE 型のデータをとるときに，M に OP か VAR かを与えて，どちらの可変部をもつレコードにするかをきめる．

ディスクリミナントにまだいろいろな値のとりうる余地の残っているレコード型は，制限なしの基底型であり，これに対して，ディスクリミナントの制限 (discriminant constraint) をつけたものは，部分型ということになっている．ディスクリミナントは Pascal のタグフィールドのように，レコード内でひとつの場所を占めているから，値を代入できるわけだが，うしろの構造に関係する重要な位置なので，値の変更には制限がある．オブジェクト宣言でとったレコード変数では，レコード全体を一斉にとりかえるなら，ディスクリミナントは変更してよい．しかし後述するアクセス型変数の先にとったレコードでは，たとえ一斉にでもディスクリミナントは変更できない．

アクセス型の例はすでにディスクミナントの説明の LINK 型として現れた．大体は Pascal のポインタ型と同じと思って差しつかえないが，多少注意を要するのは次の点である．

ⅰ）レコード型とアクセス型は，相互参照になることが多く，Pascal ではポインタ型を先に定義すれば，ポインタの被参照型は未定義で参照してもよいという特例があったけれども，Ada では，不完全な型宣言 (incomplete type declaration) で名前だけ導入しておく．前掲ディスクリミナントの例に NODE の宣言が2回現れるが，はじめのものが不完全な型宣言で，アクセス型 LINK の露払いになっている．LINK の宣言がすむと NODE の本物の宣言がくる．

ⅱ）アクセス型のデータの指す動的変数をヒープ領域にとるには，Pascal ではポインタ型の変数を実パラメタにした標準手続き *new* を使ったが，Ada では予約語 **new** とそれにつづく動的変数の型名，初期値を書いたアロケータによる．アロケータはヒープ領域にその型名の場所をとり，そこへのアクセス型の値を返す．同じ被参照型へアクセスしている複数のアクセス型があり得るから，アロケータの返す値の型は，アロケータの使われる文脈によって確定する．

ⅲ）被参照型変数を表わすのに，Pascal ではポインタ型変数に↑をつけて示したが，Ada ではアクセス型変数はそのうしろに被参照変数の配列の添字やレコードの選択形を書くと，レファレンス剝がしが自動的に行われる．↑に相当する記号はない．そのかわり被参照変数全体を示すには **.all** の選択形とする．ヒープをとる領域やゴミ集めその他の細部は省略する．

プライベート型については静的なスコープの節で，タスス型については動的なプロセスの節でのべる．

型はほとんどの場合，型宣言をして型名にしてから使う．型宣言，オブジェクト宣言，手続き，関数宣言には Pascal にあったような順に関する制限はない．宣言は順々にエラボレート (elaborate) される．（エラボレートに対するうまい訳はまだないが，以下では確認と書いたところがある．）

さて，これまで説明した型のデータを一次子 (primary) として，式を構成するのだが，式は関係式 (relation)，また関係式を同種の論理演算子でつなげたものの形をとる．つまり，一旦 **and** でつなぎだしたら，**and** 以外は使えない．**and** 以外の論理演算子には **or, xor, and then, or else** があり，あとのふたつは左から式の真偽が確定したら，以後の関係式は評価しないという短絡制御 (short circuit control) を表わす．関係式は単純式 (simple expression)，または単純式間の大小等不等関係，単純式の範囲や型のチェックの形をとる．単純式は項 (term) と加減演算子の列（最前部には単項演算子が許される），項は因子 (factor) と乗除演算子の列，因子は一次子，またはその一次子によるべき乗の形をとる．

比較演算子は＝，/＝，＜，＜＝，＞，＜＝
加減演算子は＋，－，&
単項演算子は＋，－，**not**
乗除演算子は＊，/，**mod**, **rem**
べき乗演算子は＊＊．

これらの演算子については，優先順位を変更することはできないが，一部のものについては，その機能を既存の型について，または別の型について，関数宣言の形で再定義することができる．たとえば BOOLEAN 型に対して＋を論理和にしたければ

```
function "+" (X, Y : BOOLEAN) return
                    BOOLEAN is
begin return X or Y; end;
```

とすればよい．

加減乗除は整数と浮動小数点に対しては，同じ型の被演算子について規定され，結果も同じ型，固定小数点を含む演算については，ここでは省略する．整数の除算の剰余には **mod** と **rem** の2種類がある．**mod** は剰余の符号を除数に合わせるもの，**rem** は被除数に合わせるものである．負の除数で割り切れた場合，

mod の結果は0としている.

アグリゲートは合成型のすべての要素に対応するデータの集まりである. 要素4個の整数型一次元配列のアグリゲートは

(3, 2, 9, 8)

(3, 2, 4=>8, 3=>9)

(1..4=>0)

などのように, また前掲の二進木のセルに対しては

(**null**, **null**, 15)

(L_LINK|R_LINK=> **null**, **others**=>15)

などのように書くことができる.

3. 文と効果

タスク関係を除けば Ada の文は簡単である. 文にはいくつでも goto 文でとぶためのラベルをつけることができる. ラベルは<<L1>>のように2重の大小記号で囲む. goto 文の方は「**goto** L1;」のように書く. ついでだが, goto 文でとべるのは goto 文と並んでいる文か, goto 文を含む文と並んでいる文かの程度である.

さて代入文については:=の左辺の変数に右辺の式を代入するもので, 代入文の構文は最後のセミコロンまでである. 両辺の型は同じでなければならず, 変数についている制限に合っていなければならない. 配列は対応する要素ごとに代入される.

if 文については 図-1 の例でのべた. **else** の前に「**elsif** 条件**then** 文の列」をいくつも置くことができる. 条件とは BOOLEAN 型の式のことである.

case 文もある. これは

```
case CH is
when 'A'..'Z'=> READ_ID;
when '0'..'9'|'+'|'-'=> READ_NUM;
when others=> SPECIAL_CH;
end case;
```

のように書く. =>のあとは文の列でよい. **when** のあとの場合わけは, すべての場合を尽していなければならない.

繰り返しは loop 文に統一された. for 文ともいうべきものは 図-1 に示した如くである. 「**while** 条件 **loop** 文の列 **end loop**;」という while 文のようなものも書けるし, **loop** だけから始まる loop 文もある. **loop** で始まる loop 文は無限ループだが, 中の exit 文, **return** 文, あるいは **terminate**; で脱出できる. loop 文にはループ名(「名前:」の形)をつけてよい. exit 文 **exit**; は一番中のループからとび出すが「**exit** ループ名;」だとその名前のループまで一度にとび出す. ループ名はループパラメタの前にきて選択形を作ることができる. つまり 図-1 の入れ子のループで, 外のループに OUTER: と名をつけると, Iのかわりに OUTER.I と書いてよく, そうすれば内のループパラメタにふたたびIを使うこともできる.

Pascal では文と同じ資格のブロックが書けないという批判がでたけれども, その点 Ada は Algol 60 の伝統を継承した. すなわち「**declare** 宣言の列 **begin** 文の列 **end**;」という形のブロック (block) が使える. これにはまたブロック名(「名前:」の形)を前づけでき, 宣言した名前と選択形をつくるのに用いる. Algol 60 の伝統によればブロックこそプログラム本体になるのだが, Ada のブロックにはその資格はない. 手続きが主プログラムになる.

return 文は手続きからは「**return**;」で(または最後の文の終了で)帰るのに, また関数からは「**return** 式;」で帰るのに使う. この式の値は, 関数が返すはずの値の制限のチェックをうける.

空文は **null**; である.

手続きの呼出し (call) は 図-1 の PUT や NEW_LINE に見た如く, 手続きの名前にカッコ内の実パラメタを書きセミコロンをおく. 実パラメタと仮パラメタの対応は, 位置によってつけても, 「仮パラメタ名=>実パラメタ」のように名前によってつけてもよいが, 両方式の混在も, 一旦名前の対応が現れたら以後位置による対応はつけないという条件の下に許されている. デフォールトな値をもつ **in** モードの仮パラメタには実パラメタはなくてもよい. 手続き名もオーバロードしてよいから, どの手続きを呼出すかは, 実パラメタの型や, 名前の対応から解決しなければならないことがある.

タスクにエントリ (entry) の呼出しというのがあるが, 大体において手続きの呼出しと同じと思ってよい(後述).

4. 静的なスコープ

Ada における名前のスコープ (scope), つまり有効範囲の規則は, 基本的には静的なそれである. すなわちプログラムの字面の上での位置関係できめるものである. ただ, パッケージとか, 分割コンパイル (sepa-

```
package LISP_SYSTEM is
  type SEXPR is private;
  T: constant SEXPR;
  NIL: constant SEXPR;
  function CAR (X: SEXPR) return SEXPR;
  function CDR (X: SEXPR) return SEXPR;
  function CONS (X,Y: SEXPR) return SEXPR;
  function ATOM (X: SEXPR) return SEXPR;
  function EQ (X, Y: SEXPR) return SEXPR;
private
  type CELL (ISATOM: BOOLEAN);
  type SEXPR is access CELL;
  type CELL (ISATOM: BOOLEAN) is
    record case ISATOM is
      when TRUE=> PNAME: STRING;
      when FALSE=> CAR, CDR: SEXPR;
      end case;
    end record;
  T: constant SEXPR: =new CELL (TRUE, "T");
  NIL: constant SEXPR=new CELL (TRUE, "NIL");
end LISP_SYSTEM;
```

パッケージ宣言

```
package body LISP_SYSTEM is
  function CAR (X: SEXPR) return SEXPR is
    begin if not X.ISATOM then return X.CAR;
          end if;
    end CAR;
  function CDR (X: SEXPR) return SEXPR is
    begin if not X.ISATOM then return X.CDR;
          end if;
          end CDR;
  function CONS (X, Y: SEXPR) return SEXPR is
    begin return new CELL (FALSE, X, Y);
    end CONS;
  function ATOM (X: SEXPR) return SEXPR is
    begin if ISATOM then return T;
      else return NIL;
      end if;
    end ATOM;
  function EQ (X, Y: SEXPR) return SEXPR is
    begin if X=Y then return T;
      else return NIL;
      end if;
    end EQ;
end LISP_SYSTEM;
```

図-3 パッケージ本体

rate compilation) とかがあるので多少厄介である．またスコープとは何とかすれば名前が見えるかも知れない部分であり，そのほかに Ada ではスコープの一部分にその名前が直接見える部分 (directly visible) を定義する．

まずパッケージから説明しよう．パッケージは近年はやりの抽象的データ型を実現する手段として導入したといってよいであろう．つまりある性格のデータを表わすための型と，それに付随した操作をまとめて定義し，利用者はその型のデータを宣言したり，与えられた操作をほどこしたりすることができる．しかしその型が実際にどういう形で実現されているかはわからない．この目的のためにパッケージには利用者から見える部分 (visible part) と見えない部分 (private part) がある．見えない部分のうち，外にみせている型の実際の定義や，その型に関連したオブジェクトの実際の宣言は，見える部分と一緒になってパッケージ宣言 (package declaration) となる．また外に見せている手続きや関数の本体と，オブジェクトの初期化のためのプログラムなどは，パッケージの本体 (package body) に書く，**図-3** はパッケージの例のとして書いた LISP_SYSTEM である．

このうち上半分がパッケージ宣言，下半分がパッケージ本体である．宣言のうち，中ほどの **private** から上が外から見える部分，下が見えない部分であり，本体の中も外からは見えない．だから利用者には Lisp のデータがアクセス型なのか，配列による実現なのか全くわからない．わかるのは SEXPR という型，T と NIL の定数，CAR, CDR などの Lisp の基本関数である．SEXPR の型定義が **private** になっていることからわかるように，この型をプライベート型という．プライベート型にはさらに制限つきプライベート型 (limited private) もあるが説明は省略する．実際の宣言や本体がどこにあるかは例をみてほしい．

ところで SEXPR 型変数 FOO を宣言し，T と NIL を CONS したものを初期値として代入しようとしたらどうするか．とにかくパッケージの宣言の前では何もできない．パッケージ宣言のあとで，名前 LISP_SYSTEM が見えている部分で

```
FOO : LISP_SYSTEM . SEXPR
  :=LISP_SYSTEM . CONS(LISP_SYSTEM . T,
                       LISP_SYSTEM . NIL);
```

と書く，なぜかというと，LISP_SYSTEM の見える部分では，(あるいは LISP_SYSTEM のスコープのある部分では，) SEXPR などパッケージの外に見せる部分で宣言した名前は，スコープをもつので，何とかすれば使えるからである．しかしそれらの名前はパッケージ宣言と本体の中でしか直接には見えない．だからじかに SEXPR とは書けない．そこで見える名前の LISP_SYSTEM を使って選択形にした．このように Ada の文法書には，どこで宣言した名前についてはスコープはどこ，直接見える場所はどこと細かい規定がある．選択形はあまりにもわずらわしいので，別

の方法がふたつ用意してある．そのひとつは冒頭にのべた use クローズで，ある宣言の部分に「**use** LISP SYSTEM；」と書くと，SEXPR などは，この宣言の部分で宣言したかのように直接見えてくるのである．もっとも use クローズは万能でなく，見えるようにすることによって名前の区別ができなくなるような場合には，見えるようにはならない．もうひとつはリネームで，**subtype** を使ったり，**renames** を使ったりして新しい名前を用意する．たとえば

```
subtype SEXPR is LISP_SYSTEM.SEXPR;
T : SEXPR renames LISP_SYSTEM.T;
CAR (X : SEXPR) return SEXPRrenames
                    LISP_SYSTEM.CAR;
```

のようにする．

5. 動的なプロセス

Ada ではタスクを用いて並行処理が記述できる．タスクはタスク型として宣言し，その型のオブジェクトを宣言したりヒープにとったりしていくつでも作ることができるが，オブジェクトがひとつのときはわざわざ型宣言しないでよい．**図-4** は型宣言をしない例である．タスクはタスク宣言とタスク本体とからなる．タスク間の通信はエントリの呼出しとそのアクセプト (accept) で行われるが，タスク宣言にはそのタスクの受け持つエントリを宣言する．普通は手続き宣言のような形をとる．タスク本体は，このタスク型のオブジェクトが作られたときに起動される文の列を定義するためのものだが，その実行に必要な局所的な宣言をしてよい．この文の列の中には，宣言のところに並べたエントリに対する accept 文を含ませる．accept 文はエントリの呼出しを手続き呼出しとみたときの手続本体の役割りをはたす．エントリの呼出しと accept 文の実行は同期をとって行われる．これをランデブ (rendezvous) という．つまりエントリ呼出し，accept 文のいずれかに先に到達したタスクは相手のタスクの到着を待つ．accept 文実行中はエントリ呼出しのタスクは停止し，accept 文が終了すると，それぞれのタスクの以後の文は独立に実行される．データはパラメタの形で渡す．

これだけの機能では，無駄に待つことが多く，自由な制御ができないので，呼ぶ側にも呼ばれる側にも相手側の状態によって行動をかえる機能がある．呼ぶ側のは条件つきエントリ呼出し (conditional entry call) で

```
task BUFFER_PROCESS is
entry ENTER_QUEUE (X : SEXPR);
end BUFFER_PROCESS;
                タスク宣言
task body BUFFER_PROCESS is
  Q,S : SEXPR : =NIL;
  task PRINT_PROCESS is
  entry START_PRINT (X : SEXPR);
  end PRINT_PROCESS;
  task body PRINT_PROCESS is
  R : SEXPR;
  begin
    loop
      select accept START_PRINT (X : SEXPR) do
          R : =X; end START_PRINT;
          PRINT (R);
      or terminate;
    end select;
  end loop;
  end PRINT_PROCESS;
begin
  loop accept
    select ENTER_QUEUE (X : SEXPR) do
      S:=COPY (X) end ENTER_QUEUE;
      if Q=NIL then
        select START_PRINT (S);
        else Q : =CONS (S, NIL);
        end select
      else Q:=NCONC (Q, S);
      end if;
    else if Q /= NIL then
        select START_PRINT (CAR (Q));
          Q:=CDR (Q);
        else null;
      end select;
    end if;
      end select;
  end loop;
end BOFFER_PROCESS;
```

図-4 タスク本体

```
select エントリ呼出し 文の列
else 文の列
end select ;
```

の形で，アクセプトの用意ができていればエントリ呼出しとその後の文の列へ，そうでなければ **else** につづく文の列へ進む．

呼ばれる側のは選択待ち (selective wait) で

```
select accept 文 文の列
or accept 文 文の列
.....
else 文の列
end select;
```

の形で，エントリ呼出しで待っているものがあるとそれに対応する accept 文とその後の文の列，ひとつもなければ **else** につづく文の列へ進む．正確な構文にはさらにいろいろ付着するが，それらは省略した．

図-4 を説明する．これは Lisp の S式の PRINT をバッファリングするものである．必要な名前は見えるものとする．タスク宣言と本体で，BUFFER_PROCESS というタスクオブジェクトができるが，これはただちに局所的な宣言を確認して，実行文部の実行を開始する．この例は宣言の中で PRINT_PROCESS というタスクも宣言しているので，そのオブジェクトができ，こちらの実行文部の実行の方が先に開始される．この実行文部は loop 文であり，その構成要素は選択待ちである．今 **terminate**; は説明が厄介だからないことにすると，accept 文しかないから，DELETE_QUEUE のエントリまちになる．

外側のタスクの実行文も loop であり，その中は選択待ちになっているが，ENTER_QUEUE のエントリが呼ばれなければ，**else** の方へゆく．この方の if 文はQが NIL なので結局なにもしないで，ループをくりかえしている（ここには工夫の余地がある）．

ここには示してない別のタスクは，PRINTしたいS式を実パラメタにもって，ENTER_QUEUE のエントリを呼び出す．すると ENTER_QUEUE の accept 文の実行に進む．同期するのはXのコピーをSに代入するところまでである．これがすむとこのタスクは呼んだタスクとはなれ以下の文の実行にゆく．すなわちバッファを構成しているQが空ならすぐに条件つきエントリ呼出しを試る．つまり DELETE_QUEUE が待ち構えていれば，それを呼びだすし，そうでなければ，Qにつないでおく．Qが NIL でないときは DELETE_QUEUE は忙しいであろうから，とりあえずQの最後に NCONC でつないでおく．

ENTER_QUEUE が呼ばれないで BUFFER_PROCESS タスクがひまなら，たえずQが NIL でないかぎり，条件つきエントリ呼出しで，すぐに呼べるかどうか調べ，呼べるならQの先頭を渡して PRINT してもらう．渡せた場合はQを CDR(Q) でおきかえるが，これは下請けのタスクとは独立におこなう．もしエントリがすぐに呼べなければ，空文 **null**; を実行する．

PRINT_PROCESS の実行文部は accept 文で待っていたが，BUFFER_PROCESS からエントリ呼出しをうけると，同期して実行中に，Xとして受けとったデータを局所的変数 R にうつす，そしてゆっくりと PRINT の手続きを使う．(PRINT は Lisp とちがって手続きにしてある)．

タスク間の制御と多少異なるが，Ada の例外処理 (exception) について簡単に述べる．ここまでの説明では一切省略してきたが，Ada では変数への代入，パラメタの引き渡しなどのとき制限に合っていないと CONSTRAINT_ERROR の割出しが起きる．割出しには文法できめた標準のものと利用者が例外宣言で宣言するものとがある．標準のものは上述のもの以外に NUMERIC_ERROR, SELECT_ERROR, STORAGE_ERROR, TASKING_ERROR がある．これらの割出しは文法できめた条件により起きるが，利用者の宣言した割出し raise は文で起こす．

一方，手続き，関数，パッケージ，タスク本体の最後や，ブロックの最後には例外処理部 (exception handler) を

exception と **end**; の間に
when 例外名=>文の列

の形を書くことで用意することができる．ある実行文部で例外の割出しがおきたとき，その最後にその名前の例外処理があると，制御はそちらへ移る．そして例外名につづく文の列を実行して，はじめの実行文部が終ったことになる．もしその例外名の例外処理がなければ，実行文部の外へ割出しを伝播する．割出しは宣言の確認中にも起きうるが，それはこの宣言のスコープにある例外処理でなく，一段外の例外処理につかまることになっている．

このほか generic, representation specification など，説明したいことはいろいろあるが，省略する．

参 考 文 献

1) United States Department of Defense, Reference Manual for the Ada Programming Language, Proposed Standard Document (July 1980). この複製版「プログラム言語 Ada 基準文法書」は 12 月に共立出版から出た．
2) 筧 捷彦：Ada——米国国防総省新言語，情報処理，Vol. 21 No. 9, pp. 975-979 (Sep. 1980).

(昭和55年11月20日受付)