記号処理 12－7
(1980. 6. 20)

SLisp（単一割当て etc. Lisp）

横井俊夫，元吉文男
（電子技術総合研究所）

1. はじめに　データフロー計算機に対する関心が高まっている。筆者らも、記号処理をも有効に行なう汎用計算機として研究を進めている。しかし、日本の現状は、未知の部分が多く、研究も小規模な試論にとどまるものが多く、その将来像を具体的に印象づけるには至っていない。先進的なアメリカやヨーロッパにおける研究を見ても、学ぶべき所は多々あるにしても、どの部分も確立したといいがたいのが現状である。

データフロー計算機のイメージの具体化，少なくとも、研究の展開の仕方のイメージの具体化には、次の3点に関して、広く同意の得られる提案がなされねばならない。

(1)汎用プログラム言語　(2)計算機構（計算モデル）　(3)アーキテクチャ

本稿の目的である(1)を議論する前に、(2)と(3)に簡単にふれる。Dennisにより、原始データフロー言語として基本的な計算機構が提案され、次に手続（再帰）呼び出しの機能と構造体データを扱う機能が加えられ拡張された。さらに、オブジェクト指向の考え方を取り入れ副作用を含むモニタの記述を可能にするWengのストリーム型データ、繰返し演算での並列度を高めるためのArvindのアンフォールディング解釈法（Unfolding Interpreter）等々の拡張が提案された。これらの努力は、現在までに考案され活用されてきた色々な計算の機構を含め、出来るだけ多くのものをデータ駆動という枠組の上に組み上げようというもので、(1)と(3)を結びつけるという意味でも、まず確立されねばならないものである。しかし、アメリカにおける現状も、検討すべき主立った事柄に一応言及したというのが現状で、まだまだ整理された計算モデルとは言い難い。特に並列計算のモデルとして理論的な大きな展開が期待されるだけに、出来るだけきれいに整理したものとしての確立が急がれる。今までの筆者等の努力は、この点に向けられており、一応の成果を得るに至っている。

(3)に関しては、筆者らは、いまだ具体的な提案は行なってはいない。現状に対する感想を一言述べておく。まず、アーキテクチャの骨格となる部分が、あまりに複雑で脆弱すぎる様に思われる。いずれ、この上に色々なものが積み上げられていくのであろうから、骨格は単純で堅牢なものでなくてはならない。次に、データフロー計算機は、処理装置は多量に安く（その数は現在のキャッシュメモリの容量並み）、記憶装置はさらに多量に安く供給されるという前提でアーキテクチャを決めていかなければならない。そうしなければ、本当のデータフローの有効性は生まれてこないように思われる。この安く多くという事実は、VLSI技術によって達成されるのであるから、VLSI化という前提でアーキテクチャを考えるべきである。その意味で現状を見ると、ハードウェアが高かった、あるいは高いということを意識しすぎているように思われる。いずれにしろ、思いきった様々なタイプの提案が精力的に行なわれるべきである。

次に汎用プログラム言語について、基本的な考え方と何故SLispなのかを概説する。データフロー計算機で処理されるか現在の逐次型計算機で処理されるかによってプログラム言語は異なったものになるのであろうか。基本的な考え方は、

次の2点にしぼられる。

(a) プログラム言語は、自然で正確な計算の記述系という観点から決められるものであるから、処理する計算機の性質に左右されるものではない。

(b) 既存言語の枠から離れ、本来あるべきものとしてプログラム言語を追及していくと、駆動型の計算機構を前提とした言語になる。

すなわち、プログラム言語は本来データフロー計算機を前提としたもので、その機能を制限することにより逐次型計算機でも処理可能となる。このことは、データフロー計算機の優位性を主張すると同時に、言語から見れば、逐次型からデータフローへは連続して移行できるということでもある。

次に、何故Lispから出発するかである。着目するLispの特徴は次の3点である。

(1) その基本となる計算機構は、副作用無しである。しかも、リスト構造という構造体データによって複雑なデータ構造を副作用無しに扱う機構をもつ。

(2) すべての記号処理機能にとって、最も基本となるリスト処理機能を持つ。

(3) 一般的で非常に有用なプログラミングのパラダイムを与えている。

データフロー計算機にとって、(1)は不可欠の要素である。たとえ、数値計算に応用をしぼっても副作用を無くするために複雑な構造体データを必要とする。そのような機構に関しては、Lispの周辺に最も多くの経験が蓄積されている。(2)は、汎用化及びこれから発展する新しい応用分野からデータフローの有用性を広めていくという観点から重要である。(3)は、プログラミングの形式としての汎用性の観点から重要である。このような利点から、まず出発点として選ぶが、Lispの現状は、決して満足すべき状態ではない。実用化を急ぎ取り入れた諸機能の不整合さは、ますます目立ってきている。しかし、PROLOG, Lucid, SCHEME, Clu, VAL等々の成果から、それらを克服し、さらに機能を拡張していく手立てが明らかになってきた。そこで提案されるのが、このSLispである。"S"は、Single Assignment, Standard, Self-Contained, Smart, Structured等々の"S"である。

次章以降に、SLispの第1回の紹介として、代入式の等式化，スコープ規則の静的化，条件式・ブール関数の論理式化，リストのタプル化等を中心に説明する。各説明は、次のような筋書きに従がう。プログラムの記述系として見た時、これこれの欠点がある。それは、これこれの手立てで矯正できる。それは、データ・フロー・マシン（DFMと略記）にこの様に適合するようになる。また、機能を限定すると、逐次型マシン（SQMと略記）ではこの様に処理することができる。

2. 代入式の等式化

計算結果に名前をつけ、後から必要な個所で参照できるようにする機能は、計算を自然に・簡明に・効率良いものとして表現するために不可欠のものである。そのために現在のLispのProg文は、記憶場所の確保（Prog変数の宣言）と、それへの書き込み（Set, Setq）と読み出し（変数の評価）という副作用を伴う機構を、用意した。これは、広く使われている汎用プログラム言語の基本的な計算機構(記述)をそのまま取り込んだものである。その副作用によって、少々の記憶領域の節約は達せられるものの、誤りを生ずる大きな原因を生みだすことになった。そこで望まれるのは、副作用を伴わない名前付けの機構である。その1つの方法が、単一割当て規則（Single Assignment Rule）、1つの変数には、1度しか代入を行なわないという規則による表現である。1例を、

```
Block( t1 := a*a;
       t2 := b*b;
       t3 := t1+t2;
       t4 := t1-t2;
       t5 := Sqrt(t3)/Sqrt(t4); )
```

Fig.1 単純な単一割当て規則

```
Block( x := a*a;
       y := b*b;
       Then
       x := x+y;
       y := x-y;
       Then
       x := Sqrt(x)/Sqrt(y)  )
```

Fig.2 グループ化された単一割当て規則

```
Block( .
       x := ...
       .
       Then
       .
       x := F(x);
       .
       Then
       .
       x := G(x);
       .
     )
```

(a) 単一割当て

```
Prog((x ...)
     .
     x := ...
     .
     x := F(x);
     .
     x := G(x);
     .
   )
```

(b) 副作用(代入)

```
.
.
a := 3;
.
b := 2;
Then
x := Block((a  b)
           x := a*a;
           y := b*b;
           Then
           x := x+y;
           y := x-y;
           Then
           Sqrt(x)/Sqrt(y) )
```

Fig.3 変数の参照関係

Fig.4 複合式

Fig. 1 に示す。[注)] これにより、代入文は、等式の成立という意味合い（明らかに完全な等式ではない）を持つことになる。しかし、この単一割当て規則だけでは、ある程度プログラムが大きくなると、変数名がふえ見通しを悪くする。そこで、実行文をグループに分け、名前の参照の仕方に工夫をこらす。ブロック内のプログラムは、THEN を区切りに用いて、実行文グループの系列として表わされる。グループ内のある実行文の右辺変数は、前方にあるグループをさ前より順々にたどり、同じ変数名を左辺に持つ最初の代入文を参照する。したがって、グループ内については単一割当て規則が守られねばならない。この新しい名前付けに従って Fig. 1 を書き直すと、Fig. 2 のような見通しの良いものとなる。Fig. 2 だけからでは、副作用を伴う代入文との区別は、明らかではない。その違いを Fig. 3 に示す。この機構の原理は、続き関数（Continuation）を用いることによって、代入をラムダ結合（Lambda-binding）に置きかえるということである。代入文の右辺には、複合文として、ブロック自身を書くこともできる。その時、先頭に、外部を参照する変数名を列挙し、外部とのインターフェースを明示する。（Fig. 4）

この新しい名前付けの機構に基づくプログラムは、2種のマシン上で次のように実行される。

（DFM）：Fig. 3 の (a) で矢印を上から下にたどる。すなわち、値がおきると、その変数を参照しているすべての個所に、その値を送り出す。そして右辺の変数値がすべて揃うと、ただちにその文は実行されると解釈すると、素直に、データ駆動に翻案することができる。

注) SLisp では、英小文字を先頭に英小文字，数字，限られた特殊文字で構成される文字列は、変数（Eval モード）を表わし、英大文字を先頭とする文字列は、アトム（Quote モード）を表わす。

例えば、Fig. 5 のようなマイクロ系列群に分解される。少なくとも、グループ内の実行文は並列に実行される。

(SQM)：Fig. 3 の (a) で矢印を下から上にたどる。実行を指示された文の右辺で参照している変数の値をさがしにいくという方法で実行される。計算値は後の参照にそなえて通常は、スタック上に置かれる。例えば、Fig. 2 の2番目のグループの実行が終わった時点でのスタックの内容は、Fig. 6 のようになる。この図によって、副作用無しに名前付けを行なうということの意味が、一層明らかになったものと思う。名前付けの履歴がスタック上に残されている。

```
=>a Then a*a=>(t1.1  t2.1)

=>b Then b*b=>(t1.2  t2.2)

t1:=>x Then x+y=>t3
   =>y

t2:=>x Then x-y=>t4
   =>y

t3:=>x Then Sqrt(x)=>t5.1

t4:=>x Then Sqrt(x)=>t5.2

t5:=>x Then x/y=>t6
   =>y

t6:=>x .....
```

Fig.5 マイクロ系列群

・	
・	
・	
a	3
b	2
・	
・	
x	9
y	4
x	13
y	5

Fig.6 スタックの全中状態

3. スコープ規則の静的化

構造化プログラミングの精神は、プログラムテキスト上で、実行時の動的な変化をすべて把握できるようにするということであった。その精神からすれば、Lisp の動的スコープ規則は、大きな規約違反である。もともと、ラムダ計算そのものは、静的スコープ規則であるともいえるし、Lisp において動的にしたのも、インタプリタを簡単化すること以外に大きな理由があった訳ではない。実際に書かれたプログラムを見ても、全プログラムが参照するグローバル変数か、そのプログラムだけのローカル変数かという両極端の場合がほとんどで、動的な結合をうける自由変数の利用は極まれであり、また動的である必要性も少ない。そこで、S Lisp ではスコープ規則を静的なものとする。この点は、多くの汎用言語と同じ機構となる。

グローバル変数は "harmful" であるといわれる様に、単純にスコープ規則を強制するだけでは、見通しの良いプログラムになるとはいいがたい。そこで最近の言語では、手続きの定義の際に、仮引数の指定だけでなく、出力値に関する記述、スコープ規則に従って参照する外部変数の指定を書かせる。さらに、各変数に参照を許す手続き名の記述を要求する場合もある。このようにして、各プログラム・モジュールの外部とのインタフェースを明示させることにより、見通しの良く誤りの生じにくいプログラミングを可能にしようとしている。S Lisp でも、関数の定義に際して外部とのインタフェースを次の様に記述する。

<formal-argument> | <output-variables>
| <free-variables> :

Fig. 7に一例を示す。ここでは、引数や変数にタイプの指定は書かれていない。タイプ指定の無い場合は、いかなるタイプでも受け付け実行される。このタイプレスの機構は、会話型でのプログラミングを手軽なものとするLispの良さである。しかし、後からプログラムを見た場合は、タイプ指定は、理解を大いに助けるものとなる。SLispでは、タイプ指定が、コンパイラによる誤り検出用として利用できる。仮引数内の"?"は、その数と相対的な位置が、出力変数の記述に対応する。数が、その関数のarityを示す。位置は、出力がどの場所に対応するかで、SLispの大きな特徴であるPROLOGとLispとの融合のための手立ての1つである。

Fig. 7の関数定義式の表記法からも明らかなように、静的スコープ規則に従うということは、定義式を等式化された代入式と同等に扱えるということである。大きな違いは、再帰的な名前の参照が許されることである。それを明示するのが、Rec式(Fig. 8)である。相互に再帰的に呼び合う関数群(再帰グループ)を一まとめにする。Rec式内の代入式の右辺の関数名は、まずRec式内の同じ左辺変数を探してからというように、参照規則が拡張される。

```
Block(
    .
    .
    b := 5;
    .
    f := = (a  ?)|(r)|(b):
           Block(
             x := a*a;
             y := b*b;
             Then
             x := x+y;
             y := x-y;
             Then
             r := Sqrt(x)/Sqrt(y) );
    .
    x := f(6);
    .
    .
)
```

Fig.7 関数定義式

```
Rec( <name> := <argument-spec> : <body>
                .
                .
                .
     <name> := <argument-spec> : <body> )
```

Fig.8 再帰グループ

以上のような静的スコープ規則と処理マシンの関係は、以下のようになる。(DFM)変数の結合をどう扱うかという方から、λ-計算の計算機構は大きく3種類に分かれる。

(1) 変数を、結合された値ですべて置き換える。
(2) スタック上に結合情報を蓄え、変数値が必要になった時、スタックに探しにいく。
(3) 結合が行なわれると、参照している個所すべてにその値を送りだす。

(1)は、Churchによるλ-計算のもともとの定義である。(2)は、LandinがSECDマシンとして提案して以来、Lispを初めとする実用言語はすべてこの機構による。(3)が、これから開発すべきデータ・フロー・マシンの機構である。(2)は、高速処理を可能にしたが、(1)の持つ関数引数の一般性を失うことになる。スコープ規則を静的なものにし、自由変数を明示することによって、(3)は(1)に近づく。しかも並列処理による高速処理も当然、得られることになる。しかし、(2)の持つ大きな長所がある。会話型で1ステップ毎を解釈しつつ処理できるようにするには、(2)のような機構が必要である。

（SOM）アクセス・リンクとコントロール・リンクを別にし、Algol等の処理系の機構を取り入れれば良い。

4. 条件式・ブール関数の論理式化　Lispでは、あらゆる式が値を持つ。これを利用して、すべての式を述語として扱ってもよいという便法が設けられている。ここでいう述語とは、"真(T)"か"偽(F)"のみを値とする関数である。この便法を利用して、Fig. 9の(a)や(c)のようなプログラムが書かれる。少々効率を良くするものの、プログラムの理解度を悪くする大きな原因となる。SLispでは、まずCond式の引数のcar部，ブール関数（And, or, Not）の引数は厳密に述語のみに制限する。さらに述語の引数となる関数は、副作用の無いものに制限する。（Fig. 9 (b), (d)）。従って、ブール関数（And, Or）の引数の順番を入れかえても、同じ値を得る。これは、数学的な論理記号と同じ意味となり、ブール関数の意味を簡明に取り扱えるようになる。同様に、Cond式を整理し、その意味が簡明に定まるようにする。まず、Cond式の引数となる対の意味付けをはっきりとさせる。Dijkstraの見張付命令の考え方を用い、この対を見張付き式として定義する。（Fig. 10 (a)）。そして、Cond式を見張付き式の集合の中で、値を持つもののどれかを選択するという、非決定的な選択を行なうものと定義し直す。こうすることによって、Dijkstraの指摘通り、Cond式の意味は簡明な論理式として表わすことができる。

```
(Cond  ((Setq  x  (Cdr  y))  ..... )
       (T  ..... ))

      (a)


x := Cdr(y);
Cond((~Null(x)  ..... )
     (Null(x)  ..... ))

      (b)


And( Cdr(a) Cdr(b) Cdr(c) )

      (c)


And(~Null(Cdr(a) )) ~Null(Cdr(b)) ~Null(Cdr(c)))

      (d)
```

Fig.9 Cond式とブール関数への整形

```
<antecedent> -> <consequent>

     T -> x = x
     F -> x = ⊥

     (a)


Cond( <guarded-expr>
        :
        :
      <guarded-expr> )

     (b)
```

Fig.10 見張付式とCond式

上記のような定義を各マシンでどう処理するかを以下に示す。
（DFM）ブール関数については、すべての引数を並列に評価し、And（Or）は、どれか1つでも"偽(真)"になったら、全体が"偽(真)"となるように処理することができる。Cond式に関しては、同じように、すべての見張付き式を並列評価し、値の定まるのが得られ次第、式全体の評価を終えたことにすることができる。ここでは，Cond式の引数としては見張付式としているが，任意の式に拡張できる。いずれにしろ、上記の定義通りの処理が可能である。さらに、見張付き式については、もしconsequent部にも副作用の無い式のみを書くように制限すると、antecedent部と並列に処理することができる。Fig. 11にそのような見張付き式の表記法(a)とAssoc関数の例(b)を示す。このAssoc関数は，各ドット・ペア

の検査を次々と起動し，最初に見つけたペアを値として返す。
（SQM）ブール関数については，すべての述語が発散しなければ，定義通りの評価が行なえる。Cond式については，すべてのantecedent部が発散せず，それが"偽"であれば，他の条項付き式の評価に移るという機構により定義通りの評価となる。

5. リストのタプル化　　リスト構造は，二進木を表わすものであるが，実際の使われ方を見ると一次元の列として処理される場合が多い。cdr部を次々とたどっていく場合が非常に多いということである。Assoc関数もその例である。Lispマシンでは，cdr部がポインタではなく，それが指すものを置くという圧縮リストを設け，処理の高速化と記憶領域の節約を計ろうというものもある。人工知能向言語などでは，この事実を陽に認め，基本データ構造にリストよりタプル(n-対)を置くものもある。タプルはvector, sequenceなどと呼ばれることもあり，参照は一次元配列と同様で，変更や生成に関しては，リストと同様の動的な変化も許される。（DFM）一次元列としてみる場合，リスト構造の逐次性が，並列度を高める障害になる。タプル化により，非常に大まかに言えば，nからlog nへの高速化が達せられる。しかし，タプルの配分とゴミ集めのうまい機構を考案しなければならない。（SQM）タプル化により主記憶への参照回数を減らすことができる。

6. おわりに　　SLispの紹介（オー対）として4つの特徴を述べた。残りのものについては，次の機会にゆずる。ただ，雰囲気を理解していただくために，8-queensの問題の骨組みだけをSLispで書きFig.12に示す。繰返し文の表記法はLucidに準ずる。QueenとNo-attackは論理プログラミングでいう述語である。Queenは，i-列に対し，行rを非決定的に選択する。No-attackは，i-列，r-行の女王が盤-jに対し安全であるか(Success)か否(Fail)かを決定する。非決定的な部分を記述するのが述語の役割である。Create-board, Next-board, Print-boardは通常の関数で，それぞれ，新しい盤面を作り出す，ある盤面に新しく女王を加えた次の盤面を作り出す，盤面を出力するものである。このような非決定的プログラムは，DFMでは並列処理制御で，SQMでは，後戻り制御で処理される。

<antecedent> -» <consequent>

(a) 並列条件対式

```
Assoc(u  v) =

Cond(Null(v) -> Nil
     Otherwise -> Cond(Atom(Car(v)) -> Error
                       Otherwise -> Cond(u=Caar(v) -> Car(v)
                                         ~(u=Caar(v)) -»Assoc(u Cdr(v)))))
```

(b) Assoc関数

Fig.11 並列条件付式と使用例

```
Eight-queens := ():
                Block( First:(i  j) := (1  Create-board());
                       Next:(i  j) := (i+1  Block((Queen: i  r?);
                                                  (No-attack: j  i  r);
                                                  Next-board(j  i  r) ));
                       Print-board(j) As-soon-as i=9)
```

Fig.12　8-queensの問題

謝辞：研究の機会と示唆に対し，渕一博パターン情報部長に感謝する。