解　説

ストラクチャード・プログラミング用言語*

筧　　捷　彦**

1. ストラクチャード・プログラミングと言語

1.1 問題の背景

E. W. Dijkstra によって提唱されたストラクチャード・プログラミング[1] (structured programing, 以下 SP と略記する) が，計算機関係者の間で，中心話題となったのには，それなりの理由があったといえる．

●ソフトウェアが巨大化し，その“危機”という言葉が語られる程の状況にあった（——おそらくは，今なおその状況にある）こと．

●「外界の問題解決」の手段としてソフトウェアが作成される．このため，その問題および解決した結果に重点がおかれ，プログラミングは，ある意味で，できさえすれば，という形でのみ取り扱われてきたこと．

●プログラミング関係者の間では，それぞれに，“うまく”プログラミングするための工夫なり，手段なりが，実際的な意味で作り出されていた．ただ，多くの場合，「外界の問題」ほど語られずにいたこと．

●そこへ，Dijkstra が，プログラミング自身を1つの問題として明確に提起し，解決の方針として SP を打ち出したことで，これらの議論が一度に表に出てきた．SP 旋風の生じた理由は，こうしたことにあったと筆者は考えている．

プログラミング自身を正当な問題として捉え，その解決方法をさぐり出すこと．これこそが中心の課題であり，SP はそのために提唱された1つの方針にすぎない．他にも有効な方針なり手段なりが，今後提唱されることもあるであろう．課題が解決を得たわけではないのである．

1.2 SP の捉え方

以下の議論を進めるにあたって，ここでの SP の捉え方を明確にしておこう．これが，かならずしも，Dijkstra のいう SP，あるいは，ふつうに語られている SP とは一致しないかもしれないことを，予め断っておく．

●　プログラミング

プログラミングとは，与えられた問題に対し，その解決するアルゴリズムを，最終的には，与えられた言語の上で記述することである．

問題が記述される言語や概念の世界と，最終的にアルゴリズムを記述する言語（およびその付随する概念）の世界との間にはふつう大きな隔りがある．考えられるアルゴリズムは，多くの場合，前者の世界の中か，あるいはそれに近い概念の世界の中で考え出される．これを，一度に後者の世界に持ちこんではならない．

●　S P

最初のアルゴリズムを記述する．そこでの概念を，最終目的とする言語（概念）の世界に近づいた概念の世界の中で記述し直す．こうした手続きの繰返しとしてプログラミングすることを SP という．

“近づいた”という言葉には，あいまい性がある．実際のところ，どれが“近づいた”ものであるのかは，最初からはわからないのがふつうである．したがって SP には，逆もどりも生じる．つまり，どこかの段階までもどって，それ以降に行った分を全部やりなおすことが生じうる．

どれ程“近づく”のがよいか，という問題もある．Dijkstra は，「一時に多くを決めるな」[1] という形で1つの目安を与えている．

上の定義に出てくる以外のものは，SP のための手法ないしは，SP の帰結として得られるものと考えておくことにする．たとえば，SP と同一に語られることの多いものを取り上げてみよう．

●　降下型プログラミング

降下型 (top-down) というのは，SP に比較的よく合致したプログラミングの方法だといえる．しかし，

---

* Languages for structured programming by Katsuhiko KAKEHI (Rikkyo University, Faculty of Science, Department of Mathematics)

** 立教大学理学部数学科

下位の概念の世界が，先に構成されたとしても，SPであることにはかわりはない．ハードウェアに近いところから始めて，多くの概念世界を順々に構成して作りあげたという，THE システム[2]も，SP によるものと考える．

● **go to**-less プログラミング

SP の各段階は，それぞれ，ある（上位の）概念で記述されていることを，より下位の概念を用いて，アルゴリズムとして記述することである．このとき，アルゴリズムを構成するのに用いられる制御の流れに関する部分を，よくまとまった形のものにしておくことが望まれる．つまり，制御の流れに関する高位の概念世界を用意することになる．もし，この概念が最終の言語にも揃えられているものならば，帰結として **go to** は使われずに済む．これが，**go to**-less の所以であって，**go to** を用いないことがそのまま SP であるというのも，**go to** を使うと SP でないというのも，ともにここでの SP の捉え方からはずれる．

1.3 SP と言語

SP を，何段階にもわたる概念規定の書き直しとみるとき，それに適した言語は何か，という問題が生じる．ひとつには，自然言語を用いるという方法がある．人間が新しい概念を作り出していったとしても，それが語られるのは，自然言語の中であるから，まさにうってつけの言語だといえる．実際，Dijkstra が SP をやってみせている場合でも[1]，たくみに自然言語を（あたかも，新しいプログラミング言語があって，その中では自然言語も自由に使いうるかのように見せながら）用いていることに留意しよう．

ただ，自然言語でよい，としてしまったのでは以後の話しのしようがないから，もう少し状況を明確に設定することにしよう．

● プログラミング言語

最終的には与えられた計算機の機械語でアルゴリズムを記述しなければならない．ただ，プログラミング言語が設計され，そのコンパイラが作られている状況下では，与えられたプログラミング言語を最終のものとみなして SP を進めてもよいであろう．

結局，これは，プログラミングに適した言語を開発する，という古くからの問題に帰着する．ただ，主眼が，それを用いてプログラミングする過程にあることに着目しよう．ハードウェアの技術があり，それによって作られた計算機があり，その計算機に基づいて，「人間向き言語」を設計するという方向ではなしに，プログラミングの方から，言語が定まり，それに従って計算機を作る，という方向に進むことになる．実際，Dijkstra の講演[3]からも，それを指向している様が読み取れる．

● 計算機の手助け

SP は，最終の出来上りのプログラムだけを考えているわけでなく，（途中で出てきた）上位の概念で記述されたものも，正当に取り扱おうとしている．プログラミングの過程で逆もどりが生じうるから，という理由だけでなく，こうした記述を含めて SP があり，最終のプログラムも，これらとともに理解されるべきものであるからである．

SP という一連の作業に，計算機を利用することも重要な問題として考えられるべきである．ただ，本論では，この点にあまり触れないことにする．

2. 制御の流れの取扱い

go to 論争[4]をひきおこした，制御の流れの問題を取り上げてみよう．

Dijkstra がその投稿[5]で述べているように，動的な実行という過程を，プログラムされた静的な文字列の中にあらわすためには，動的な過程を把握する概念をうまく構築しなければならない．Dijkstra の提唱は，

（1） 遂次的な動作の列
（2） 一連の動作の繰返し
（3） 2つの一連動作の選択
（4） 別の一連動作の代行

の4つを基本概念とする．これらは，これまでのプログラミング言語にも多くの場合取り入れられていた概念である（実際 ALGOL 60 をとってみても，これらを基本としている）．以下，個々に取り上げて，考えてみることにする．

2.1 逐次的動作の列

1つの動作の完了をまって，次の動作が始まる，という形のものは，アルゴリズム記述の上で基本的なものである．現在の計算機の多くは，この形で機械語が設計されている．このことから，これまでのプログラミング言語の多くは，自然にこの機能を備えている．

ただ，ある一連の動作を，1つの動作と見なすということはしばしば生じるにもかかわらず，これに相応する機能はかならずしも，プログラミング言語には，はいっていなかった．ALGOL 系の言語と呼ばれるものは，いわゆる複合文とかブロックとかいった形で実現している．

2.2 一連の動作の繰返し

繰返しは，計算機で仕事をする場合の中心的な作業である．ただ，まったく同一の動作が繰り返されることはなく，毎回少しずつ変化があるのが普通である．これらの変化は，変数とか配列の中身とかの変化として記述され，制御の流れの方は，同一のものの繰返しとして捉えられている．

これまでのプログラミング言語にも，この繰返しの機能は組み込まれていた．ただ，特定の変数の中身にだけ着目する——しかも多くは限定された変化のし方しか許されていない——ものだけが組み込まれていた．これは，必ずしもプログラミング上の必要をすべて自然な形で満たしてくれる，というわけにはいかないものである．Dijkstra は，

while α **do** β

repeat β_1; β_2; …; β_n **until** α

の形のものを採用するように唱え，これまでの

for $v:=e_1$ **step** e_2 **until** e_3 **do** β

の形のものより，自由度を高めるように提唱している[5]．

繰返しの部分については，特に議論百出の感がある．その理由は，繰返し，ないしはループの本質に根ざしているようである．繰返しには，それに先立つ準備および後仕末が付随するものである．ふつう，上位の概念においては，これらをもこみにして1つの要素として取り扱われるにもかかわらず，たとえば，上にあげた例などは，その1部のみしか表現していない．また，これらの準備や後仕末を含めて考えると，繰返しは，2次元的なものになるにもかかわらず，プログラミング言語は1次元的でありうまく整合しない．さらに，**while** や **repeat** の場合，判定部分は，(論理)式に限定されているが，かならずしも条件がこうした式という静的なもので表現できないことも，反論を呼ぶ理由である．

こうした点に関して，繰返しの構造をより強力にしようとする試みもあるようである[6]．いずれにしても，構造を1つ定めれば，それでは自然に書けない例が見出される状況では，これが解決案という線におちつくには，時間がかかるであろう．

2.3 動作の選択

判定して，いくつかの候補である動作の1つを選択することもプログラミングには基本的である．これまでに採用されてきたものは，候補が1つ，2つと限定された形のものである．

if α **then** β

if α **then** β_1 **else** β_2

さらに，これを n まで拡張したものも提唱されている．例えば，PASCAL[7] の

case e **of** α_1: β_1;
α_2: β_2;
⋮
α_n: β_n

end

や，BLISS[8] の

select e **of nset** α_1: β_1;
α_2: β_2;
…
α_n: β_n

tesn

のように，e の値が，広い意味で条件 α_i をみたすとき，β_i を選択するというものがある．

いっそ，α_i を任意の条件式としてしまい，LISP 1.5[9] のように，

cond $\alpha_1\rightarrow\beta_1$; $\alpha_2\rightarrow\beta_2$; …; $\alpha_n\rightarrow\beta_n$ **end**

という形のものを用意すれば，一挙に済んでしまいそうである．ただ，歴史的な影響（プログラミング言語とはいえ，言葉であるから，最初に慣れ親しんだものの影響は大きい）から，**if-then-else** 風のものまでは併合できそうにはない．また，BLISS や PASCAL での構造で注意すべき点は，着目しているもの(e)を明確に抜き出して書かせていることであろう．つまりこうした構造では，制御の流れの変化をひきおこす原因となるものにかなりの主眼がおかれている．このことは，先に述べた繰返しにも言えることで，**while** や **repeat** であったとしても，なお **for** の形のものが愛用される（つまり，ピッタリとするものがある）ことと関係している．

2.4 一連動作の代行

いわゆるサブルーチンに相当するものである．サブルーチンという概念もまた，プログラミングの歴史の初期から重要な役割を果たしてきたので，プログラミング言語の多くが含有している．

制御の流れについていえば，再帰的なサブルーチンの呼出しを認めるか否かが1つの分岐点となるであろう．再帰的呼出しは，一般にそれが高価な——現在の計算機上で実現されるときしばしば効率の悪い事態に立ちいたる——ため，悪く評価されがちである．プログラミング上再帰的呼出しがあることが，問題のすじ道を明らかにしてくれる場面があることも事実である．

現在の計算機を用い，しかもなお効率の悪いオブジェクトしか作り出されないとしたら，そして，“効率の良”いオブジェクト・コードが望まれるなら，再帰的呼出しを用いて書いた動作を，そうでないものにさらに書直しする．これもまたSPの内（効率よいオブジェクトを作り出す，という外的条件の下では）である．

SPは，繰り返しての書き直しが進んでゆく．この過程が——およびその記録が残されることが——重要であることは述べた．現在のプログラミング言語では，こうした過程は，すべて，プログラマが勝手にやることになっていて，出来上ったプログラムに，その痕跡を残すことは難しい．唯一ともいえる方法は，サブルーチンの形で残すことであるが，ここでもまた“効率の良さ”という例の問題が頭をもたげてくる．

その意味から，マクロの様な機能が備えられていることが望ましい．つまり，記述は，別に行われるが最終的にオブジェクト・コードになったときは，埋め込まれたものとなるような，記述単位がほしい．ただ，SPでの書換えが進行するあいだ，先にある（上位概念での）記述が一定のままとは限らないから，こうした問題も，SPを助けるソフトウェア・システムの方の問題と考える方がよいのかもしれない．

2.5 その他の制御の流れ

以上にみたものの他にも，制御の流れに関する構造も要求される場面がある．つまり，もともと互いに連絡しあって動作する2つの別の制御の流れがある，として問題を捉えた方がよい場面もある．こうした概念としてコルーチン（coroutine）がある．

同様に，もっと多くの制御の流れを想定するのが自然なこともあるであろう．いわゆる並行処理（parallel processing）であるが，範囲を広げすぎるので，この問題は，これ以上触れないことにする．

2.1～2.4で述べた部分についても，まだ最終的なものが得られているとはいえない．今後，いろいろな構造が提唱され，実験されていくであろう．そうした実験の道具としては，拡張性をもった言語（extensible language）も有用であろう．

3. データ構造の取扱い

go to 論争にからめて，制御の流れの構造についてはSPとの関連が多く語られているが，それに比べてデータ構造が，あまり語られることがないのはどうしたことであろうか．

Dijkstraのプログラミング例[1]をみても，データに関しては，整数値と，論理値と，それらの配列程度を使ってみせているだけで，それ以上のことは述べられていない．SPの中心が，巨大化した基本ソフトウェアにあることを考えると，これは不思議なことである．プログラミング一般において考えられる種々のデータ構造は，基本ソフトウェアには必要でない，ということなのであろうか．いや，事態は逆の方向にあると見るべきであろう．

SPが，与えられた最終的な言語——つまりは現存する計算機の機械語——でのプログラムに至る過程であるとするなら，データ構造の方も，現在の計算機で直接取り扱えるものだけを最終の形とせざるをえない．現在の計算機が，一次元的な番地というものを介して取り扱うメモリの上に構成されている以上，最終的な構造として唯一可能なのは配列だということになる．また，その配列の個々の要素は，整数，実数，文字，ビット列等のみということになる．

だからといって，プログラミング言語が，これらだけをデータの構造として持てばよい，というのは制御の流れの取扱い方と比べて，ひどく片手落ちのものといえる．機械語には **go to** しかないから，という理由だけでプログラミングに **go to** だけおけばよい，とする議論を排除したのと同様に，プログラミングの側からの要請を中心に考えてみる必要があるだろう．

3.1 基本データ

基本データとしては，整数，文字(列)，ビット(列)，場合によって実数といったものが考えられる．他に，いくつかの名前で呼ばれるものの集り，といったデータを考えることができる必要もあるだろう．

たとえば，コンパイラが，その言語の基本要素の類別をしている場面を考えよう．この場合，

(*constant*, *name*, *delimiter*)

といった3種だけの区別が生じるとしよう．これらの類別結果を扱う場面のデータに対しては，この3種のみの区別ができるものであればよい．もちろん，整数を用いて表現することも可能だが，それは，上位の概念を下位の概念で表現し直すという過程を行ったことに他ならない．つまり，SPでの1過程であって，上位概念での記述の場では，こうしたデータを使える必要がある．

3.2 配　列

配列は，前に述べたとおりの事情からも，たいていのプログラミング言語に含有されている．何次元のも

のまでが必要か，という問に対しては，実際面からしか答えることはできない．極端なことを言えば，1次元の配列さえあれば，多次元の添字を1次元に写すことは比較的容易だから，十分とさえいえる．ただ，多次元の配列も，1つの有用な概念だから，これらのことが容易に表現できる工夫（例えばBLISSのstructure）が必要であろう．

3.3 リスト構造，スタック

リスト構造とかスタックといった概念も，基本ソフトウェアでは重要な役割を果たす．

リスト構造は，機械語では，番地をデータとして扱うことで実現される．これに相当する機能を，いわゆるポインタとしてデータに含めた言語も多い．しかし，ある意味で，このポインタは，制御の流れでの **go to** に相当するものであり，このままでよしとするのは，不釣合いである．

スタックもまた，リスト構造の1つと考えることもできる．そのデータの取扱い方の特性から，直接配列を用いて実現されることが多い．

こうしたデータ構造が，一度に，現存の計算機のメモリの構造上に表現されるところまでプログラミングが進行するのは，あまりSPであるとはいえない．したがって，データ構造をうまく表現するための手段を準備しておく必要がある．

必要となるデータ構造は，それこそ問題ごとに大きく異なることがあるから，言語には，自由にデータ構造を設定できる機能がほしい．いわる拡張性をもった言語にはこうしたデータ構造を拡張性を持つものもある．ただ，注意しなければならないのは，データ構造がメモリ上での表現のみで定まるのではなく，それに対応する各種の演算とこみにして存在する点である．このとき，メモリ上での表現の部分は，実現するための方法としてのみ必要な部分であり，そのデータ構造を用いる場面とは切りはなされているべきものであろう．

演算の方は，ふつうサブルーチンの呼出しの形で実現される．これらのサブルーチンは組になって定義されている必要が生じる．そして，これらの組の中だけで，メモリでの実現方法が管理されていればよい．こうしたまとまりに，いわゆるブロック構造は適当ではない．つまり，実現しているメモリを，これらのサブルーチンの組にだけ固有のものにするために，ブロックに入れたとすると，外からは，そのサブルーチンさえも引用できなくなるからである．この点で，SIMURA 67[10] が採用している方法には着目してよいであろう．

4. 言 語 例

SP用言語は，いろいろと開発されつつある状況であるが，ここでは，よく知られた——数年以上を経た——ものを取り上げて，SP用としての適性を検討してみることにする．

Dijkstra 他による，「Structured programming[11]」と題した本が出ている．SPのある意味で主唱者達の力作であるから，この本の中で用いられている言語を取り上げることにする．Dijkstra自身が執筆している部分は，特に言語を明示していていないので，結局，PASCAL と SIMULA 67 を取り上げることにする．他に，SP が，システムプログラムを1つの対象としていることから，システムプログラム用言語の1つとして BLISS を取り上げることにする．

4.1 PASCAL

PASCAL は ALGOL 系の言語であって，N. Wirthを中心として開発されたものである．そのコンパイラも，PASCAL で書かれ，複数の機種上で使われている．

PASCAL はそれまでの ALGOL 系の言語と比べると，実用性——あるいは良いオブジェクト・コードが作れることに相当の重点をおいて設計された．

したがって，設計の時点でどれ程 SP が意識されていたかはわからない．ただ，Wirthの著書[12]でみるように，SP 的なプログラミングへの指向も十分に感じることのできる言語である．

特徴的な点を列挙して，検討してみよう．

■ 制御の流れの構造

ALGOL 60 でのブロックに相当するものとしては，手続き（の本体）だけを考えている．したがって，変数等の宣言は，手続きごとにのみ可能である．**begin end** の組は，式での( )の組と同様の働きを文に対して行なうだけの機能に制限されている．

選択の機能は，**if-then, if-then-else** と，

case e **of** c_1: S_1; c_2: S_2; $\cdots c_n$: S_n **end**

が果たす．c_i は，e と同じデータ型の定数で，e の値と‘等しく’なるものを選択する．

繰返しは，**while-do, repeat-until** および **for-do** がある．ただし，**for-do** は，ALGOL 60 での **step** に相当するものが，± 1 に限られて形になっている．

手続きは，いわゆるサブルーチンに相当するものと関数に相当するものがある．再帰呼出しもできる．

以上の外，**go to** も並存している．ただ，**go to** で飛越しのできる範囲は，原則として，その**go to** を含む手続きの本体である．手続き本体の外へ飛び越すにはそれなりの宣言を必要とする．いずれにせよ，用意された他の構造だけでは，十分でないことが認識されているものと思われる．

■ データ構造

基本のデータとして，整数，実数，論理値，文字をおく．さらに，スカラ型と称して，有限個の名前づけられた値の集合だけを考えるデータを設定できる（整数等は，スカラ型のものと考える）．こうした基本のデータの上に，以下に掲げる操作を繰り返し適用することで，自由にデータ構造を作り出すことができる．

（a） 部分範囲　既存のスカラ型の1部分のみを扱う．

（b） 配列　既存の型を要素とする配列．その添字としては，部分範囲型のもののみを考える．

（c） レコード　いくつかのデータを，それぞれに部位名をつけてひとまとめにする．（c）をくり返し適用すれば，COBOL でのデータ構造風のものを作り出せる．このとき，最後の部位にのみ，可変な要素を入れることが可能である．

（d） 集合　与えられたデータ型の値の部分集合を値として考える．

（e） ファイル　与えられた型を記録の単位とするファイルを作る．

（f） ポインタ　与えられた型のデータを指すポインタを作る．

それぞれの構成の仕方に対応して，特定の演算が準備されている．特に，データを（プログラムの記述の構造にしばられずに）実行時に自由に作り出すことができる点は，ポインタ型の存在とともに，リスト処理をも包含しようとした設計方針を強く感じさせる．

ただし，この機能には，いわゆるガーベジコレクションが伴っていない．したがって，一度作り出したものはプログラマが（ポインタの値をとっておくなどして）管理しなければならない．これは，処理系の作成の便宜を考えてのことであるが，概念の整理上は，中途はんぱな感じがする．

いずれにせよ，PASCAL は，データ構造を自由に設定できることがその特徴である．ただ，データ構造の内部構造の点で充実はしているが，そのデータ構造を外からながめての演算の設定等の面では，次にのべる SIMULA 67 に一歩ゆずるものがある．

4.2 SIMURA 67

シミュレーション用の言語という面を強く持ちあわせているが，ここでは，汎用言語としての面からながめてみることにする．

SIMURA 67 は O.-J. Dahl らによって開発された言語である．ALGOL 60 におきかわるもの，という意識も強く class という概念の導入を除いては，ほぼ ALGOL 60 をそのまま使っていると考えてよい．

■ class

SIMULA 67 では，ALGOL 60 と同じ形のデータが存在する．つまり，整数，実数，論理値，文字と，これらの配列である．これらの他に，object というものを考える．

object とは，データと共に，手続き（や，シミュレーション用言語なので，途中でとまっている制御の流れ）をも組にしたものである．object は，実行に伴ってつぎつぎに作り出されてゆく．

class とは，こうした object のひな形として，プログラムに書かれたものである．あるいは，広い意味でのデータ構造（配列をメモリと見なせば，メモリ上での表現方法と，外部からみたとき概念上考えられる演算をくみにしたもの）を直接に表現したものである．

作り出された object は，ポインタによって指し示される．その object に属するデータや手続きは，ポインタと共に，その object を定義している class 内の名前を用いて引用される．

class の定義を引用して，新しい object を作り出す演算 **new** が用意されている．こうして，自由に object を作り出すことができる．

例を考えてみよう．

```
class stack (m); integer m;
begin array S[1: m];
      integer i;
      procedure pushdown (x); real x;
          begin i := i+1;
                if i≤m then S[i]←x
          end;
      procedure popup (x); real x;
          begin if 1≤i∧i≤m then x←S[i];
                i := i−1
          end;
      i := 0
end
```

この class を呼出すと，内に，大きさ m の領域と，*popup* および *pushdown* の2つの手続きの組がえら

れる．class の本体はブロック (**begin-end**) であり，そのブロックの本体は *i* := 0 である．この文は，**new** で呼出された時実行される．つまり，初期設定の部分も書き込んでおくことが可能なのである．

今，
s1 :− **new** *stack* (10);
s2 :− **new** *stack* (15);
とすると，2つの *stack* という object が作り出されたことになる．今後
s1. pushdown (*a*), *s1. popup* (*a*)
は，*s1* の *stack* に対する演算を，
s2. pushdown (*a*), *s2. popup* (*a*)
は，*s2* の *stack* に対する演算を意味することになる．

いま，class をある概念レベルに相当するものとして捉えるなら，さらに下位の概念レベルにあたる class から，それが構成されることも考えておかねばならない．SIMULA 67 では，class の定義の結合という手段を導入してこれを可能としている．

■ 制御の流れの構造

ALGOL 60 からそのまま受けついだものの他に，コルーチンに相当する機能をもつ．制御の流れの主体も，また object を中心として考えているから，object 間で，制御のやりとりが自由にできるようにしてあるわけである．

コルーチン機能もまた，シミュレーション用ということから出てきたものである．しかし，class という考え方と共に，非常にすっきりした形で汎用言語の中に持ち込んだ点が，SIMULA 67 の特徴といえる．

ただ，処理系の構造を考えるとき，ガーベジコレクションを含め，かなり大がかりになることは，現在の計算機の構造の上で考えるとき，欠点といえるかもしれない．

4.3 BLISS

BLISS は，W.A. Wulf を中心に，システムプログラム作成用として考えられた言語である．その特徴は，個々のデータに型を考えないこと，そして，番地の概念を大幅に採用している点にある．つまり，現在の計算機のメモリそのものを直接言語のデータとして出発したものといえる．まさにシステムプログラム向きであり，MOHL (Machine Oriented Higher-level Language) である．しかし，このことを出発点としてみとめてしまえば，その上に展開される言語構成には，SP 志向のものを多く見出すことができる．

■ 制御の流れの構造

選択用として **if-then**, **if-then-else** の他に，前に触れたように，**case** や **select** を有する．繰返しについても，**while-do**, **do-while**, **until-do**, **do-until** とともに，**incr-from-to-by-do** や，**decr-from-to-by-do** という，ALGOL 60 での **for-do** に相当するものを有する等多彩な構造をほこっている．

さらに，こうした繰返しや，ブロック，手続きからの抜出しを (**go to** を用いずに) 書くための記法が多数用意されている．こうしたことが可能なのも，BLISS では，式と文との区別を廃し，どんな構文要素も結果として値を返すようになっているからにほかならない．しかし，これはまた，繰返し中で抜け出しの条件を判定しておきながら繰返しの外で再び判定を要することになり，批判のある[6] 点でもある．手続きは，再帰呼出しをも許すが，そうした必要のないものは，別の定義のし方を用意して，オブジェクト・コードの効率向上にも留意している．また，2つ以上の制御を作り出し，互いにコルーチンとして動作させるための道具が準備されている．

■ データ構造

現在のメモリに相当するものを直接データとして考える．このため，番地が表面に出てくる．たとえば，変数を書いたとき，BLISS で評価して得られる値は，その変数の (メモリに割り当てられた) 番地である．番地から値を取り出す演算子として "." が用いられる．こうして，*x* を変数とするとき，.*x* はその中身，..*x* は，*x* を間接番地とみた値，といった風の取扱いになる．

個々の変数が，どんなデータを中身とするかは，それに施す演算による．つまり，アセンブラ言語並みである．これは，いわゆる高級言語の行き方に反するものであるが，システムプログラム用としての便宜を第1に考えた結果であろう．

BLISS では，データ構造を，メモリへの表現のし方として捉えている．したがって，多次元の配列も，与えられた添字の組から，その指定される配列要素の番地を計算する機構だと考える．こうした機構を簡潔に定義する道具 (BLISS の用語で structure という) が用意されている．

データ構造に対して，BLISS 流のものだけでは不十分であることは前にも述べたとおりである．しかし MOHL としては (実用性も考えて)，程よくまとめられたものといえる．

5. 今後の課題

SP 用言語というものについて，制御の流れとデータの構造の面から，望ましい性質等を調べるとともに3つの言語をとりあげて，それぞれが，どう SP とかかわっているかをながめてきた．

しかし，ここで考えている SP というものに対しては，プログラミングし終った結果を留めるための言語だけでは不十分である．SP という過程をうまく書き留めてくれるものを準備する必要がある．つまり，SP のための，ソフトウェア・システムを検討してみることが，大切であり，また，今後の課題である．この中で，SP 用言語もまた育っていくものであると，筆者は考えている．

参考文献

1) E. W. Dijkstra: "Notes on structured programming", EWD 249, Technical University, Eindhoven, Netherland (1969)*
2) E. W. Dijkstra: The structure of the "THE"-multiprogramming system, CACM, Vol. 11, No. 5, pp. 341～346 (1968)
3) E. W. Dijkstra: The humble programmer, CACM, Vol. 15, No. 10, pp. 859～866 (1972)
4) B. Leavenworth, ed.: Control structures in programming languages—"The GO TO controversy"—Debate, SIGPLAN Notices, Vol. 10, No. 11, pp. 53～91 (1972)
5) E. W. Dijkstra: Go to statement considered harmful, CACM, Vol. 11, No. 3, pp. 147～148 (1968)
6) P. Abrahams: "Structured Programming" considered harmful, SIGPLAN Notices, Vol. 5, No. 4, pp. 13～24 (1975).
7) N. Wirth: The programming language PASCAL, Acta Informatica, Vol. 1, pp. 35～63 (1971)
8) W. A. Wulf, et al., BLISS: A language for systems programming, CACM, Vol. 14, No. 12, pp. 780～790 (1971)
9) J. McCarthy, et al.: "LISP 1.5 Programmer's Manual", MIT Press, 1962
10) K. Babcicky and G. M. Birtwistle: Class distinction in SIMULA—some aspects of general programming language, SOFTWARE, Proc. of Conference, Or. D. JEVANS (Gd), Auerbach Poblishers Inc, Princeton, pp. 129～158 (1970)
11) O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare: "Structured programming", Academic press (1972)
12) N. Wirth: "Systematic programming—an introduction", Prentice-Hall (1973)

(昭和50年7月11日受付)

* (1)は(11)の第1部として収録されている．