

リスト上のパターン・マッチングの機能を備えた 記号処理用言語

横内 寛文

(東京工業大学 理学部 情報科学科)

1. はじめに

パターン・マッチングは記号処理に関連した人工知能用ソフトウェアにおいて重要な機能の一つである。この報告では、リスト上のパターン・マッチングの機能を備えたプログラム言語をLISPをもとに設計して、LISP プログラムに変換するコンパイラについて述べる。

SNOBOLにおけるパターン・マッチングは記号列上のものであるのに対して、ここで考えているのはリスト（特にLISPのデータ構造）におけるものである。パターン・マッチングとは与えられたデータがあるパターンに照合するか否かを調べる手続きであるが、次節で matching という用語で正確な定義を与える。さらに matching との対で generating と呼ぶ機能を提案する。generating とは与えられたパターンのデータを生成する手続きをいう。1組の matching と generating はいわゆる production system の production と同様な概念である。matching と generating を繰り返すことによりデータの変換処理を実現できる。

重要な課題はパターンの記述法である。パターンの記述はプログラム言語以外でもすでに行なわれている。例えば論理学で用いる公理型、数学の公式の多くがそれである。自動証明を行なう際には公理型とのパターン・マッチングとして、数式処理システムでは公式とのパターン・マッチングとしてそれぞれのパターンを利用している。プログラミングにおいても通常に近い自然な記述法でパターンを書けることが望まれる。この報告ではこの点を重点的に述べ、パターンの一記述法を提案する。

一般に、LISP でシステムを作る際には、データの外部表現である記号列と内部表現であるリストとの相互変換ルーティンが必要である。本システムでは pair grammar [3] を用いてユーザが記号列とリストの対応を定義して、相互変換ルーティンを自動的に生成させる機能を備えている。このうち入カルーティンは記号列として書かれたパターンをリストに変換する際にも利用される。

2. パターンの記述法

パターンは metaexpression (metaexp と略記する) と呼ぶ特別な S-expression によって記述される。metaexp のうちパターンを表わす最小単位を unit-metaexp と呼ぶ。unit-metaexp は LISP の atom か (atom . S-expression) の形をしている S-expression である。unit-metaexp の atom は metavariable (metavar と略記する) と呼ぶユーザがその種類とともに指定する。metavar の種類を type と呼ぶ。unit-metaexp の形、すなわち atom か否かはその metavar の type ごとに定まっている。unit-metaexp はあるパターンを表わすとともに正確にはその metavar の type ごとに決められた matching ないしは generating を行なう LISP 関数を表わしている。metaexp は unit-metaexp から構成された LISP の S-expression で、パターンを表わすと同時にその unit-metaexp が表わす関数より構成される matching function ないしは generating function を表わしている。例えば、 U_1, U_2, U_3 を unit-metaexp, U_1^m, U_2^m, U_3^m を U_1, U_2, U_3 に対する matching function, U_1^g, U_2^g, U_3^g を generating function

とすると, $\text{metaexp}(U_1, U_2, U_3)$ に対する matching function は,

$$\lambda[[x]; \text{not}[\text{atom}[x]] \& \text{not}[\text{atom}[\text{cdr}[x]]] \& U_1^m[\text{car}[x]] \& U_2^m[\text{cadr}[x]] \& U_3^m[\text{caddr}[x]]]$$

であり, $\text{generating function}$ は

$$\lambda[[]; \text{cons}[U_1^g[[]]; \text{cons}[U_2^g[[]]; U_3^g[[]]]]$$

である。以上の定義から, データ D の metaexp M による matching とは, M 中の unit-metaexp に対する matching function よりある規則に従って構成した関数を D に apply することに他ならない。ある規則とは何かということについての厳密な定義は省略するが, 上の例から明らかであると思う。generating についても同様である。各 type ごとに決められた unit-metaexp の matching および $\text{generating function}$ は直観的な意味に合うように定義されなくてはならないが, 原則として次のような性質を満たすように構成されている。(a) 同一の unit-metaexp は同一のリストを表わしている; (b) matching が成功した場合各 metavar にその unit-metaexp に対応する被 matching データが代入される。ただし, 実際の代入は metavar の前に "\$" を付けた atom になされる; (c) generating に先だてて matching あるいはユーザのプログラムによって各 metavar に値が代入されていなくてはならない。また, あるパターンを含むこと等表現できるように unit-metaexp が metaexp (特に submetaexp と呼ぶ) を含むことを許す。一般に metaexp は階層構造を成すことになる。 unit-metaexp のどの位置が submetaexp であるか等も type ごとに定まっている。さらに, unit-metaexp が表わす関数を詳細に決定させるために, 各 metavar に attribute とその値を付けることができる。attribute の種類とその意味等も type ごとに定められる。現在, 約 20 の type が用意されているが, このうち代表的なものだけを解説する。ただし, 以下は概説であり実際には言語の理解に応じて対応する関数を詳細に制御できるようになっている。

(1) meta-form (プログラム上では FORM という名前を用いる。以下同様)

unit-metaexp の形は atom で任意のリスト (データ) を表わす。matching においては attribute PROPERTY とその値として 1 引数関数を付けることができる。 matching function は $\lambda[[x]; \text{prog2}[\$metavar:=x; t]]$, attribute PROPERTY とその値 p が付けられた場合は $\lambda[[x]; p[x] \& \text{prog2}[\$metavar:=x; t]]$ と定義される。 $\text{generating function}$ は $\lambda[[]; \$metavar]$ である。

(2) meta-constant

unit-metaexp の形は atom でその metavar 自身を表わす。すなわち, matching function は $\lambda[[x]; \text{eq}[\text{quote}[\text{metavar}]; x]]$, $\text{generating function}$ は $\lambda[[]; \text{quote}[\text{metavar}]]$ である。

(3) meta-list

unit-metaexp は ($\text{metavar} \cdot \text{submetaexp}$) の形をして, ($\text{metavar} \cdot \text{submetaexp} \cdot \dots \cdot \text{submetaexp}$) のパターンを表わしている。例えば, top level の要素に少なくとも 1 つ "A" を含むリストは, L と M を meta-list , A と NIL を meta-constant として ($L \ A \ M$) と表わせる。この metaexp の unit-metaexp は ($L \ A \ M \cdot \text{NIL}$), ($A \ M \cdot \text{NIL}$), A , ($M \cdot \text{NIL}$), NIL の 5 個であることに注意せよ。 meta-list の matching function は,

$$\lambda[[x]; m[x] \& \text{prog2}[\$metavar:=x; t]]$$

$$m[x] = [\text{matching-function-of-submetaexp}[x] \rightarrow \text{prog2}[\$metavar:=x; t];$$

$$\text{atom}[x] \rightarrow \text{nil}; \quad t \rightarrow m[\text{cdr}[x]]]$$

と定義される。ただし、matching が成功した際には $\$!metavar$ に submetaexp に対応するリストが代入される。generating function は、

```
λ[[];g[$metavar]]
g[x] = [eq[x;$!metavar] → generating-function-of-submetaexp[];
        t → cons[car[x];g[cdr[x]]] ]
```

と定義される。例えば、先の metaexp による matching の後、(L B M) (B は meta-constant) に対する generating を行なうと、リストの top level で最初の A が B に置き換わる。なお、上の例の matching ではリストの終わりが NIL であることのチェックが行なわれるが、NIL を meta-form とするとこのチェックが行なわれなくなることには注意せよ。また、すべての A を B に置き換える等の操作も必要になるだろうが、次節で述べる機能を用いて実現できる。

(4) meta-contain

unit-metaexp は (metavar submetaexp) の形をして、submetaexp のパターンを含むことを表わす。matching, generating function はそれぞれ

```
λ[[];m[x]&prog2[$metavar:=x;t]]
m[x] = [matching-function-of-submetaexp[x] → prog2[$!metavar:=x;t];
        atom[x] → nil; t → m[car[x]]Vm[cdr[x]] ] ,
λ[[];g[$metavar]]
g[x] = [eq[x;$!metavar] → generating-function-of-submetaexp[];
        atom[x] → x; t → cons[g[car[x]];g[cdr[x]]] ]
```

と定義される。また、取り扱うデータが Polish notation を用いて表現されている場合が多いことを考慮して、matching で submetaexp のパターンを捜す際に上記の他 leftmost と leftmost-innermost の2つの方法を attribute SEARCH によって指定できる。attribute SEARCH の値を NORMAL, LEFT, INNER と指定するとそれぞれ上記の関数、leftmost および leftmost-innermost でパターンを捜す matching function が対応付けられる。generating function もそれに伴って変化する。

(5) meta-value

unit-metaexp の形は atom で、現在の $\$metavar$ の値を表わしている。ユーザのプログラムで $\$metavar$ に値を代入させるか、あるいはその metavar を meta-form 等に指定して matching によって値入させ、後 meta-value と宣言しなおして matching で使用できる。generating に対しては meta-form と同等である。

(6) meta-substitute

unit-metaexp は (metavar m_1 m_2 m_3) の形をしている。ただし、 m_1, m_2, m_3 は submetaexp である。 m_3 による generating を行ない、その結果のリスト中の m_1 のパターンをすべて m_2 のパターンに置き換えたりストを表わす。 m_1 のパターンの探索の方法として、meta-contain 同様 attribute SEARCH により3種類を指定できる。例えば、SUB を meta-substitute で attribute SEARCH の値を INNER, A, V を meta-form, & を meta-constant とすると (SUB (& A A) A V) は $\$V$ の値のリスト中、(& A A) をすべて A に置き換えたりストを表わしている。

(7) meta-or, meta-and, meta-not

meta-or と meta-and の unit-metaexp の形は (metavar S_1 S_2 ...) で、

meta-not は (metavar S) である。ただし, S, S₁, S₂, ... は submetaexp である。matching function はそれぞれ,

$$\begin{aligned} &\lambda[[x]; (S_1^m[x] \vee S_2^m[x] \vee \dots) \& \text{prog2}[\$metavar:=x;t]], \\ &\lambda[[x]; S_1^m[x] \& S_2^m[x] \& \dots \& \text{prog2}[\$metavar:=x;t]], \\ &\lambda[[x]; \text{not}[S^m[x]] \& \text{prog2}[\$metavar:=x;t]] \end{aligned}$$

である。ただし, S^m, S₁^m, S₂^m, ... は S, S₁, S₂, ... に対応する matching function を表わす。generating function は meta-form と同様である。

(8) meta-define

unit-metaexp の形は atom で, attribute PATTERN とその値として metaexp を付ける。matching function は,

$$\lambda[[x]; m[x] \& \text{prog2}[\$metavar:=x;t]],$$

m = the matching function of the metaexp attached to the metavar

と定義される。すなわち, unit-metaexp の表わすパターンを attribute で定義していることになる。attribute として付ける metaexp 中に定義しようとしている metavar を書くことができ, 再帰的な定義が可能である。例えば, A を含むか否かの matching は meta-contain を用いる他に meta-defined を次のように定義して D による matching でも実現できる。すなわち, A を meta-constant, B を meta-form, OR を meta-or, D の attribute PATTERN の値を (OR A (D . B) (B . D)) とする。generating function は meta-form と同じである。

(9) meta-define1

meta-define の拡張である。attribute PATTERN の値として, ((metavar . α) metaexp) を付ける。ただし, α は任意の S-expression。unit-metaexp は (metavar . α) の α 中 NIL 以外の atom の位置に submetaexp を入れた形をしている。attribute PATTERN の metaexp 中の NIL 以外の α の atom を対応する submetaexp に置き換えた metaexp をパターンの定義と解釈して, meta-define と同じ matching を行なう。generating は meta-form と同じである。meta-define1 を用いると先の meta-list, meta-contain と同様な機能を実現できる。L と C を次のように定義すれば, L, C は matching に対してそれぞれ meta-list, meta-contain と同様な働きをする。すなわち, L と C を meta-define1, A を meta-form, OR を meta-or, L と C の attribute PATTERN の値をそれぞれ ((L . P) (OR P (A . (L . P))))

((C . P) (OR P ((C . P) . A) (A . (C . P))))

とする。meta-define1 は meta-define とともにパターンの記述能力を飛躍的に高める。

3. 言語

言語設計について述べる。プログラムは図1のように LISP の関数定義と同様な方法で書かれる。function は通常の LISP 関数と matching あるいは generating 等を行なうシステム固有の特殊関数とから通常の規則に従って構成させた S-expression である。従って, 我々の言語は見かけ上 LISP とまったく同じである。ただし, 特殊関数がコンパイラによって対応する通常の LISP 関数群に変換される点が異なる。コンパイラによって変換される特殊関数を operation function と呼ぶ。interpretive に matching ないしは generating を行なえば, operation function

をコンパイルすることなく通常の LISP 関数として定義できるわけだが、効率を上げるためにコンパイルされる。以下、個々の operation function について解説する。それぞれの書式は図 2 にまとめて示してある。

```
DEFINE (function-name function) ...
```

図 1

```

(DECLARE (TYPE (type metavar ... ) ...
          ATTRIBUTE (metavar (attribute value) ... ) ... ) form)
(MATCH metaexp form)                (GENERATE metaexp)
(PARALLEL form ((MATCH metaexp) form) ... (T form))
(CHANGE form ((MATCH unit-metaexp) form) ... )
(BACK (MATCH metaexp form) form)    (SAVE form)

```

図 2

(1) DECLARE

各種の宣言を行なうためのものである。コンパイラにより第 2 引数の form に変換されるが、第 1 引数に書かれた情報がコンパイラに通知される。図 2 で示した宣言は metavar の登録と attribute の指定である。この他にも多種の制御が可能である。各宣言は再宣言されない限りプログラム・テキスト上第 2 引数に書かれた form 中で有効である。DECLARE は Algol の block 文と同様な働きをする。

(2) MATCH

第 1 引数の metaexp に対応する matching function を第 2 引数の form に apply させた form に変換される。interpretive に考えた場合、第 1 引数は quote されているものとして処理される。従って、実行中に metaexp を生成して matching を行なうことはできない。また、metaexp は記号列で与えることもでき、この場合は次節で述べる入力ルーティンを通して S-expression に変換した後に処理される。記号列の表現は atom の印字名を用いる。

(3) GENERATE

metaexp に対応する generating function を空リストに apply させた form に変換される。metaexp に対する注意は MATCH と同様である。

(4) PARALLEL

同一のデータに対して複数の matching を行なう際に効率を上げるためのものである。コンパイラにより次のような効果を持つ form に変換される。第 1 引数に書かれた form の値に対してそれぞれの matching を行ない、どれかが成功すれば対応する form を eval してその値を返し、すべての matching が失敗した場合は T に対応する form の値を返す。この際に、matching が並行に行なわれる。厳密には各 metaexp 中の共通部分をできるだけまとめて matching が行なわれる。従って、同一のデータに対していくつかの matching を行ないたい場合、metaexp を並べて書くだけで自動的に最適化される。

(5) CHANGE

リスト中のあるパターンをすべて別のパターンに置き換える処理を実現するためのものである。図 2 中の unit-metaexp の metavar はすべて同じでなくてはならない。これを change の metavar、その type を change の type と呼ぶ。change の type として許されるのは前節のうち meta-list と meta-contain の 2 種類だ

けである。CHANGEは次の処理をするformに変換される。changeのtypeが定める方法でそれぞれのsubmetaexpのmatchingを並行に行ない、どれかが成功すれば対応するformをevalした値のリストに成功したパターンのリストを置き換える。さらにmatchingを繰り返し新しいリストを作っていく。matchingの方法は,meta-list, meta-containの本来のmatchingと同様である。リストの生成の方法はそれぞれ厳密に定められているが省略する。CHANGEを用いるとリスト中のtop level要素のAをVに、BをWに置き換えるには、次のような関数を定義すればよい。

```
(LAMBDA (X) (DECLARE (TYPE (CONSTANT A B V W) (LIST L M) (FORM NIL))
  (CHANGE X ((MATCH (L A M)) (GENERATE (V M)))
    ((MATCH (L B M)) (GENERATE (W M)))
    )))
```

(6) BACK

backtrackingを実現するためのものである。コンパイラにより次の処理をするformに変換される。matchingを行ない成功すればBACKの第2引数のformをevalして、Tであれば値Tとして返り、Fであればmatchingを続行する。この操作を繰り返してmatchingが失敗すれば値Fで返る。

(7) SAVE

matchingが成功して次の処理を行なう時に、一時的にmatching成功時の情報を保持させるためのものである。プログラム・テキスト上SAVEの左で最も近いMATCHで使用する変数(\$metavarと\$!metavar)をすべて入-bindして、SAVEのformをevalするようなformに置き換えられる。

4. 入出カルーティンの自動生成

我々のシステムは、ユーザが記号列とリストの対応関係をpair grammar [3][7]を用いて定義し、入出カルーティンを自動的に生成させる機能を備えている。ここで用いるpair grammarとは、記号列とリスト上の1組の文脈自由文法で、非終端記号と生成規則がそれぞれ対になっているものを指す。両者の文法の非終端記号と生成規則に対応付けがあるために、導出木にも対応関係が定義できる。図3のように導出木を経由して記号列とリストの対応が定められる。厳密な議論は省略するが、このような原理に基づいて入出カルーティンが生成される。

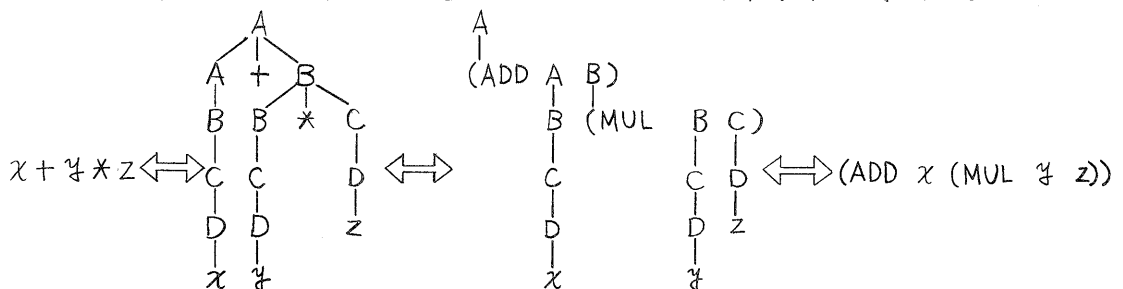


図 3

例えば、算術式とそのリストの対応は図4のように定義る。図3で示した導出木は図4のpair grammarに対するものである。図4中でTERMINALに続くリストは記号列に対する終端記号である。これらは予約語として取り扱われ、予約語に区切られた記号列はidentifierとしてまとめられる。RULE中のXがこのidentifier

に対応している。非終端記号は RULE の左辺 A, B, C, D である。このうち A が開始記号として処理される。システムは図 4 のような pair grammar を読み込んで、TOLIST と TOSTRING という名の入出力関数を自動的に生成する。ただし、図 4 中 TERMINAL の記号と記号列に関する生成規則の右辺の記述は atom の印字名を用いる。また、生成される入出力関数を取り扱う記号列は文字のリストである。

```

INOUT
((IN TOLIST) (OUT TOSTRING)
 (TERMINAL + - * / ** '$$(' '$$')')
 (IDENTIFIER X)
 (RULE (A (+B      (PLUS B))
         (-B      (MINUS B))
         (A+B      (ADD A B))
         (A-B      (DIF A B))
         (B         B))
        (B (B*C      (MUL B C))
          (B/C      (DIV B C))
          (C         C))
        (C (D**C      (EXP D C))
          (D         D))
        (D (X         X)
          ($$('A)' A))
          ))

```

記号列上 リスト上

図 4

```

INOUT
((IN INPUT) (OUT OUTPUT) (IDENTIFIER X)
 (TERMINAL '$$' '=' '$$' '>' 'V & V E ~
            '$$(' '$$')' ;)
 (RULE (A ($$'B = A' (%EQUIV B A))
         (B         B))
        (B ($$'C B' (%IMPL C B))
          (C         C))
        (C (DVC      (%OR D C))
          (D         D))
        (D (E&D      (%AND E D))
          (E         E))
        (E (~E       (%NOT E))
          ($$('VX)'E' ((%ALL X) E))
          ($$('EX)'E' ((%EXIST X) E))
          (X         X)
          ($$'X(F)' (X . F))
          ($$'(A)' A)
          (F (A       (A))
            (A;F     (A . F))
            ))

```

図 5

5. 例

論理式を冠頭和積標準形 (prenex-conjunctive normal form) に変換する問題を例に取り上げる。図 5 と図 6 が pair grammar と関数定義の一例である。図 5 の pair grammar は、metaexp の記述でも用いるため述語記号あるいは関数記号の引数に論理式を書くことを許すように定義されている。図 6 中の metaexp は atom の印字名として書かれてあり、システムはこれらの atom を分解 (explode で) して文字のリストとし、DECLARE の INMETA の次に書かれた関数 INPUT によって対応する s-expression に変換して処理している。INPUT は図 5 の pair grammar から生成された入力関数である。“GENSYM” は新たな atom (G1, G2, ...) を作り出す通常の LISP 関数である。図 6 のプログラムの解説にあたっては次の点を注意しておく。第 1 に、Vv あるいは Yv の scope 中に Vv あるいは Yv が現われることがあり得る点である。このため STEP 1 の操作が若干複雑になっている。第 2 に、change の metavar の matching の方法を、attribute SEARCH により leftmost と leftmost-innermost とに使い分けている点である。第 3 に、第 2 節で解説していなかった attribute NEXT を使っている点である。change の type として meta-contain が指定されている場合で、submetaexp の matching が成功して新しいリストを作った際には、通常その新しいリストに対して再 matching が行なわれるが、attribute NEXT とその値 T が付けられた場合は再 matching が行な

```

DEFINE
  (TRANS (LAMBDA (EXP)
    (DELRARE
      (TYPE (FORM A B C D G NIL) (CONTAIN P Q R) (NOT NOT) (OR OR)
        (CONSTANT %EQUIV %IMPL %OR %AND %NOT %ALL %EXIST)
        (SUBSTITUTE SUB) (DEFINE DEF DEFL) (LIST L M)
      ATTRIBUTE
        (P (SEARCH INNER) (NEXT T)) (Q (SEARCH LEFT)) (R (SEARCH INNER))
        (SUB (SEARCH LEFT) (NEXT T))
        (DEF (PATTERN (OR A (DEFL L DEF M))))
        (DEFL (PATTERN (NOT (OR (%ALL A) (%EXIST A)))))
      INMETA INPUT)
    (PROG ()
      * STEP 1 : ELIMINATE ALL REDUNDANT QUANTIFIERS.
      (SETQ EXP (CHANGE EXP
        ((MATCH $$$Q((VA)NOT(DEF))') (GENERATE $$$'NOT(DEF)'))
        ((MATCH $$$Q((HA)NOT(DEF))') (GENERATE $$$'NOT(DEF)')) ))
      * STEP 2 : RENAME VARIABLES.
      (SETQ EXP (CHANGE EXP
        ((MATCH $$$'((VA)B)') (PROG2 (SETQ G (GENSYM))
          (DECLARE (TYPE (VALUE A))
            (GENERATE $$$'(VG)SUB(A;G;B)'))))
        ((MATCH $$$'((HA)B)') (PROG2 (SETQ G (GENSYM))
          (DECLARE (TYPE (VALUE A))
            (GENERATE $$$'(\G)SUB(A;G;B)')))) ))
      * STEP 3 : ELIMINATE ALL OCCURRENCES  $\supset$  AND  $\equiv$ .
      (SETQ EXP (CHANGE EXP
        ((MATCH $$$P(A  $\supset$  B)') (GENERATE $$$'~AVB'))
        ((MATCH $$$P(A  $\equiv$  B)') (GENERATE $$$'(~AVB)&(AV~B)')) ))
      * STEP 4 : MOVE  $\sim$  ALL THE WAY INWARD.
      (SETQ EXP (CHANGE EXP
        ((MATCH $$$Q(~(VA)B)') (GENERATE $$$'(\A)~B'))
        ((MATCH $$$Q(~(HA)B)') (GENERATE $$$'(VA)~B'))
        ((MATCH $$$Q(~(AVB))') (GENERATE $$$'~AV~B'))
        ((MATCH $$$Q(~(A&B))') (GENERATE $$$'~A&~B'))
        ((MATCH $$$Q(~A)') (GENERATE A))) ))
      * STEP 5 : PUSH QUANTIFIERS TO THE LEFT.
      (SETQ EXP (CHANGE EXP
        ((MATCH $$$R((HA)BV(\C)D)')
          (DECLARE (TYPE (VALUE C)) (GENERATE $$$'(\A)(BVSUB(C;A;D)'))))
        ((MATCH $$$R((VA)B&(VC)D)')
          (DECLARE (TYPE (VALUE C)) (GENERATE $$$'(VA)(B&SUB(C;A;D)'))))
        ((MATCH $$$R((HA)BVC)') (GENERATE $$$'(\A)(BVC)'))
        ((MATCH $$$R((VA)BVC)') (GENERATE $$$'(VA)(BVC)'))
        ((MATCH $$$R(BV(\A)C)') (GENERATE $$$'(\A)(BVC)'))
        ((MATCH $$$R(BV(VA)C)') (GENERATE $$$'(VA)(BVC)'))
        ((MATCH $$$R((HA)B&C)') (GENERATE $$$'(\A)(B&C)'))
        ((MATCH $$$R((VA)B&C)') (GENERATE $$$'(VA)(B&C)'))
        ((MATCH $$$R(B&(\A)C)') (GENERATE $$$'(\A)(B&C)'))
        ((MATCH $$$R(B&(VA)C)') (GENERATE $$$'(VA)(B&C)')) ))
      * STEP 6 : DISTRIBUTE, WHENEVER POSSIBLE, V OVER &.
      (SETQ EXP (CHANGE EXP
        ((MATCH $$$R((A&B)VC)') (GENERATE $$$'(AVC)&(BVC)'))
        ((MATCH $$$R(AV(B&C))') (GENERATE $$$'(AVB)&(AVC)')) ))
      * STEP 7 ORDER & AND V TO ASSOCIATE TO THE RIGHT, RESPECTIVELY.
      (SETQ EXP (CHANGE EXP
        ((MATCH $$$Q((A&B)&C)') (GENERATE $$$'A&(B&C)'))
        ((MATCH $$$Q((AVB)VC)') (GENERATE $$$'AV(BVC)')) ))
      (RETURN EXP)
    )))

```


われない。図7は生成関数 input, output, trans の適用例を示したものである。図7中の compress は文字のリストからその印字名を持つ atom を生成する LISP 関数である。

```
input[explode[(VX)((VY)P(X)V(VX)Q(X;Y) ⊃ ~(VY)R(X;Y))]]
= ((%ALL X) (%IMPL (%OR ((%ALL Y) (P X)) ((%ALL X) (Q X Y)))
    (%NOT ((%ALL Y) (R X Y)))))
= s (auxiliary definition)
trans[s]
= ((%ALL G3) ((%EXIST G1) (%AND (%OR (%NOT (P G3)) (%NOT (R G3 G1)))
    (%OR (%NOT (Q G1 Y)) (%NOT (R G3 G1)))))
= s' (auxiliary definition)
compress[output[s']]
= (VG3)(EG1)((~P(G3)V~R(G3;G1))&(~Q(G1;Y)V~R(G3;G1)))
```

図 7

6. おわりに

リスト上のパターン・マッチングの機能を備えたりリスト処理用言語について述べてきた。ただし、以上は概要でありその他にもいくつかの機能を備えている。そのうち重要な機能には次のようなものがある。(1) matching の backtrack の制御、(2) 高効率化のための attribute、(3) type 定義のユーザへの開放、(4) 実行中に matching, generating function を作り出す機能等がそれである。システムは HLISP[2] で書かれており、現在ディバック中である。プログラムの大きさは、翻訳ルーティンがカード 2000 枚 (コメント, indentation を付けて)、type 定義 1000 枚, pair grammar 処理 1500 枚, システム提供ライブラリ 500 枚, 合計 5000 枚である。

パターン・マッチングの機能を持つ言語としては、SNOBOL を代表に、SAIL, PLANNER, CONNIVER, QA4, QLISP, INTERLISP, POPLER 等 ([1]) 多種存在しているが、本稿で提案した言語はパターンの自然な表現と強力なパターン・マッチングの実現をめざしている。第2節で解説したように metaexp, unit-metaexp, metavar, type と呼ぶ概念を導入して、パターン・マッチングとそれから自然に導かれる generating を定義した。パターンを表わす式、すなわち metaexp はある関数を定めていて、その関数の実行を matching ないしは generating と定義した。metaexp はあるパターンを表現していると同時にある関数を定めているわけである。換言すれば、metaexp はそれが定める関数によってあるパターンを表現していることになる。このことにより複雑な metaexp に対する matching, generating を厳密に定義することができる。また、matching, generating の詳細な制御はすべて type 定義の与え方にまとめることができる。どれだけの type をいかに定義するかはむずかしい問題であり、経験によって決められるべきことであろうが、type 定義とコンパイラ本体を分離できるため、type の追加、変更は自由に行なえる。パターンを通常のようにデータの“鑄型”と考えることもできるだろうが、特にリスト上の複雑なパターン・マッチングを実現するには限界があるように思われる。

本稿で述べた metaexp は、matching ないしは generating の関数を表わしてい

ると同時に、データの specification としての性格も兼ね備えている。構造を持ったデータに対する specification と verification に metaexp を用いることを、Reynolds [4] と同様な方法で考えてみるのも意味があるように思う。また、一般に人工知能用言語に必要な機能 ([1]) は、パターン・マッチング以外にもいくつかあるだろうが、本稿で述べたシステムはパターン・マッチングにのみ焦点を絞って設計されている。将来、完全な人工知能用言語に拡張することあるいは他の人工知能用言語に本稿で述べた方法を応用することを考えている。

なお、本稿で述べたシステムは Prover Generator [5][6] の拡張である。最後に、日頃御指導いただいている小林孝次郎先生、徳田雄洋助手に感謝申し上げます。

参考文献

- [1] Daniel G. Bobrow and Bertram Raphael : New Programming Languages for AI Research, Third International Joint Conference on Artificial Intelligence, 1973.
- [2] Eiichi Goto : Monocopy and Associative Algorithms in an Extended LISP, Technical Report 74-03, Information Science Laboratories, University of Tokyo, 1974.
- [3] T.W. Pratt : Pair Grammars, Graph Languages and String-to-graph Translations, JCSS 5, 1971.
- [4] John C. Reynolds: Reasoning About Arrays, CACM, Vol.22, No.5, 1979.
- [5] 横内寛文: Prover Generating System の研究と作成, 卒業論文, 東京工業大学, 1978.
- [6] Hirofumi Yokouchi : The Design and Implementation of a Prover Generator, 情報処理学会第19回全国大会, 1978.
- [7] 横内寛文: Pattern Matching を基礎においた人工知能用言語システムに関するレポート, 個人メモ, 1979.
- [8] Hirofumi Yokouchi : A Programming Language for List Pattern Matching, 情報処理学会第19回全国大会, 1979.