第2編

プログラミングパラダイム

「友らよ，君たちはいうのだな，趣味と味覚は論争の外にあると。
しかし生の一切は，趣味と味覚をめぐる争いなのだ。」

ニーチェ
「ツァラトゥストラ」
（手塚富雄訳）

第５章

構造化プログラミング

5.1 プログラムの構造

プログラムの構造は，通常，その記述に用いるプログラミング言語によって規定される。従来の手続き型言語がそのプログラム構造を規定する要素の主なものは，

①モジュール構造
②制御構造
③データ構造

である。

(1) モジュール構造

「モジュール」という言葉は一般的すぎて多義的であるが，ここではプログラムの分割の単位と考えておく。狭い意味ではコンパイルの最小単位，広い意味では機能的に意味のある単位である。具体的には，以下のようなものである。

●サブルーチン（FORTRAN）
●関数（FORTRAN，Pascal，C）
●手続き（COBOL，Algol，PL/I，Pascal）

一般には，このようなモジュールの個々の機能およびモジュールの間の呼び出す側と呼び出される側の関係によって，プログラムの全体の構造のわかりや

すさが影響される。たとえば，FORTRAN のようにモジュール間の共通変数があるものは，各々のモジュールの機能は，そのモジュールのソースコードを見ただけでは理解するのが難しい。また，モジュールを呼びだすときの実引数と仮引数の対応のとり方に自由度の大きいものはインタフェースの誤りを生じやすい。Algol や Pascal のように，ブロック構造形式のものは，変数などの名前の有効範囲が明確になる利点がある反面，ブロックの入れ子構造が深くなると誤解も生じやすい。したがって，モジュール構造およびその関連の共通変数，引数受け渡し機構，ブロック構造の有無などがプログラムのわかりやすさに重要な役割を持っている。

(2) 制御構造

手続き型言語では，プログラムの実行順序を規定する制御構造を以下のような制御文で表現する。

- ●分岐処理（goto 文，exit 文，return 文など）
- ●選択処理（if 文，case 文など）
- ●反復処理（do 文，while 文，until 文など）
- ●手続き呼び出し（call 文など）

各モジュール単位のプログラムの機能を理解するには，この制御文を頼りに実行順序を追っていく必要があるが，その時，goto 文のような無条件分岐が多用されていると，制御構造の理解は容易ではない。

(3) データ構造

手続き型言語では，プログラムで扱うデータ構造はデータ型によって規定される。このデータ型は以下の括弧内のような宣言文によって記述される。

- ●数値（integer，real など）
- ●文字列（character など）
- ●ビット列（bit など）
- ●真理値（boolean など）
- ●ポインタ（＊など）

●配列（dimension，array など）
●構造体（structure，record など）

プログラムの処理対象となる実際のデータがこれらのデータ型によって的確に表現でき，その処理内容も適切な実行文を用いて記述できたときはプログラムはわかりやすい。しかしながら，一般には手続き型言語ではデータ型は基本的なものだけを提供しているので，データの意味を理解するのが難しいような使い方も多い。異なるデータ型の変数の間のデータ値の代入時に型変換を行う機能や異なるデータ型の変数の記憶領域を重ねあわせてよいものなどは，プログラムの記述に便利な反面，処理内容をわかりにくくしている。

5.2　プログラムの理解容易性

1970年頃から，ソフトウェアの大規模化に対応するため，その切り札として構造化プログラミング技術が注目され始めた。従来の作りやすさ優先からわかりやすさ優先へのパラダイム転換が行われた。

(1)　良構造プログラム

★ *構造のきれいなプログラムは，わかりやすい。*

プログラムの良し悪しを決める評価基準は，ハードウェアが高価であった1970年頃までは，性能が良いとか，少ないメモリで稼動することであった。しかし，プログラムの規模が大きくなり，モジュール間のインタフェースが複雑になってくると，高信頼性の確保が難しくなるため，できるだけ誤りを作り込まないことが重要である。また，このような開発コストの高い大規模ソフトウェアは，稼動時に少し機能的に不十分になったからといって，簡単に作り直すことができないため，機能変更や機能拡張を容易に行えることが重要である。このような信頼性や保守性向上のためには，プログラムが「良構造」（well-structured）であることが最も基本的要件である。

これは作りやすさからわかりやすさへのパラダイム転換であり，プログラミング言語の評価基準は書きやすさよりも読みやすさ，すなわち記述容易性 (writability) よりも理解容易性 (readability, understandability) が重要になる。

記述容易性とは，プログラム化しようとしている処理手順がいかに簡単に記述できるかという指標であり，先に述べた手続き型言語の主目的はここにあったといえる。アセンブリ言語よりもより少ない行数で記述できること，言いかえれば高級言語で記述したプログラムをコンパイラが機械語プログラムに変換したときの展開率の大きさが重要視されたという意味で，従来の高級言語は「量的高級化」，すなわち，作りやすさの追求であった。

これに対し，理解容易性とは，記述されたプログラムの処理内容がいかに簡単に読み取れるかという指標である。読みやすさの重要性は以下の理由からも当然であろう。

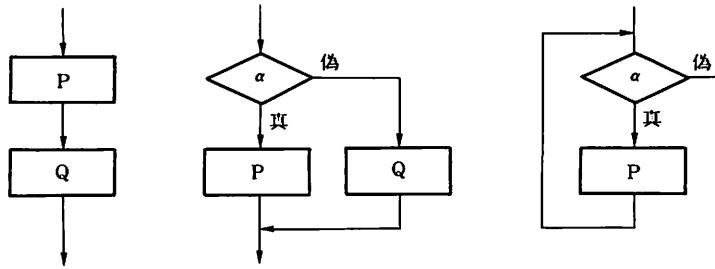
- ●開発時に，プログラムを書くのは一度でも，読む回数は多い。
- ●開発後に，プログラムの改造を開発者と別の人が行うことが多い。

このように，従来の量的高級化に代わり，「質的高級化」，すなわち，わかりやすさを追求した手続き型パラダイムの言語を本書では構造化プログラミング言語と呼ぶ。

(2) 構造化コーディング

★ *goto 文のないプログラムは，わかりやすい。*

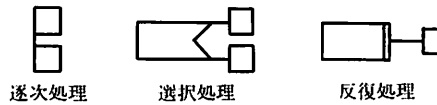
従来の手続き型言語は，プログラムの実行の流れを制御する機能を提供しているが，その1つである goto 文のような無条件分岐を多用するとプログラムの制御構造が複雑になり，理解が難しくなる。そこで，goto 文を使用せず，プログラムの制御構造は，図 5.1(a) に示すような逐次処理，選択処理，反復処理の3種類の組み合わせで表現する方式を E. W. Dijkstra が 1968 年に提案した。このように制御文に限定した構造化技法を構造化コーディングと呼ぶ。



(a)　逐次処理　　(b)　選択処理　　(c)　反復処理

(a)　従来のフローチャートによる記法

(b)　NSチャートによる記法



(c)　PADによる記法

図5.1　構造化コーディングのための制御構造の基本形

この方式の是非をめぐって「goto論争」が行われたが，基本的考え方は定着している。ただし，D. Knuthによってgoto文を用いた構造化プログラムが提示され，例外処理などの限定された使用の必要性は認められている。なお，すべてのプログラムはgoto文なしで記述できることは，C. BohmとG. Jacopiniによって理論的に証明されている。

このような構造化コーディング技法は，従来からプログラムの制御構造をわかりやすく図式表現するために用いられてきたフローチャートについても適用された。従来は図5.1(a)に示すような表記法がとられていたが，この方式では，**図5.2**に示すように，goto文に相当するあるボックスから他のボックスへの矢印が自由に書けてしまう。そこで，このようなgoto文に相当する処理が記述できないような構造化図式がいくつか提案され，使用されている。その表記法の例を図5.1の(b)と(c)に示す。(b)はNS (Nassi-Shneiderman) チャートと呼ばれ，**図5.3**のように使用される。(c)は，二村らによって提案されたPAD

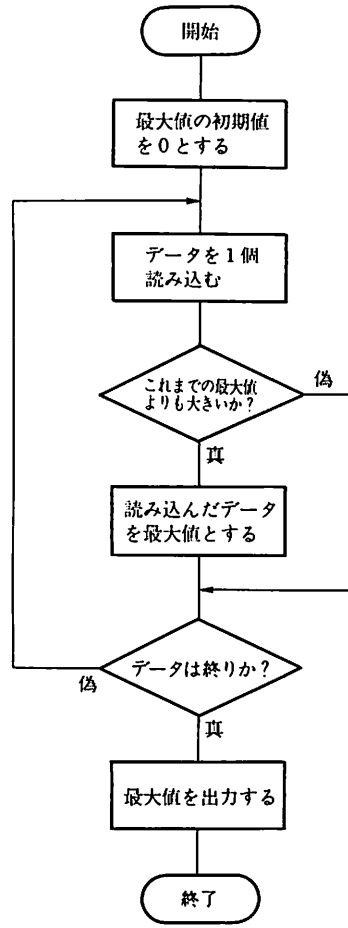


図5.2 フローチャートの例（最大値を求めるプログラム）

(Problem Analysis Diagram) と呼ばれる記法で，図5.4がその使用例である。

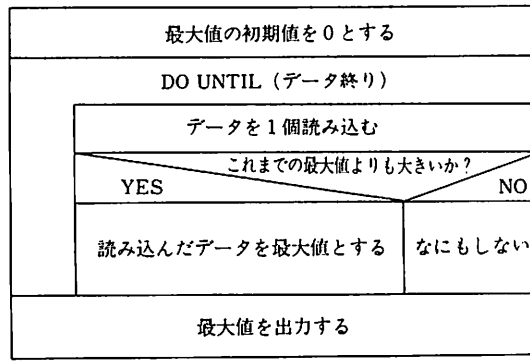


図5.3　NSチャートの例（最大値を求めるプログラム）

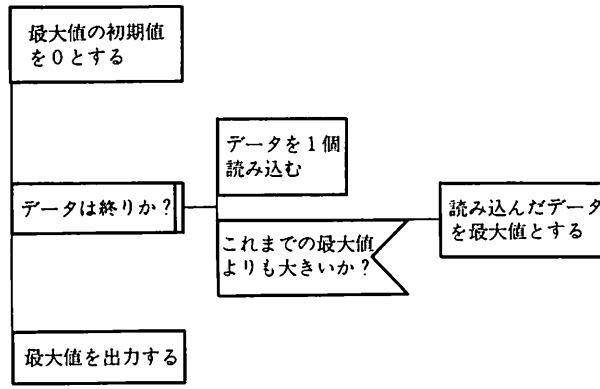


図5.4　PADの例（最大値を求めるプログラム）

(3)　情報隠蔽

★

モジュールの外部仕様と内部仕様が完全に分離され，そのモジュールを利用するために内部仕様を知る必要のないプログラムは，わかりやすい。

goto文の排除は，1つのモジュール内のプログラムの制御構造をきれいにするものであるが，プログラムの規模が大きくなると個々のモジュール内の構造

よりもモジュールの分割の仕方およびその結果として得られるモジュール間の関係の方がプログラム理解に重要である。情報隠蔽(information hiding)の概念は，このような観点からD.L.Parnasがモジュール分割の評価基準として導入したものであり，以下のように要約できる。

- ●各モジュールの機能，すなわち他のモジュールから呼び出されるときに必要となる外部仕様（インタフェース）をその機能の実現方式（インプリメンテーション）とは独立に決定する。
- ●各モジュールの実現方式に関する詳細な設計情報を他のモジュールから参照できないように隠しておく。

このようにモジュール分割を行えば，あるモジュールのプログラム構造を理解するときには，それが呼び出しているモジュールのプログラム構造まで調べる必要はなく，その外部仕様さえ理解しておけばよいので，プログラムの理解容易性が向上する。また，あるモジュールの実現方式を変更した場合でも，外部インタフェースを変更しなければそのモジュールを呼び出しているモジュールを変更する必要がないので，プログラムの保守性が向上する。

5.3 段階的詳細化技法

★

木構造のモジュール階層構造を持ち，上位のモジュールほど抽象度が高いプログラムは，わかりやすい。

5.3.1 トップダウン設計

プログラムの規模がある程度大きくなるとモジュールに分割して作成することになる。この時，モジュール構成を決める方法としてトップダウン設計技法とボトムアップ設計技法がある。ある程度見通しのきくものや過去に経験のあるものは，最初に幾つかの重要なモジュールを決めてから全体を設計するボトムアップなやり方が可能である。しかし，一般に新しいシステムの開発では，試行錯誤しながらトップダウンに設計していかざるをえない。

このトップダウン設計技法は段階的詳細化技法 (stepwise refinement) と呼ばれ, 1970 年代の初めに E. W. Dijkstra と N. Wirth によって提案された方法である。まず，始めに大まかなことを決め，次第に詳細な設計へ進む。このとき重要なことは，各段階の抽象レベルで意味的に閉じたプログラムを記述することである。たとえば，ある抽象レベルでデータを導入したとき，そのデータの構造に関する宣言のレベルとそのデータを参照する処理記述のレベルが同じ抽象度でなければならない。そして，次の段階でまたそれらの詳細化を行うという過程を繰返し，最後に実際のコンピュータで処理できるレベルに到達すれば，プログラムが完成する。

5.3.2　段階的詳細化の例

たとえば, 本方式の提案者である E. W. Dijkstra が提示した, 最初の 1000 個の素数を求めるプログラムを開発する場合を考えてみよう。

まず，始めにプログラムを次のように記述する。

[ステップ 1]

```
" print first thousand prime numbers "
```

これは，これから作成しようとするプログラムの要求仕様というべきレベルのものである。

次に，この具体的な処理方法として，テーブル T を導入し，まず 1000 個の素数をこのテーブルに書き込んだあと，まとめて印字出力をすることに決定すると，ステップ 1 のプログラムは次のように詳細化される。

[ステップ 2]

```
" table T "
" fill table T with first thousand prime numbers "
" print table T "
```

さらに，このテーブルTのデータ構造を配列変数とすることに決定すると，ステップ 2 のプログラムは次のように詳細化される。

[ステップ3]

```
" integer array T [1:1000] "
" make for I from 1 to 1000
      T[I] equal to the Ith prime number "
" print T[I] for I from 1 to 1000 "
```

このような段階的詳細化を最終的にコンピュータで実行可能なプログラムが得られるまで繰り返す。

この段階的詳細化技法の基本思想は，E. W. Dijkstra の言葉を借りれば，プログラミングというものは本来非常に知的で難しい作業なので，「一度に1つの決定」(one decision at a time) だけをすることを繰り返しながら最終的なプログラムに到達することによってはじめて誤りの無いプログラムを作れるというものである。

これは，従来，詳細設計とコーディングの2つの工程に分かれていたプログラミングを統一的手法で一体化したものと考えることができる。

5.3.3 実際のプログラム例

この段階的詳細化技法において，プログラムの理解容易性の観点から重要なことは，詳細化の各段階がプログラムとして明示的に記述され，プログラム全体が各々の段階の抽象度を保持したモジュール階層構造となることである。段階的詳細化技法を適用しても，最後の段階で得られたプログラムだけをプログラムとして残したのでは，プログラムの理解容易性や保守性の効果は半減する。ところが実際にはプログラミング言語として従来の手続き型言語を用いた場合，次のような欠点がある。

①処理手順の抽象度に合わせたデータ定義を可能にするためのデータ型定義機能が無い。

②処理手順の詳細化を通常のサブルーチンのような手続きで表現した場合，性能劣化を招く。

③手続き名を1つの識別子で表現するため，名前の付け方が難しい。

ここでは，筆者等が開発し，制御用応用プログラムの開発に使用されている

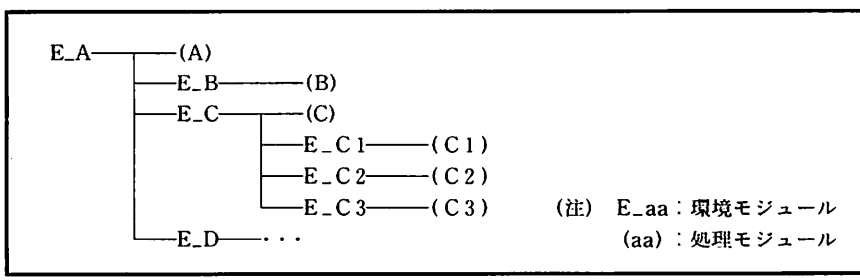
リアルタイムシステム向け構造化プログラミング言語SPL (Software Production Language)を紹介する。上記の従来の手続き型言語の欠点を解決するため，SPLでは，次のような段階的詳細化技法の支援機能を設けている。

①処理の詳細化にあわせてデータ構造の詳細化の過程を記述するために，新たなデータ型を定義できる。
②処理の段階的詳細化技法に伴って発生する手続き呼び出しの実行効率の向上を計るため，手続きのインライン展開を指示できる。
③各段階の処理内容とデータの内容を的確に表現するために，手続き名およびデータ型名は複数の識別子を用いて自然語風に記述できる。

実際に段階的詳細化技法を用いて開発したプログラム例を**図5.5**に示す。これは，SPLコンパイラが分割コンパイル方式の実現のために，仮想記憶上でモジュールライブラリを処理するときに用いるページングアルゴリズムを分析するプログラムである。全体で600行のプログラムであるが，図にはその一部を示している。図の(a)は，木構造のモジュール階層を表している。

図の(b)は最上位レベルのプログラム，図の(c)は第2レベルのプログラムの一部を示している。上記①の特徴であるデータ型定義機能の例として，PAGE SEQUENCEというデータ型名が最上位で導入され，そのデータ型の変数PSEQが宣言されている。そして，そのデータ型の詳細は第2レベルのプログラムの中で以下のように定義されている。

```
type PAGE SEQUENCE = array (1000)
```



(a)　モジュール階層

図5.5　段階的詳細化技法を適用したプログラムの例（続く）

```
environment E_A;
  dcl;
    var  PSIZE : int,
         PSEQ : PAGE SEQUENCE;
    const SSIZE=64;
  end;
end E_A;
process A(E_A);
  function PAGING opt(main);
    for PSIZE=SSIZE,SSIZE*8 through PSIZE*2
      repeat
        GENERATE PAGE SEQUENCE (PSEQ) FROM TRACE DATA;
        ANALYZE PAGE SEQUENCE (PSEQ);
        APPLY PAGING ALGORITHMS TO PAGE SEQUENCE (PSEQ);
      end;
  end PAGING;
end A;
```

(b) 最上位レベルのプログラム

```
environment E_B (E_A);
  spec;
    function GETADR(ADRX : int(2)) opt(sub);
  end;
  dcl;
    const ENDMARK = -1;
    type PAGE SEQUENCE = array(1000);
  end;
end E_B;
process B (E_B);
  function GENERATE PAGE SEQUENCE (PSEQ) FROM TRACE DATA opt(open);
    var NO : int,
        ADRX : int(2);
    NO = 1;
    repeat
      GETADR(ADRX);
      if ADRX = ENDMARK then PSEQ(NO) = ENDMARK;
                             exit LOOP;
      end;
      PSEQ(NO) = ADRX / PSIZE + 1;
    end LOOP;
  end GENERATE;
end B;
```

(c) 第2レベルのプログラム

図5.5 段階的詳細化技法を適用したプログラムの例

上記②の特徴であるインライン展開指示機能の例として，図の(c)の手続きANALYZE PAGE SEQUENCEの定義のところで，**opt(open)**が指定されている。また上記③の特徴である手続きやデータ型名の自然語風記述は見たとおりである。

5.4　データ抽象化技法

★ *データの構造とそのデータ操作の記述レベルの抽象度が高いプログラムは，わかりやすい。*

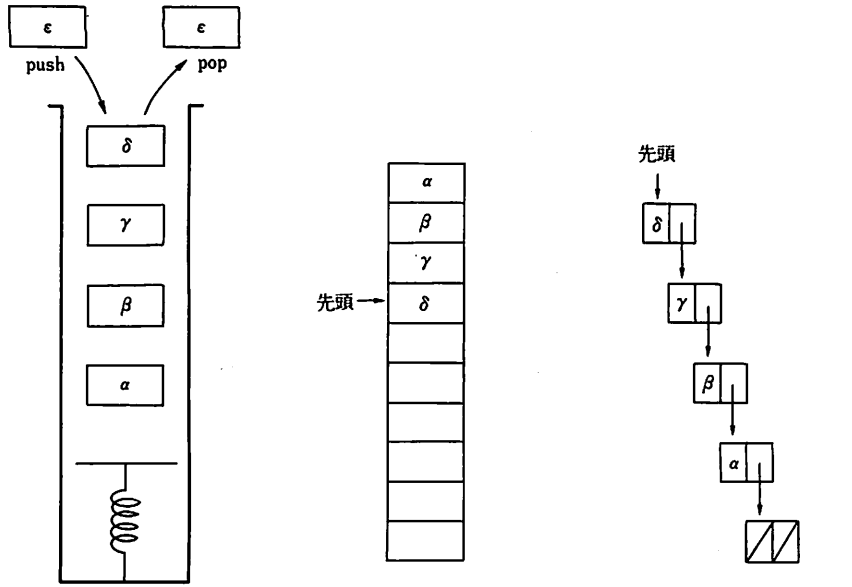
5.4.1　抽象データ型

データ抽象化(data abstraction)の基本的な考え方は，データの詳細な構造とデータ操作手続きをまとめて定義し，カプセル化(encapsulation)するとともに，そのデータへのアクセスは，データ操作手続きを通してのみ可能とするものである。このような方式には次のような利点がある。

①そのデータの参照側では，詳細なデータ構造を知る必要がなく，データアクセス時にはその処理内容に対応するデータ操作手続きを呼び出すだけでよいので，より抽象的なレベルでプログラムを記述できる。

②そのデータの定義側では，データ操作手続きの外部インタフェースさえ変更しなければ参照側への影響がないので，データ構造やデータ操作手続きの処理手順（アルゴリズム）を自由に決めてよい。そのため，処理速度や所要メモリ量などの外部条件に応じて実現方式を変更することが容易である。

たとえば，スタックのデータ構造を考えてみよう。スタックとは，図**5.6**(a)に示すように，データを順に積み重ねたり，積み重ねたデータを先端から順に取り除いたりできるようなデータ構造をいう。データの出し入れが先に入れたものほど取り出すときは後になるという意味で先入後出（FILO：first-in last-out）の構造になっている。いわゆるキューが先入先出（FIFO：first-in first



(a) スタックの概念図　(b) 配列構造による実現例　(c) リスト構造による実現例

図5.6 データ抽象化の例（スタックの場合）

-out）の構造になっているのと対照的である。このスタックのための基本的なデータ操作は以下の2つである。

- push：スタックへのデータの追加
- pop ：スタックからのデータの取り出し

このスタックのデータ構造を実際にプログラム内で実現する方法としては，図の(b)のように配列構造を利用する方法や図の(c)のようにリスト構造を利用する方法が考えられる。たとえば，整数型データを100個まで蓄積できるスタックをPascalで配列構造を用いて実現すると，図5.7に示すようなプログラムになる。

このような準備をしておけば，実際のスタックを用いた処理は，2つのデータ操作手続きであるPUSHとPOPを用いて記述すればよく，スタックを利用する側は詳細なデータ構造を知る必要がない。そして，たとえば後で処理速度よりもメモリの節約が重要になり，スタックのデータ構造を配列構造からリス

```
var STACK : record
              TOP : integer;
              ARY : array  [1..100]  of integer
              end;

procedure PUSH ( X : integer );
  begin
    if STACK.TOP = 100 then ERROR( 'stack over' )
                       else begin
                              STACK.TOP = STACK.TOP + 1;
                              STACK.ARY [STACK.TOP]  := X
                            end
  end;

procedure POP( var X : integer );
  begin
    if STACK.TOP = 0 then ERROR( 'stack empty' )
                     else begin
                            X := STACK.ARY [STACK.TOP] ;
                            STACK.TOP := STACK.TOP - 1
                          end
  end;
```

図5.7　Pascalによる抽象データの定義(整数型スタックの例)

ト構造に変更した場合でも，スタックを利用する側はプログラムを変更する必要がない。

データ抽象化の基本思想は先に述べた情報隠蔽と同じである。しかしながら，このように抽象データを直接定義する方法は，類似の機能を有するデータが幾つも必要になったとき，その度にデータ構造とデータ操作手続きを定義しなければならない。そこで，実際にはデータ抽象化機構は抽象データ型（abstract data type）のユーザ定義機能として実現されることが多い。

1970年代には，構造化技法の研究の一環として抽象データ型機能を備えたプログラミング言語の研究が盛んに行われた。Simula 67のclassの概念を基礎にしてCLU (MIT)，Alphard (CMU) などが開発された。筆者らも先に述べた段階的詳細化機能とデータ抽象化機能を備えたSPLを1977年に開発して以来，現在も制御用応用システムに適用している。さらに，1980年代に入ってAda（米国国防総省）が開発されている。ここでは，CLUとAdaの例を紹介する。

5.4.2 CLUの例

抽象データ型を備えたプログラミング言語CLUは，1974年頃にMITのB. Liskov女史が開発したものである。スタックは，図5.8のように定義される。1行目では，stackという名前の抽象データ型はtというデータ型の引数を持ち，データ操作手続きとしてはcreate，push，popを用意することを示している。2行目では，stack型を実際にはtというデータ型の配列で実現することを示している。3行目以降では，3つのデータ操作手続きを定義している。

ここで定義した抽象データ型stackを実際にプログラムの中で使用するときは，たとえば，以下のように宣言する。

```
a : stack[int] : =stack[int]$create( ) ;
```

この宣言により，整数型データ用スタックの実体が作られる。このstack型変数aの操作は，データ操作手続きpushおよびpopを用いて次のように行う。

```
stack[int]$push(a, 123) ;
b : =stack[int]$pop(a) ;
```

CLUのデータ抽象化機能の主な特徴は以下のようなものである。

①データ構造とその操作手続きをclusterと呼ぶ入れものの中にカプセル

```
stack=cluster  [t:type]  is create, push, pop
       rep=array  [t]
  create=proc ( ) returns (cvt)
           return (rep$new( ))
           end create
  push=proc (s:cvt, x:t)
        rep$addh(s, x)
        end push
  pop=proc (s:cvt) returns (t) signals (empty)
       if rep$empty(s)
       then signal empty
       else return (rep$remh(s))
       end
       end pop
  end stack
```

図5.8 CLUによる抽象データ型の定義(STACK型の例：文献(23)より引用)

化できる。なお，この入れ物の名前は言語により異なり，SIMULA や Smalltalk-80 では class, Alphard では form, Ada では package と呼ばれる。

②抽象データ型の引数としてデータ型を指定できるので，整数型データ用のスタックと実数型データ用のスタックを別々に定義する必要はない。たとえば，先の例のように整数型のスタックが必要なときは変数宣言時に実引数として **int** を指定すればよい。

③データ操作手続きの呼び出しは「抽象データ型名 $ 手続き名」の構文形式で行うため，手続き名はプログラム全体で一意に決める必要はない。異なる抽象データ型の定義では同じ手続き名を使ってもよい。

④データ操作手続きの定義内では抽象データ型の詳細なデータ構造に対する操作が必要なので，**cvt** という特殊な型指定を行うことにより，それを可能としている。先の例では，push という手続きの第1引数は stack 型であるが，定義側の仮引数の型指定を **cvt** とすることにより，stack 型データ s を配列とみなし，配列の最後尾に第2引数のデータを追加する addh という配列型操作を可能にしている。

5.4.3 Ada の例

Ada は，米国国防総省（DoD：Department of Defence）が米軍の組み込み型コンピュータシステム（embedded computer system）に代表される制御用リアルタイムシステムのための高水準プログラミング言語として開発したものである。1970 年代中頃，国防費の中でコンピュータ関連の費用が急増しはじめており，とりわけソフトウェア費用の伸びが大きく，そのままでは国防費の大半がソフトウェアに費やされてしまうという危機感があった。当時，陸軍，海軍，空軍は，それぞれ独自のプログラミング言語を使用していたため，相互の流用性がまったくなく，しかも多種多様な機種が導入され，異なる機種毎に類似のソフトウェアを新規に開発していた。そこで，DoD は，1975 年に，保守性，移行性に優れた機種独立の高水準言語の開発に着手し，1980 年に完成した。その後，1988 年に ISO 規格，1991 年には日本でも JIS 規格となった。余談になるが，Ada という名前は，英国の詩人 Byron の娘の Augusta Ada の名前に由来している。彼女は，先に2章で述べた計算手順の自動化をもたらした C.

Babbageの協力者で，女性プログラマの元祖ともいわれる。

Adaは，委員会でまず要求仕様を作ることからはじめたため，具備すべき機能が膨大になり，複雑で大きな言語仕様になってしまったが，その反面，1970年代の主要なプログラミング技法を取り入れたものになっている。1955年のFORTRANに始まったプログラミング言語の量的高級化が1965年のPL/Iに集約されたように，1970年前後に始まった質的高級化がAdaに集約されたと見ることができる。Adaの主な特徴は以下のようなものである。

①データ抽象化技法を含むパッケージ機能
②ランデブと呼ばれるタスク間通信機構を備えた並列処理機能
③割込み処理を容易に記述できる例外処理機能
④移行性と性能の両立のための機種依存部記述機能

ここでは，第1の特徴であるデータ抽象化機能について，汎用体パッケージを用いたスタックの例で説明する。

(1) 抽象データの作成

整数型データ用のスタックや文字型データ用のスタックなど，いろいろなデータ型用のスタックを簡単に作るための元になる汎用体パッケージSTACKの定義を**図5.9**に示す。

汎用体を示すキーワード**generic**に続く2行は，この汎用体から実体（専用体）を作り出すときに指定するパラメータを宣言している。1番目のSIZEは，スタックに入るデータの最大個数を指定するもので，正の整数である。POSITIVEは，整数型INTEGERの部分型で，値として1から整数型の最大値までをとることが次のように定義されている。

```
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;
```

2番目のパラメータのITEMは，スタックに入るデータのデータ型を指定するもので，その詳細なデータ構造は外部からは見えない。

次にキーワード**package**に続く3行は，パッケージSTACKの外部仕様を示すもので，可視部(visible part)と呼ばれている。パッケージSTACKの操作手続きとして，ITEM型の入力パラメータを持つPUSHとITEM型の出力

```
generic
  SIZE : POSITIVE ;
  type ITEM is private ;
package STACK is
  procedure PUSH(E : in ITEM) ;
  procedure POP(E : out ITEM );
  OVERFLOW, UNDERFLOW : exception ;
end STACK ;

package body STACK is
  type TABLE is array(POSITIVE range <> )of ITEM ;
  SPACE : TABLE(1..SIZE );
  INDEX : NATURAL := 0 ;

  procedure PUSH(E : in ITEM) is
  begin
    if INDEX >= SIZE then
      raise OVERFLOW ;
    end if ;
    INDEX := INDEX + 1 ;
    SPACE(INDEX) := E ;
  end PUSH ;

  procedure POP(E : out ITEM) is
  begin
    if INDEX = 0 then
      raise UNDERFLOW ;
    end if ;
    E := SPACE(INDEX) ;
    INDEX := INDEX - 1 ;
  end POP ;

end STACK

-- 例外処理 (OVERFLOW, UNDERFLOW) の定義省略
```

図5.9 Adaによる抽象データの定義（スタックの例）(文献(24)より引用)

パラメータを持つPOPがある。例外処理として，OVERFLOWとUNDERFLOWを用意する。これらの例外処理はたとえば次のように定義される。

```
exception
  when OVERFLOW=>PUT("STACK IS OVERFLOW");
  when UNDERFLOW=>PUT("STACK IS EMPTY");
```

キーワード **package body** 以下がパッケージ STACK の本体で，外部からは見えない。最初にデータ型 TABLE を ITEM 型の配列として定義している。添字部分はここでは指定していないが，次の変数 SPACE の宣言のところで，そのデータ型を TABLE とすると共に，添字の範囲を具体的に指定している。この SPACE が実際のスタックエリアになる。その次の変数 INDEX は，スタックに積まれた最後のデータを指すポインタであり，0 以上の整数値をとるので，整数型の部分型である NATURAL 型が指定されている。初期値 0 も指定されている。その後にデータ操作手続き PUSH と POP の手続き本体が定義されている。**raise** は例外を発生させる文である。

さて，実際のスタックの実体は，この汎用体パッケージを用いて，次のようにして得ることができる。

```
package STACK_INT is new STACK(200, INTEGER);
package STACK_BOOL is new STACK(100, BOOLEAN);
```

このようにして生成した整数型スタックのパッケージと論理型スタックのパッケージへのアクセスは，その専用の操作手続きを用いて次のように行う。

```
STACK_INT. PUSH(123);
STACK_BOOL. POP(A);
```

(2) 抽象データ型の作成

次に，STACK をデータ型として利用するための元になる汎用体パッケージの定義を **図 5.10** に示す。

汎用体を示すキーワード **generic** に続く行は，(1) と同様に，この汎用体から実体（専用体）を作り出すときのパラメータとして，スタックに積むデータの型を指定することを宣言している。

キーワード **package** に続く可視部は，(1) と同様の操作手続き PUSH と POP，例外処理 OVERFLOW と UNDERFLOW の宣言に加えて，先頭で

```
generic
  type ITEM is private ;
package ON_STACKS is
  type STACK(SIZE : POSITIVE) is limited private ;
  procedure PUSH(S : in out STACK ; E : in  ITEM) ;
  procedure POP(S : in out STACK ; E : out  ITEM) ;
  OVERFLOW, UNDERFLOW : exception ;
private
  type TABLE is array(POSITIVE range<>) of ITEM ;
  type STACK(SIZE : POSITIVE) is
    record
      SPACE : TABLE(1..SIZE) ;
      INDEX : NATURAL := 0 ;
    end record ;
end ;

-- パッケージ本体(ON_STACKS)の定義省略
-- 例外処理(OVERFLOW, UNDERFLOW)の定義省略
```

図5.10　Adaによる抽象データ型の定義（STACK型の例）(文献(24)より引用)

STACKの型宣言をしている。スタックの大きさはパラメータとして指定するようにしている。キーワード **limited private** はこのSTACK型のデータへの操作が後で定義する専用の操作手続きに限定され，一般の代入や一致または不一致の比較も不可であることを規定している。

キーワード **private** に続く部分は，密閉部 (private part) と呼ばれ，ここで宣言された内容はパッケージの外部からは見えない。ここでは，パッケージの定義内だけで用いるデータ型TABLEの型宣言と可視部で宣言したSTACKの詳細構造が規定されている。TABLEについては(1)と同様である。STACKは，スタック本体を示すSPACEとポインタINDEXを組み合わせたレコード型として実現している。

なお，パッケージ本体の定義は省略しているが，PUSHおよびPOPの定義に関して，スタックが引数で与えられるために，SPACEとINDEXへのアクセスがS. SPACEおよびS. INDEXの形でなされる以外は同じである。

さて，実際のスタックの実体は，この汎用体パッケージを用いて，次のようにして得ることができる。

```
declare
  package STACK_REAL is new ON_STACKS(REAL) ;
use STACK_REAL ;
  S : STACK(100) ;
begin
  ...
  PUSH(S, 3.14) ;
  ...
end ;
```

ここでは，まず汎用体パッケージ ON_STACKS の実体として STACK_REAL を生成し，use 節を用いてその可視部の STACK，PUSH，POP を外部から見えるようにしている。この時点では，まだスタックの本体は作られていないので，次に他のデータ型と同じように STACK 型の変数宣言をしている。この時にスタックの大きさを実引数 100 で指定している。PUSH(S, 3.14)は，このようにして生成されたスタックの実体へのアクセスの例である。

ここで(1)の抽象データの例と(2)の抽象データ型の例を比較してみる。(1)では，汎用体パッケージから実体を生成したときにすでに抽象データとしてのスタック本体がパッケージとして作られていた。(2)では，汎用体パッケージからの実体の生成は STACK 型という抽象データ型の生成を意味し，スタック本体の生成はさらにそのデータ型を用いた変数宣言で行う。そのためスタックの大きさを指定する場所や操作手続きのアクセス方法が両者で異なっている。たとえば，PUSH を用いてあるデータをスタックに積むとき，(1)ではスタック本体のあるパッケージの実体の名前を付加して，STACK_INT. PUSH(123)のように記述する。一方，(2)では，スタック本体のある変数名を実引数として指定して，PUSH(S, 3.14)のように記述する。