トップダウン・プログラミング言語ＳＰＬにおけるモジュール概念について

野木兼六*，中所武司*，中田育男*，浜田亘曼**，林　利弘***
（日立製作所*システム開発研究所，**日立研究所，***大みか工場）

１．はじめに

ストラクチャード・プログラミングが提案されて以来、ソフトウェア開発方式の改善が活発に進められている。ストラクチャード・プログラミングという概念は、明確には定義されていないけれど、２つの手法から成ると言われている。すなわち、１つは、ＧＯＴＯ文の代りに構造化された制御文を使用し、静的構造と動的構造の対応が単純なプログラムを作成するという手法であり、もう１つは、段階的な抽象化により、階層化されたプログラムを作成するという手法である。当初は、ＧＯＴＯ論争〔1〕としてよく知られているように、多くの議論が前者に集中していたが、近年は、データの抽象化の議論に見られるように、後者の重要性が認識され、これをサポートするプログラミング言語〔2,3,4〕がいくつか提案されている。最近ではさらに、複合設計法〔5〕、モジュール・インターフェイス記述言語〔6,7〕、要求仕様記述言語〔8,9〕など、より上位の設計手法に対する関心が高まっている。

制御用計算機の分野でも、アプリケーション・ソフトウェアの多様化・大規模化に伴ない、プログラムの品質向上および生産性向上が強く望まれるようになってきており、ストラクチャード・プログラミングを始めとする最近の新しいソフトウェア開発手法を導入する気運が高まっている。一方、この分野では、従来からアプリケーション・ソフトウェアの標準化、問題向言語（ＰＯＬ）の開発が盛んに行なわれてきたが、プログラムの変更容易性・拡張性に問題があり、ユーザの多様なニーズに応えることが困難であった。このため、標準化パッケージや問題向言語定義パッケージの開発に適した、より柔軟性の高いプログラミング言語に対する期待が大きくなっている。

ＳＰＬ（Software Production Language）は、このような状況に対処するために開発されたプログラミング言語で、次のような方針に基づいて設計された。

(1) 本格的なストラクチャード・プログラミング機能をサポートし、構造化・階層化されたプログラムの作成を可能にする。このようなプログラムの開発においては、トップダウン手法だけでなく、ボトムアップ手法も使用できるようにする。ストラクチャード・プログラミングでは、通常、トップダウン手法が使用されるが、標準化パッケージや問題向言語定義パッケージなどの開発では、ボトムアップ手法が使用されていると解釈することができる。これらの手法は、どちらも、プログラムの階層化に基づいており、統一的な言語体系によって達成することができる。

(2) 自然語風でわかりやすいプログラム記述を可能にする。従来のプログラミング言語は計算機向きにできており、設計書とプログラムとのギャップが非常に大きい。このため、設計書からプログラムへの翻訳過程において、誤りが混入しやすく、でき上ったプログラムのドキュメント性にも問題がある。階層化されたプログラムにおいては、各階層は抽象化された概念を使用して記述されるので、自然語風でわかりやすいプログラム記述が特に重要となる。また、自然語風な記述は、問題向言語の開発においても必須のものである。

(3) コンパイル時における、オブジェクト・プログラムの編集や最適化を可能にする。階層化されたプログラムでは、比較的小さな手続が数多く宣言され、そのリンケージ・オーバヘッドが無視できなくなる。このため、手続のインライン展開機能が必須になる。また、インライン展開の際に、プログラムの実行環境情報に従って、処理ルーチンの選択や不要コードの削除などの処理を行なうことによって、汎用的に作成した標準化パッケージを専用化して使用することができる。

ＳＰＬのように、プログラムの階層化に基づくプログラミング言語を設計しようとする場合には、ソフトウェアの設計手法に対する深い考察が必要になる。すなわち、プログラミング言語は、あくまでも、設計結果を表現するものであるから、階層化されたプログラムはどのようにして設計すべきかという設計手法が明確になっていなければ意味がない。そこで、以下では、まず設計手法に対する筆者達の考え方を述べ、それから、この考え方とＳＰＬにおけるモジュール概念との関連を中心に、ＳＰＬの設計思想を述べることにする。

２．トップダウン設計手法

制御用を始めとするリアルタイム用アプリケーション・ソフトウェアは、通常数十個のタスクから構成され、各タスクは数個のモジュールを含んでいる。しかも、これらのタスクは、互いに深い関連をもっており、共通データを介して情報を交換しながら非同期に動作する。このように複雑なソフトウェアの開発において、設計が如何に重要であるかは言うまでもない。従来、プログラム・エラーの大部分は、設計ミスによって起っている。

設計が重要であることはわかっていても、実際にうまく設計するのはなかなか難しい。それは、設計というものが高度に知的な作業であって、多くの経験と深い洞察力を必要とするからであろう。また、設計者によって、やり方も違ってくる。そこで、ここでは、設計によってでき上るプログラムのモジュール構造はどんなものであって、それをどう表現すれば良いかということを考えてみる。モジュール構造を表現する方法の中で、最も親しみのあるものは、図１に示すような機能階層図であろう。これは機能分割の結果を表現したもので、子モジュールの機能は親モジュールの機能の部分機能であることを示している。機能階層図は、プログラムを構成するモジュールの全体における位置付けを把握するのには便利

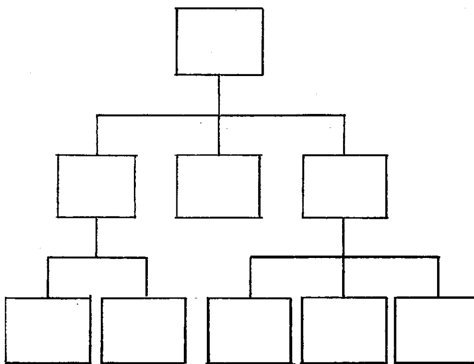

図１．機能階層図

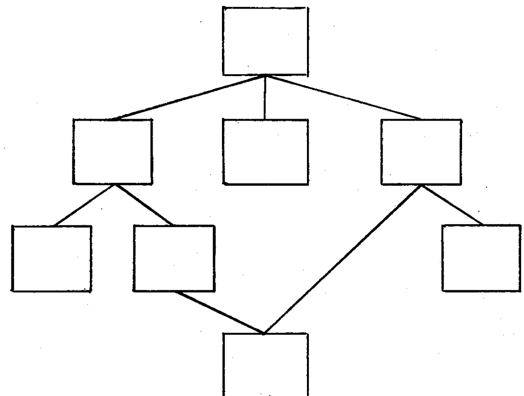

図２．制御関連図

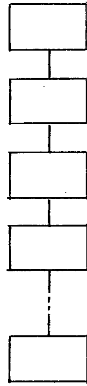

図3．Dijkstraのモジュール構造

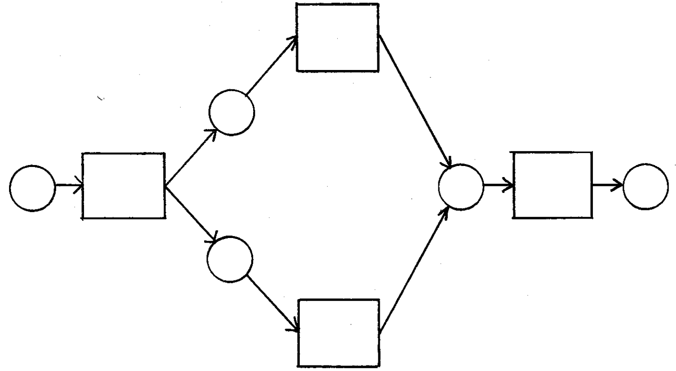

図4．データ関連図

であるが、これだけでは、モジュール間の関連を表現することができない。このため、図2に示すような制御関連図がよく使用される。これはモジュール間の制御関係を表現したもので、上位モジュールは（線で結ばれた）下位モジュールを制御することを示している。機能階層図と制御関連図は全く違う情報を表現したものであるが、この2つはよく混同される。何故混同されるかというと、モジュール間の制御がサブルーチン・コールになっている場合には、制御関係が機能階層図の木構造と重なってしまうことがあるからである。このため、モジュール間構造は木構造になるとか、ならないとかいう議論が起ってくる。機能階層図は、必ず木構造になるが、制御関連図は、一般にグラフ構造になる。制御関連図が木構造にならないのは、次のような場合である。

(1) 複数のモジュールで使用される共通モジュールがある場合。
(2) 再帰的に使用されるモジュールがある場合。
(3) 入口になるモジュールが複数個ある場合。

Dijkstra [10] がストラクチャード・プログラミングを提案したときに、図3に示すようなモジュール構造を主張したことは有名である。これは"真珠の首飾り"と呼ばれているもので、段階的に抽象マシンを設定することによってモジュール化を行なった場合の結果を表現したものである。抽象マシンというのは共通モジュール的な性格をもっているので、このモジュール構造図は制御関連図の特殊な場合であると考えることができる。制御関連図は、モジュール間の関連を把握するのに有効なものであるが、モジュール間共通データとの関連を表現することができない。このため、図4に示すようなデータ関連図が使用される。これはモジュール間共通データに対するモジュールのアクセス関係を表現するもので、各モジュールは矢印の元のデータから情報を入力し、矢印の先のデータに情報を出力することを示している。

さて、以上述べたように、モジュール構造の表現法はいろいろあるが、どれもモジュール構造を一面から見たものであり、互いの関係が余り明確でなく、混乱が生じていたように思われる。特に、機能階層図と、制御関連図およびデータ関連図との相違を明確に認識する必要があろう。このためには、機能分割の段階において、モジュール間の関連をどうとらえているのか、あるいはとらえるべきかということを分析しなければならない。

設計の初期段階では、与えられた目標機能から出発して、それをいくつかの部分機能に分割するという作業を繰り返すことによって、モジュール構造の土台となる機能階層構造を決定する。このとき、1つの機能をいくつかの部分機能に分割するという時点で、部分機能を実現するモジュール群にモジュール間の関連が発生する。従って、機能階層図は垂直方向のモジュール構造を表現しており、制御関連図やデータ関連図は水平方向のモジュール構造を表現しているものと考えることができる。このような構造を一括して表現するのには、2次元的な図が最も適しているが、機能階層構造が木構造になることを利用して、図5に示すようなモジュール階層関連図によって表現することもできる。ここでは、機能階層構造を包含関係によって示し、制御関連構造およびデータ関連構造を、それぞれ、⇝および⟶によって示している。このモジュール構造を機能階層図で示したものが図6である。ここで注目すべき点は、(1) Bのようなメイン・モジュールがAの子モジュールになっていること、(2) Eのような共通モジュールがやはりAの子モジュールになっていること、である。(1)は一般にメイン・モジュールが1個だけ存在するとは限らないからであり、また(2)はEがAの子モジュール間で使用される共通モジュールであるからである。機能分割の過程において最も難しいことは、このEをAの子モジュールとして抽出する所である。何故これが難しいかというと、Eの機能は、CやDをさらに分割してみてからでないと、直接必要にならないからである。このため、共通モジュールをこの段階であらかじめ抽出するには、多くの経験と深い洞察力が必要になる。図3で示したDijkstraのモジュール構造は、この共通モジュールの抽出を徹底的に行なった場合に相当するものであるから、なかなか凡人には真似ができないのも当然である。

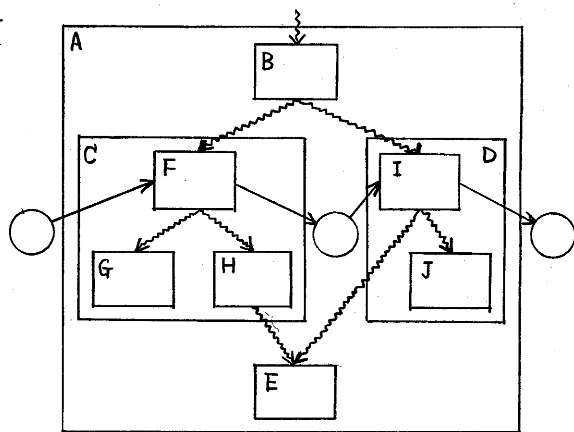

図5. モジュール階層関連図

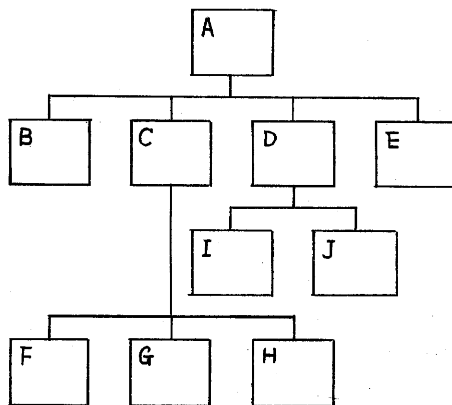

図6. 機能階層図

ここで、トップダウン設計という言葉の意味を説明しておく。これは、制御関連図における上位のモジュールから下位のモジュールへ設計を進めてゆくという意味で使用されることが多かったが、ここでは、そうではなくて、機能階層図における上位のモジュールから下位のモジュールへ設計を進めてゆくという意味で使用する。この意味では、同一階層にあるモジュールは、同時に設計するということになる。ただし、開発段階では、メイン・モジュールの方からやるトップダウン開発でも、共通モジュールの方からやるボトムアップ開発でも構わない。

トップダウン設計とはいっても、もちろん、上位から下位へ直線的に設計が進むという訳ではなく、何度かの試行錯誤を経て、モジュール構造が決定される。しかし、上位モジュールの機能を達成するために下位モジュールが設定されるという意味において、設計は、原則として、トップダウンに行なわれるものであると言うことができる。これを以下に示す。

〔トップダウン設計の原則的手順〕

<u>ステップ1</u>．目標機能とそれを実現するルート・モジュールを設定する。

<u>ステップ2</u>．その機能を実現するために提供しなければならない入口リソース（データや手続など）の集合と、その機能を実現するときに使用しても良い外部リソースの集合を設定する。

<u>ステップ3</u>．その機能を実現するモジュールが機能階層図における端末モジュールなら設計を終了する。そうでない場合には次のステップに進む。

<u>ステップ4</u>．その機能をいくつかの部分機能に分割し、各部分機能を実現する子モジュールと、子モジュール間の共通データを設定する。

<u>ステップ5</u>．これらの間の制御関連構造やデータ関連構造を明確にし、各子モジュールに対して、ステップ2からステップ5までを繰り返す。

<u>3．プログラム構造</u>

既に述べたように、プログラミング言語は、設計結果をなるべくそのままの形で表現できるようになっていることが望ましい。従来のプログラミング言語は、この点でははなはだ不十分である。何故ならば、そこではコンパイル単位内の構造のみと考えているため、設計において最も重要なモジュール構造を殆んど表現することができないからである。これに対して、ＳＰＬは、前節で説明したトップダウン設計手法に基づいてプログラム構造が設計されているので、設計結果を自然に表現できるようになっている。すなわち、機能階層図における非端末モジュールと端末モジュールに対応するものとして、ＳＰＬは環境モジュールと処理モジュールという２種類のモジュールを持っている。

環境モジュールは、主にモジュール間の共通データ（定数、変数、データ型）を宣言するもので、手続は宣言できない。共通データの宣言は、宣言ユニットによってグループ化される。宣言ユニットは、データ関連図におけるデータに対応するものと考えられる。リアルタイム用アプリケーション・ソフトウェアでは、モジュール間（あるいはタスク間）の共通データが比較的多く使用され、重要な役割を果す。信頼性の面から見れば、モジュール間の共通データは許さず、全てパラメタとして渡すか、共通手続群の中にカプセル化することが望ましいが、これだけでは、記述が煩わしくなる場合が多く、現実的でない。

処理モジュールは、モジュール内のデータと手続を宣言するものである。１つの処理モジュールの中で、複数個の手続を宣言することができる。この機能を利用して、データの抽象化を実現することができる。

従来のプログラミング言語では、モジュール間の共通データは個々のモジュールで宣言しなければならない。本来、１個のデータに対して１個の宣言が必要であると考えるのが自然であり、複数個の宣言はインターフェイスの不一致を招く原因となる。ＳＰＬにおけるモジュール間共通データは、環境モジュールで一括して宣言され、その子孫モジュールで宣言なしに使用される。これを実現するために、ＳＰＬコンパイラは、上位モジュールのコンパイル時に必要情報をライブラリに登録しておき、下位モジュールのコンパイル時にこれを参照する。

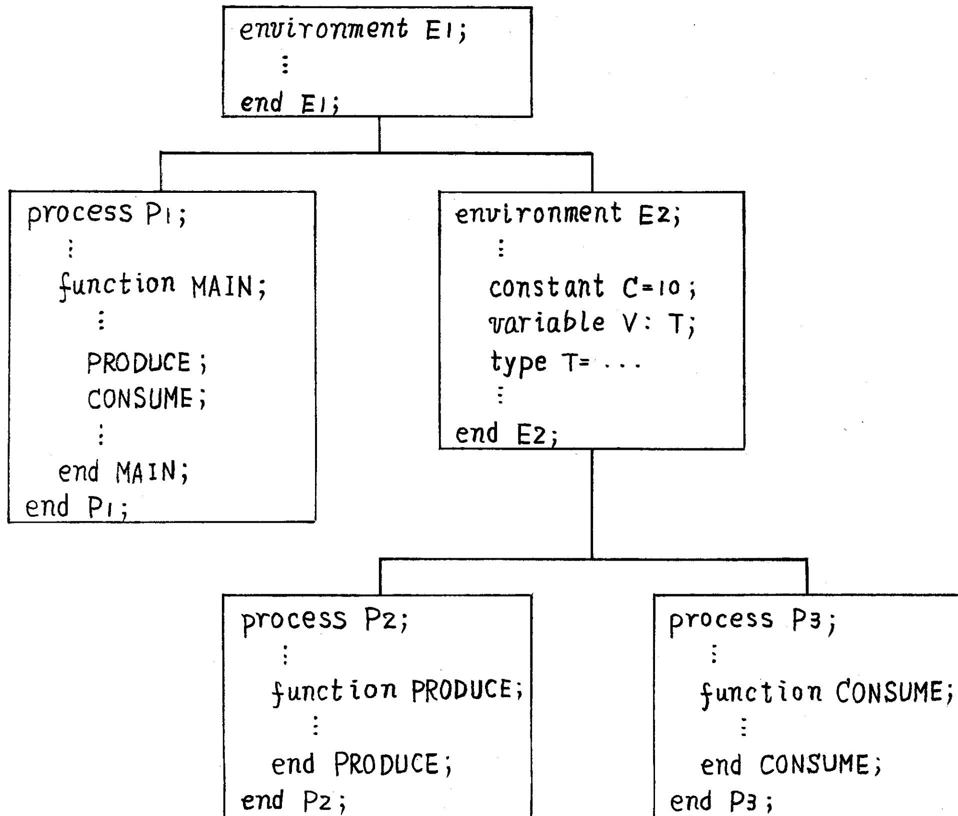

図7．ＳＰＬのプログラム構造

図７にＳＰＬのプログラム構造の例を示す。この場合、P_1 がメイン・モジュールで、P_2 と P_3 がサブ・モジュールである。サブ・モジュール間でのみ使用される共通データは E_2 で宣言される。

４．データ型宣言

最近、データ型宣言機能の重要性が認識されてきている。データ型とは何か、ということについて多くの議論があるが、現在は、データ構造の表現とそれにアクセスするオペレータの集合によって定義されるという考え方が一般的である。しかし、プログラマがデータ型を宣言する場合、次の２つの使用法がある。

(1) ある（複雑な）データ型に別名をつけて、参照する際の記述の重複を防ぐ。ＰＡＳＣＡＬのデータ型宣言はこの種のものであり、信頼性の向上やコーディング時間の短縮に有効である。この場合には、記述を簡略化するだけであって、コンパイラによるデータ型のチェックには関与しない。

(2) ある（複雑な）データ型に、全く新しいデータ型としての名前をつけて、外部からその詳細情報を隠してしまう。ＣＬＵ[2]のクラスタなどはこの種のものであり、詳細情報の保護や変更容易性の向上に有効である。また、階層化されたプログラムにおいて、各階層の記述レベルをそろえるのにも有効である。この場合には、コンパイラによって、外部で詳細情報が使用されていないかどうかがチェックされる。

ＳＰＬでは、この２つの使用法を同一のメカニズムでサポートすることを考え

た。そのために、データ型宣言は単にデータ構造の表現を与えるものとし、オペレータの集合との結合は、前節で説明したモジュール概念によって実現するものとした。また、詳細情報が使用できるか否かは、モジュール構造によって規定されるスコープに従うものとした。このようなアプローチによって、データ型と手続を対称的に取り扱うことが可能になった。図8にデータ型宣言の2つの使用法の例を示す。この場合、M_1では、T_1の詳細情報を使用することはできるが、T_2の詳細情報を使用することはできない。従って、T_1は記述の簡略化のために使用されており、T_2はプログラムの階層化のために使用されていると言うことができる。

ところで、ＳＰＬにおける変数は、全てスタティックであるため、変数の詳細なデータ型（少なくともその大きさ）はコンパイル時に確定していなければならない。たとえば、図8の例で言うと、V_2にメモリを割り付けるには、T_2の詳細情報が必要になる。従って、M_1をコンパイルする前に、M_2をコンパイルしなければならないという制限がつくことになる。変数が動的なメモリ属性（オートマティックやコントロールド）を持つ場合には、この制限を回避することができる。ＥＵＣＬＩＤ[4]のモジュールやＣＬＵのクラスタがその例である。

モジュール M1

```
type T1=struct(U1: INTEGER,
               U2: INTEGER);
function F1;
  variable V1: T1, V2: T2;
  ...
  F2(V1.U1,V2);
  ...
end F1;
```

モジュール M2

```
type T2=struct(U1: INTEGER,
               U2: INTEGER);
function F2(X: INTEGER,Y: T2);
  ...
  X = Y.U2;
  ...
end F2;
```

図8．データ型宣言の2つの使用法

```
type QUEUE OF CHAR(P: INTEGER)
       WITH LENGTH(L: INTEGER)
  =struct(TOP: INTEGER init(0),
          BOT: INTEGER init(0),
          LEN: INTEGER init(L),
          INF(L): CHAR(P));
   :
variable V: QUEUE OF CHAR(8)
              WITH LENGTH(32);
```

図9．パラメタ付データ型宣言

データ型の一般化や標準化を可能にするために、データ型宣言はパラメタを許すことが望ましい。パラメタのレベルとしては、次の３つが考えられる。

(1) 定数

データ型には、配列体の上下限、精度や長さ、初期値などの定数が現れる。これらの定数部分のみが異っているデータ型を、いちいち別のデータ型として宣言するのは大変煩わしい。このため、定数パラメタを許したいという要求がでてくる。

(2) 変数（あるいは一般式）

動的配列体などを考えると、さらに変数（あるいは一般式）パラメタを許したいという要求がでてくる。

(3) データ型

スタックやキューなどのように、データの構造を抽象化する場合には、要素のデータ型が何であっても、それとは（表面上）無関係に、アクセスするオペレータの集合を定義することができる。このため、データ型パラメタを許

したいという要求がでてくる。
　SPLは、この中で(1)のみを許している。その理由は次の通りである。既に述べたように、SPLにおける変数は全てスタティックである。このため、実行時にしかデータの大きさがわからない(2)は、一般に、許すことができない。ただし、整合寸法の場合には、メモリを割り付ける必要がないので許している。また(3)は、オペレータの仮パラメタのデータ型が実行時まで決まらないため、オブジェクト効率が低下し、実用的でない。
　データ型の参照形式は、次節で述べる手続の場合と同様に、自然語風でわかりやすいものになっている。パラメタ付きデータ型宣言の例を図9に示す。

5．手続宣言

　SPLの手続は、従来のサブルーチン、関数、マクロ等を統一したもので、細かな差異は、オプションによって指定する。オプションには、MAIN、SUB、CLOSED、OPENがある。MAINとSUBは外部ルーチンに、CLOSEDとOPENは、それぞれ、内部閉サブルーチンと内部開サブルーチンに変換される。プログラマは、メモリ効率と実行効率のどちらを重視するかに従って、これらのオプションを選択することができる。
　手続の参照形式は、プログラムのドキュメント性に大きな影響を与える。そこで、SPLでは、次のような形式を許している。

(1) 従来形式　　　　PUSH(E,S)
(2) コメント形式　　PUSH　DOWN　(E)　TO　(S)
(3) キーワード形式　PUSH　(ELEMENT=E,STACK=S)

コメント形式は、識別子の並びと、パラメタの位置によって認識される。このため、最初の識別子が同じ手続を複数個宣言することができる。この機能は、問題向言語の開発に有効である。コメント形式の利点は、実パラメタを省略したり、その順序を変えたりすることができることである。このため、プログラマは、手続の参照側を全く変えることなしに、手続のパラメタを追加することができる。この機能は、標準化パッケージの機能拡張に有効である。
　手続の参照形式は、そのオプションに依存しない。従って、手続の参照側を全く変えることなしに、オプションを変更することができる。これは、プログラムの変更容易性にとって重要な性質である。

6．コンパイル時機能

　コンパイル時機能は、柔軟性の高い標準化パッケージを作成するのに有効である。従来のプログラミング言語にも、ある種のコンパイル時機能をもつものがいくつかある。たとえば、アセンブリ言語の可変マクロ機能やPL／Iのコンパイル時機能などである。これらのコンパイル時機能は非常に強力ではあるが、信頼性の面から見ると、次のような欠点がある。

(1) コンパイル時制御文が構造化されていない。すなわち、オブジェクト・プログラムを生成するのに、コンパイル時GOTO文が使用される。このため、ソース・プログラムとオブジェクト・プログラムとの対応が非常に複雑になる。従って、コンパイル時機能を使用したプログラムのデバッグは非常に難しい。
(2) これらのコンパイル時機能は、いわゆる、テキスト・マクロを基本としている。このため、基本言語の構造、特にスコープが無視され、思わぬエラーを誘発する。

(3) コンパイル時の選択や置換の対象は、任意のテキスト片であり、必ずしも構文要素として閉じたものになっていない。このため、コンパイラは、展開する前に(すなわち、コンパイル時文を解釈実行する前に)、ソース・プログラムの文法的な正しさをチェックすることができない。

このような欠点は、コンパイル時機能をソース・プログラムの前処理機能として使用する場合には、それ程問題にはならないが、ソース・プログラムのドキュメント性を重視する場合には、致命的である。そこで、SPLでは、コンパイル時制御文を実行時制御文のサブ・セットとして構造化すると共に、テキスト・マクロに代ってシンタックス・マクロを基本とすることによって、上記の問題点を解決した。図10にコンパイル時機能の例を示す。このプログラムは、STEPの値が0に等しいか否かに従って、S_1 か S_2 を生成する。ここで%はコンパイル時機能であることを示す記号である。選択の対象となるテキストは、S_1 や S_2 のように、文あるいは文の並びでなければならない。従って、たとえば、"for I=1, 100 repeat" や "end" のような文の一部を選択することは許されない。この制限によって、生成されるオブジェクト・プログラムが必ず文法上正しいものになることが保証される。

```
function INITIALIZE
            (A(100): INTEGER,
             STEP: INTEGER)
              options(OPEN);
  variable I: INTEGER;
 %if STEP=0
   then  ┌─────────────────────┐
         │for I=1,100 repeat   │ S1
         │  A(I)=0;            │
         │end;                 │
         └─────────────────────┘
   else  ┌─────────────────────┐
         │A(1)=0;              │ S2
         │for I=2,100 repeat   │
         │  A(I)=A(I-1)+STEP;  │
         │end;                 │
         └─────────────────────┘
  end;
end INITIALIZE;
```

図10. コンパイル時機能

7. おわりに

以上、トップダウン設計手法とSPLにおけるモジュール概念との関連を中心に、SPLの設計思想を述べた。今までに、ストラクチャード・プログラミング言語がいくつか発表されているが、これらと比較して、SPLは次のような特徴をもっている。

(1) データと手順の抽象化機能が対称性をもっている。
(2) データ型と手続の参照形式が自然語風でわかりやすい。
(3) 手続のイン・ライン展開ができる。
(4) オブジェクト・プログラムの編集や最適化ができる。
(5) データ型宣言は、記述の簡略化とプログラムの階層化という2つの目的に使用することができる。
(6) 処理モジュール内でデータ型と手続をカプセル化することによって、データの抽象化を実現することができる。
(7) 環境モジュール内でモジュール間の共通データを一括して宣言し、子孫モジュール内でこれを自由に使用することができる。
(8) これらのモジュール概念によって、トップダウン設計手法に適したプログラム構造を実現することができる。

最後に、本研究にご援助頂いた、日立製作所大みか工場の定間豊副工場長、高井兵庫副技師長、ならびに技術的な討論をして頂いた、同工場の篠本学氏、森清三氏、日立研究所の高藤政雄氏、HECの加藤木和夫氏に感謝の意を表します。

参考文献

[1] Leavenworth, B.M., Control Structures in Programming Languages - the GOTO Controversy, SIGPLAN Notices 7, 11 (1972), 53-91.

[2] Liskov, B. and Zilles, S., Programming with Abstract Data Types, Proceedings of SIGPLAN Symposium on Very High Level Languages, Santa Monica, (1974), 50-59.

[3] Wulf, W. A., Alphard : Toward a Language to Support Structured Programs, Dept. of Computer Science Internal Report, Carnegia-Mellon University, Pittsburgh, April, (1974).

[4] Lampson, B.W. et al., Report on the Programming Language Euclid, SIGPLAN Notices 12, 2 (1977).

[5] Meyers, G.J., Reliable Software through Composite Design, (高信頼性ソフトウェア ── 複合設計), 近代科学社, (1976).

[6] Bridge, R.F. and Thompson, E.W., BRIDGES ── A Tool for Increasing the Reliability of References to FORTRAN Variables, SIGPLAN Notices 11, 9 (1976).

[7] DeRemer, F and Kron, H., Programming-in-the-Large versus Programming-in-the-Small, Proceedings of the International Conference on Reliable Software, SIGPLAN Notices 10, 6 (1975), 114-121.

[8] Teichroew, D. and Hershey, E.A., PSL/PSA : A Computer-aided technique for Structured Documentation and Analysis of Information Processing Systems, Proceedings 2nd International Conference on Sofware Engineering, October, (1976).

[9] Bell, T.E., Bixler, D.C. and Dyer, M.E., An Extendable Approach to Computer-aided Software Requirement Engineering, Proceedings 2nd International Conference on Software Engineering, October, (1976).

[10] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., Structured Programming, Academic Press, New York, (1972).

[11] 林, 野木, 中所, 浜田 他, 制御用ストラクチャード・プログラミング言語ＳＰＬの開発思想 他, 情報処理学会第１７回全国大会講演論文集, No. 1～4, 昭和５１年１１月.

[12] 林, 高井, 制御用ソフトウェアへのストラクチャード・プログラミングの応用とその問題点, 昭和５２年電気学会全国大会, 昭和５２年７月.