第4章　構造化プログラミング用言語機能の実現方式

第４章　構造化プログラミング用言語機能の実現方式[C9)]

4.1　概　　要

構造化プログラム，即ち，理解容易なプログラムを作成するための方法として，データ抽象化技法，段階的詳細化技法，構造化コーディング技法などがあるが，これらの技法を言語機能として実現する場合，各々に最適と思われる言語仕様を独立に決定したのでは，言語としての統一性に欠ける。そこで，前章では，まず基本言語構造として，木構造形式のモジュール階層構造を導入した。本章では，構造化プログラミング用言語機能をこの基本構造の上にどのように具体化するかについて述べるが，ここでは，前章で述べた新言語への要求機能をすべて満たしながら，言語としての統一性を保つことが重要な課題である。

4.2　データ抽象化機構

データ抽象化の基本的な考えは，データの詳細な構造とデータ操作手続き群をまとめて定義し，そのデータへのアクセスは操作手続きを通してのみ可能とするものである。このような方式の主な利点は次のようなものである。

(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，その処理内容に適した操作手続きを引用するだけで良いので，より抽象的な水準でプログラムを記述できる。

(2)　そのデータの定義側では，操作手続きの外部インターフェイスさえ変更しなければ参照側への影響がないので，データ構造や操作手続きの処理手順（アルゴリズム）は自由に決めて良い。そのため，所要メモリ量や処理速度などの外部条件に応じたプログラムの変更が容易になる。

ＳＰＬでは，先に３.３節の図３.１の例でも説明したように，対象とするデータの宣言を含む環境モジュールの下にその操作手続きを含む処理モジュールを結合することにより，データ抽象化が可能である。

しかしながら，このような方法だけでは，類似の機能を有するデータが幾つも必要になった時，その度にデータ構造と操作手続きを定義する必要が生じる。そこで，実際には，データ抽象化機構は抽象データ型定義機能として実現されることが多く，従来のデータ抽象化支援言語はこのための特殊なデータ抽象化機構を導入している。ＳＰＬでは，３.３節でも述べたように，このような特殊な方式では他の要求を満たせないことから，Pascal 風のデータ型定義機能を導入，拡張することによって類似の機能を実現した。その具体的な機構は次のようなものである。

(1)　抽象データ型の詳細なデータ構造はデータ型定義機能を用いて定義する。その際，配列の要素数，初期値，精度など，データ型指定時に定数として記述する部分はパラメータ化することができる。

(2)　この抽象データ型のデータに対する操作手続きのすべてを１つの処理モジュールとしてまとめ，

抽象データ型を定義している環境モジュールの下に結合する。

(3) この利用側では，対象とするデータを抽象データ型名を用いて宣言し，そのデータに対するアクセスは操作手続きの引用によって行う。

ここで，ＳＰＬによる抽象データ型の記述例として，スタック処理を対象にしたものを図4.1に示す。なお，ここで導入するモジュールの階層関係は図3.1に準じる。まず，図の(a)では，抽象データ型ＳＴＡＣＫが定義される。これは，スタックに積まれる要素数の最大をパラメータとするもので，その詳細なデータ構造は，等号の右側で定義される。即ち，ＴＯＰ，ＳＴＫ，ＭＡＸの3つのフィールドを有する構造体であり，ＳＴＫの配列要素数およびＭＡＸの初期値としてパラメータＬＥＮＧＴＨが使用されている。次に，図の(b)では，ＳＴＡＣＫ型データに対する操作手続きとして，ＰＵＳＨとＰＯＰが定義される。仮引数Ｓの型宣言でＳＴＡＣＫ(*)が指定されているが，ここで＊は配列要素数を実引数側と同じにすることを意味する。なお，ＳＰＬでは，手続きは機能単位であるという考えから手続き定義の先頭キーワードにはfunction 又は func を用いる。

このように図の(a)と(b)で定義した抽象データ型ＳＴＡＣＫの使用例を図の(c)に示す。この例では，長さ１００のＳＴＡＣＫ型データＳ１が導入され，そのスタックに変数ＶＡＬＵＥの値を積むための操作手続きＰＵＳＨとその値を読み出すための操作手続きＰＯＰが引用されている。ここで，モジュール間の関係をみてみると，操作手続きの定義を含む処理モジュールＰ２は，ＳＴＡＣＫ型の定義を含む環境モジュールＥ２を親モジュールとしているため，ＳＴＡＣＫ型の詳細なデータ構造の中のＴＯＰ，ＳＴＫ，ＭＡＸなどのフィールドを参照できる。しかし，ＳＴＡＣＫ型の利用側である処理モジュールＰ１は，その親または先祖の環境モジュールとしてＥ２を持たないため，ＳＴＡＣＫ型の詳細なデータ構造を直接にアクセスすることはできない。なお，この例からもわかるように，ユーザ定義データ型名の有効範囲は，手続き名の場合と同様に，制限を設けないようにした。

4.3 段階的詳細化機能

段階的詳細化技法の基本的な考え方は，プログラミングは非常に知的で難しい作業なので，「１度に１つの決定」(one decision at a time) をすることを繰返しながら，最終的なプログラムに到達することによって，誤りのないプログラムが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つの工程に分けて行っていたプログラミングを一つの統一的な手法を用いて一様化するものである。

その本来の目的はプログラミングの容易化にあるが，その詳細化の各段階をプログラムとして記述することにより，次のような利点が生じる。

(1) 従来，ＧＦＣ(general flow chart) などを用いて記述していた詳細設計書が不要になる。また，ＧＦＣを採用した場合でも，その階層構造をそのままプログラムに表現できるので，設計書とプログラムの直接的な対応がとれる。

(2) 各段階のプログラムは，その段階に応じて抽象化されているので，理解が容易である。

```
environment E2(E1);
  .....
  declaration;
    type STACK(LENGTH:int)
              =(TOP:int init(0),
                STK(LENGTH):real,
                MAX:int init(LENGTH));
  end;
   .....
end E2;
```

(a) ＳＴＡＣＫ型の定義

```
process P2(E2);
  func PUSH(V) TO(S);
    par V:real,
        S:STACK(*);
    S.TOP=S.TOP+1;
    if S.TOP .GT. S.MAX
      then ERROR(E013);
      else S.STK(S.TOP)=V;
    end;
  end PUSH;

  func POP(V) OUT OF(S);
    par V:real,
        S:STACK(*);
    if S.TOP .EQ. 0
      then ERROR(E014);
      else V=S.STK(S.TOP);
           S.TOP=S.TOP-1;
    end;
  end POP;
   .....
end P2;
```

(b) ＳＴＡＣＫ型データ操作手続きの定義

```
process P1(E1);
 func EVAL;
  var S1:STACK(100);
  .....
   PUSH(VALUE) TO(S1);
  .....
   POP (VALUE) OUT OF(S1);
  .....
end P1;
```

(c) ＳＴＡＣＫ型の使用例

図４．１　データ抽象化の例

(3) 最終段階またはそれに近いプログラムで具体化されたデータ構造や処理手順は，所要メモリ量や処理速度などの外部条件に応じて容易に変更でき，それより上位のプログラムには影響が及ぶことはない。

そこで，ＳＰＬでは，次のような段階的詳細化支援機能を設けた。

(1) 上位段階のプログラムの処理内容を自然語風に記述するために，手続き名は複数の単語の並びで構成できるようにした。

(2) 処理の詳細化に合せたデータ構造の詳細化の過程を記述するために，データ型定義機能を設けた。そして，データ内容を自然語風に記述するために，手続き名と同様に複数の単語の並びで構成できるようにした。

(3) 本方式によって生じる，一度だけ引用される手続きの実行効率の向上をはかるため，手続きのインライン展開指示機能を設けた。

先の図4.1のプログラムでは，まず図の(c)のプログラムを初めに作成したと考えると，段階的詳細化の例になる。即ち，処理モジュールＰ１の記述時にまず新しいデータ型ＳＴＡＣＫとその操作手続きＰＵＳＨとＰＯＰを導入して利用し，その後で，環境モジュールＥ２および処理モジュールＰ２の中でそれらの定義を行っている。

ここで，もう１つの例として，実際に段階的詳細化法を用いて開発した実用プログラムの一部分を図4.2に示す。これは，6.3節で述べる共通ライブラリ用ページングアルゴリズムの分析に用いたプログラムであり，プログラムリストは付録3に示すが，全体のモジュール階層は図の(a)のようになっている。このプログラム全体で用いる変数や定数名は，図の(b)に示すように，最上位の環境モジュールＥ＿Ａで宣言され，その下の処理モジュールＡでは第１段階の処理が記述されている。その後，第２段階では，第１段階で引用された３つの手続きとデータ型ＰＡＧＥ＿ＳＥＱＵＥＮＣＥ の詳細化が行われる。図の(c)は２番目の手続きの詳細化例である。その処理モジュールＣの中で引用されている手続き間の共通データが環境モジュールＥ＿Ｃで宣言されている。また，手続き定義の先頭行のオプション指定 open は，この手続きのインライン展開を指示している。一方，図の(b)の環境モジュールＥ＿Ａで引用されたデータ型ＰＡＧＥ－ＳＥＱＵＥＮＣＥ の詳細化は，その下の環境モジュールＥ＿Ｂの中で次のように行われている。

```
type  PAGE_SEQUENCE  =  array ( MAXNO ) ;
```

なお，手続きを自然語風に記述できる言語の例としてＡＬＧＯＬ[N3)]がある。例えば，ＳＰＬの手続き呼び出しである，

```
PUSH ( VALUE ) TO ( S1 ) ;
```

に対して，ＡＬＧＯＬでは，

```
PUSH ( VALUE ) TO : ( S1 ) ;
```

と記述できる。即ち，ＡＬＧＯＬでは，引数の区切りの「，」の代りに「）文字列：（」を用いて良いことになっており，上の例は，

PUSH（VALUE，S1）；

と全く同じものになる。しかしながら，このような方法では，手続き名の識別はあくまで先頭の識別子のみで行われるため，使用法が限定される。一方，ＳＰＬでは，手続き名を構成するすべての単語および引数の位置の一致によって識別するため，プログラムの信頼性を保持しながら，自然な表現の手続き名を使用できる。

また，段階的詳細化機能の１つである手続きのインライン展開機能はデフォルト機能にしており，手続き定義のoption指定部分が省略された場合は常にインライン展開を実施する。その際，引数の受け渡しはcall－by－name によって行うことになるが，option指定でsubまたはmainが指定された外部手続きの場合は，call－by－addressとなる。

4.4　外部仕様明記機能

ＳＰＬにおける手続きやユーザ定義データ型は，データ抽象化と段階的詳細化の双方で用いるため，プログラムのどこからでも引用可能としている。しかも，それらの用途に用いられる場合は，手続きやデータ型は定義される前に引用されることが多い。そこで，それらの定義側と引用側でのインターフェイスの不一致誤りを防ぐために，定義に先んじて引用する手続きやデータ型は，その外部仕様を明記させるようにした。

例えば，図4.1のプログラムにおいて，処理モジュールＰ２の中でエラー処理手続きＥＲＲＯＲが引用されているが，この手続きはエラーコードを引数に持つことだけが決められていて，実際の処理内容は未だ定義されていないとしよう。この場合，図4.1の環境モジュールＥ２の中で次のような仕様記述が必要である。

```
specification ;
  func ERROR（CODE ： int） opt（sub）;
end ;
```

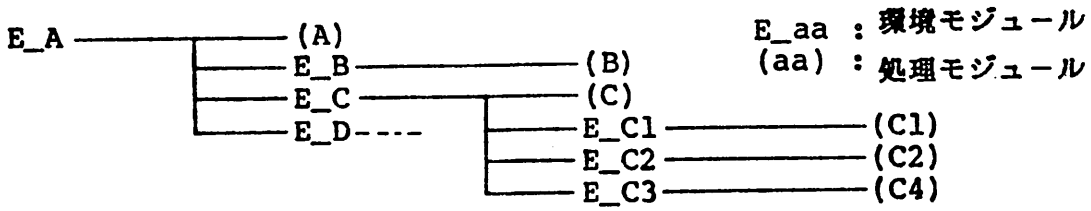
この情報に基づいて，言語処理系はモジュール間インターフェイスの型チェックを行うことになるが，その詳細は次章で述べる。

4.5　構造化コーディング

これまでに述べてきたデータ抽象化技法や段階的詳細化技法は，プログラムの機能分割を行い，理解容易なモジュール構造や手続き群を導くのに有効なものであった。一方，個々の手続きの処理手順をわかり易く記述する方法として構造化コーディング技法がある。これは，プログラム内の各文の実際に実行される順序を表わす制御構造を理解容易なものにするために，その障害となっているgoto文を排し，複合文，選択文，反復文だけを用いた単純な制御構造にするものである。

そこで，ＳＰＬの設計においてもこの技法を支援するために，制御文は次のようなものに限定した。

(1)　複合文：　幾つかの文の列を逐次的に実行するために，それらをbeginとendで囲んだもの



(a) モジュール階層

```
environment E_A;
  dcl;
    var   PSIZE:int,
          NPAGE:int;
    var   PSEQ :PAGE_SEQUENCE,
          NDATA:int;
    const MAXADR = 16384,
          SSIZE  =    64,
          MAXPG  =   256;
    const ENDMK  =    -1;
  end;
end E_A;

process A(E_A);
  function PAGING opt(main);
    for PSIZE=SSIZE, SSIZE*8 through PSIZE*2
      repeat
        INITIALIZE PAGING;
        ANALYZE PAGE_SEQUENCE;
        APPLY PAGING ALGORITHMS TO PAGE_SEQUENCE;
      end;
  end PAGING;
end A;
```

(b) 第一レベルのプログラム

```
environment E_C(E_A);
 dcl;
   var LTABLE(MAXPG):int,
       STABLE(MAXPG):(FREQ:int,
                      PGNO:int);
 end;
end E_C;

process C(E_C);
  function ANALYZE PAGE_SEQUENCE opt(open);
     EXAMINE LRU_PROPERTY;
     EXAMINE STATIC REFERENCE FREQUENCY;
     COMPARE LRU WITH STATIC;
  end ANALYZE;
end C;
```

(c) 第二レベルのプログラム

図４．２　プログラムの段階的詳細化の例

である。

(2) 選択文： ある条件に基づいて異なる処理をするもので，その条件の真偽に応じて二者択一の処理をする場合に用いるif ～ then ～ else ～ end 形式と，条件に応じて多方向に分岐するためにif ～ endの中に幾つかのis ～ then ～を並べた形式のものがある。

(3) 反復文： 繰返して実行する文の列をrepeat と end で囲んだもので，その繰返し処理の終了条件の記述方法には次のようなものがある。

(a) 繰返し処理の先頭で終了条件の真偽を判定するもの。

(b) 繰返し処理の最後で終了条件の真偽を判定するもの。

(c) 繰返し処理の制御変数に最初に初期値を与え，繰返し処理の先頭でその制御変数の値が終値を超えたか否かを判定するもの。

(4) ブロック脱出文： goto 文を完全に禁止した場合，例外処理の記述がかえって複雑になることがある。特に制御用応用プログラムにおいては，通常，割込処理や異常処理が重要な役割を果している。[S14)] そこで，その対策として，上記(1)～(3)の文の途中から抜け出すためのexit 文を設けると共に，手続きからの抜け出し用としてreturn 文，またプログラム全体からの抜け出し用としてstop 文を設けた。さらに，上記(1)～(3)の文につけたラベルを実引数として手続き呼び出しを行うことにより，その呼び出された手続きの中からそのラベルを指定したexit 文で抜け出すことも可能としている。

4.6 コンパイル時実行機能

本章では前節まで，構造化プログラミングのための直接的な言語機能について述べてきたが，次章以降との関連の深いコンパイル時実行機能についてここで述べておく。本機能は主に次の２つの目的で導入された。

その第１は，構造化プログラムの標準パッケージ化の促進である。即ち，構造化プログラミングを適用して作成したモジュールや手続きは，機能的な独立性が強く，他のシステムへ流用し易い。しかし，実際の流用時にはその一部分を変更する必要が生じることも多く，それが標準パッケージ化の障害となっている。そこで，このような標準パッケージの作成時には，コンパイル時実行機能を用いて変更部分を組み込んでおくことにより，その利用者は自分の用途に適したものを手に入れることができる。

本機能の第２の目的は，構造化されたプログラムの実行効率の向上にある。即ち，データ抽象化や段階的詳細化においては，手続きの引用側ではその機能やインターフェイス仕様を知る必要はあるが，その処理手順まで知る必要はない。そのため，引用側の前後の処理内容によっては，その手続きをインライン展開した時に冗長な処理が生じる場合がある。そこで，その手続きの定義時に，コンパイル時機能を用いて，このような冗長な処理を除くような記述が可能である。

例えば，図4.1のプログラムの手続きＰＵＳＨにおいて，スタックにデータを積む前に必ずスタッ

クに空きがあるか否かのチェックをしている。しかし，この処理は，手続きＰＵＳＨを引用する側でスタックに空きのあることが保証されている場合には無駄である。そこで，図4.3に示すように，コンパイル時実行機能を用いてエラー処理の要否に応じたオブジェクトコードの選択ができるようにすることができる。この例においては，%varでコンパイル時変数ＬＥＶＥＬの宣言を行い，%で始まる代入文でＬＥＶＥＬの値を１か２にしておく。その後，手続きＰＵＳＨが引用された時，%if文が実行され，ＬＥＶＥＬの値が１の時はエラーチェックを含むコードが生成される。

4.7 結　　言

プログラム開発において，その分野に適したプログラミング方法論の適用を徹底するためには，その方法論を具現化した言語でプログラムを記述するのが良い。本章では，新言語への要求機能を満たすために，前章で導いた木構造形式のモジュール階層構造を基礎にした，データ抽象化機構，段階的詳細化機能，構造化コーディングなどの実現方式について述べた。その中で，特にデータ抽象化と段階的詳細化に共通の核としてデータ型定義機能を導入した。そして，このデータ型と手続きは，各々，プログラムを構成する基本要素であるデータとアルゴリズムの抽象化であるという考えから，それらの構文は同じにした。

このデータ型定義機能には，配列要素数や初期値などを引数にすることを許し，データの抽象化に柔軟性を持たせている。しかしながら，例えば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の抽象データ型をまとめて定義する場合に必要となるデータ型の引数化は行わなかった。その理由は，ＳＰＬの対象分野である制御用応用プログラムでの必要性が高くないことの他に，言語の複雑化およびそれに伴う処理の複雑化を避けるためである。従って，本機能の実用的な実現方式は今後の課題である。

一方，構造化プログラミングによる負の効果についても幾つかの解決方法を用意した。即ち，まず第１に，goto 文を排したことによって例外処理が複雑化することを避けるために，ブロック脱出用の exit 文を導入した。第２には，手続き間リンケージのオーバーヘッドや冗長な処理によるオブジェクト効率の低下を防ぐために，手続きのインライン展開機能やコンパイル時実行機能を設けた。

以上，本章で述べた言語機能は制御用システムに限らず一般に有効な技法と思われる。一方，制御用システムに重要な並行処理機能については，ＳＰＬでは，従来言語のＰＣＬと同様に，アセンブラレベルのタスクマクロを手続き呼び出しの形のシステム交信文として導入しており，本機能の高度化は今後の課題である。

```
   .....
%var LEVEL:int;
   .....
%LEVEL=1;
   .....
%LEVEL=2;
   .....
  func PUSH(V) TO(S) opt(open);
    par V:real,
        S:STACK(*);
    S.TOP=S.TOP+1;
   %if LEVEL
     is 1 then
           if S.TOP .GT. S.MAX
             then ERROR(E013);
             else S.STK(S.TOP)=V;
            end;
     is 2 then S.STK(S.TOP)=V;
    end;
  end PUSH;
   .....
```

図４．３　コンパイル時実行機能の例