Vol. 26 No. 11　　情　報　処　理　　Nov. 1985

25周年記念論文

述語論理的プログラミング
——EPILOG の提案——†

淵　　一　　博††

記号処理研究会第1回研究会 (1977.7.21) の資料である．当時あまり知られていなかった Prolog および論理プログラミングの考え方を紹介し，その発展方向，可能性を論じたものである．まず SNL と SPU の組み合わせを論じた．通常の Prolog は SNL ベースだが，SPU ベースも可能性がある．また，各種構文解析アルゴリズムと論理プログラムの関連を述べた．とくに，Earley/Pratt 方式の論理プログラムを与えた．これらは Prolog の拡張方向の示唆を与える．その他いくつかのアルゴリズムの分析を通して，論理式との関連を調べた．また，流れ図式や結合グラフとの関連を論じた．これは論理プログラムの解釈・コンパイル手法と関連がある．最後に Prolog の拡張方向，問題点を整理，列挙した．これらのいくつかはその後自然に実現されたが，今後の問題に残されているものもある．8年前の論文だが，歴史の中の一つのエピソードとしての意味はあろうかと思われる．

1. プロローグ——まえがきと PROLOG の紹介

プログラムの性質の検証や自動合成において，表明や仕様の記述に述語論理が用いられることが多い．このように，述語論理をいわば補助的に用いる行き方に対して直接に，述語論理の形式そのものの中で「プログラミング」をしようという提案がなされている (Kowalski[1])．

ここでは，その考えを紹介するとともに，その考えを念頭において，実際のアルゴリズムのいくつかを分析してみる．このような具体的な分析を通して，アルゴリズムとロジックの関係を調べ，述語論理的プログラミングの具体化の方向を，いわばボトムアップ的に探ろうというわけである．そのような作業の結果，浮かび上がってくるであろうイメージ（とそれを将来実現したシステム）を，ここでかりに EPILOG と名付けておく．

はじめに，述語論理的プログラミングの考えを（いつものように）階乗を例題にして紹介する．Fact を階乗，Times を積の関係を表す述語，s を「後者」を表わす (Skolem) 関数とすると，

(F1)　Fact $(0, s(0)) \leftarrow$.

(F2)　Fact $(s(x), u)$

$\leftarrow$Fact (x, v) & Times $(s(x), v, u)$.

が成り立つ．これに

(F3)　$\leftarrow$ Fact $(s(s(0)), x)$.

を加えて，これから矛盾 ($\leftarrow$) を導く証明図のひとつが**図-1** である．これは階乗の（ひとつの）計算過程でもある．

そこで，上のような述語論理における記述そのものを「プログラム」と見なそう，というのが述語論理的プログラミングの提案である．

この例では，与えられたのは階乗の仕様そのものであるから，選んだ証明手続きが「完全」であれば，このプログラムの正当性は自明であるといえる．停止性についても同様のことがいえる．

Kowalski[1]は，このような例をいくつか挙げてその考えを説明し，具体化の問題点を論じている．彼の論点のいくつかについては後で触れる．

ところで，Kowalski の提案と同時期に，同じ考えを実際に実現したシステムに PROLOG がある．これは Colmerauer を中心にしたマルセイユ大学のグループが作製したもので，フランス語理解システム，数式処理システム，プラン作成システム，定理証明システムなどに使用されていると伝えられていたものである[2]．

これについて，最近詳しい情報が我が国にもたらされた[3]．PROLOG を用いた具体的なプログラムのひとつであるプラン作製システム WARPLAN[4] は，STRIPS の考えを拡張したもので，有名な3—ブロックの問題も解ける．数式処理については参考文献 5)

† Predicate Logic Programming——A Proposal of EPILOG——
by Kazuhiro FUCHI (Institute for New Generation Computer Technology).

†† (財)新世代コンピュータ技術開発機構

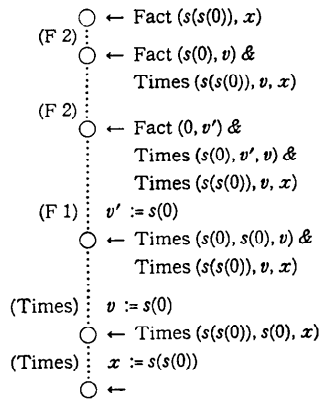

図-1 階乗の計算（証明）—SNL—
Fig. 1 Factorial computation (proof) —SNL—.

を見られたい．

ここでは，純 LISP のインタプリタを**図-2** に示す．PROLOG はなかなか良くできたシステムと判断されるので，その詳しい紹介が先かとも思われるが，ここでは別の機会を待ちたい．PROLOG（述語論理）プログラミングの感じは，図-2 でつかめるかと思う．

このプログラムの読み方は大略次のようにすればよい．左辺の正項は手続き名と見る．右の負項は，左から右へ順に実行される手続き呼び出しと見る．呼び出しはパターン・マッチング（ユニフィケーション）による．上方の式が優先される．失敗すればバックトラックが働く．このように読めば，μ-Planner[22] に似ていることが分かる．

PROLOG には，直接計算，バックトラックの制御，データベースの操作，高階操作などの機能があるが，割愛する．

述語論理の利用に対しては，いくつかの先入主があると思われる．その一つは，形式的にすぎないか，ということである．図-2 のプログラムは，他のプログラミング言語による表現と比べて，どうであろうか．

もう一つは，証明動作はきわめて遅いはずだという偏見である．PROLOG（のインタプリタ）は FORTRAN で書かれているが，大方の予想よりずっと高速であると思われる．あとである程度明らかになるように，よく作れば LISP 程度の速度を持ちうるだろう．

とはいえ，現在の PROLOG が，述語論理的プログラミングにとって必要十分というわけではない（と考えられる）．その問題点については後で触れる．

```
*** MICRO LISP IN PROLOG ***
-LIREFICHIER. +.(DG, 5).
+APPLY (s, CAR, (x. y). z, x).
+APPLY (s, CDR, (x. y). z, y).
+APPLY (s, ATOM, (x. y). z, F).
+APPLY (s, ATOM, x. z, T)-ATOM(x).
+APPLY (s, CONS, x. y. z, x. y).
+APPLY (s, EQ, x. x. z, T).
+APPLY (s, EQ, x. y. z, F)-NEQ(x, y).
+APPLY (s, LAMBDA. x. y. z, r, v)
  -BIND (s, x, r, s1)-EVAL (s1, y, v).
+APPLY (s, LABEL. x. y. z, r, v)
  -BIND (s, x. NIL, y. NIL, s1)-APPLY (s1, y, r, v).
+APPLY (s, a, r, v)-VALUE (s, a, w)
  -APPLY(s, w, r, v).
+EVAL (s, QUOTE. x. y, x).
+EVAL (s, COND. x, v)-EVCON (s, x, v).
+EVAL (s, x. y, v)-EVLIS(s, y, w)-APPLY(s, x, w, v).
+EVAL (s, a, v)-VALUE (s, a, v).
+EVCON (s, (x. y. z). u, v)-EVAL (s, x, F)
  -EVCON (s, u, v).
+EVCON (s, (x. y. y). u, v)-EVAL (s, y, v).
+EVLIS (s, NIL, NIL).
+EVLIS (s, x. y, u. v, s1)-EVAL(s, x, u)-EVLIS(s, y, v).
+BIND (s, NIL, NIL, s)-/.
+BIND (s, a. x, v. y, s1)-BIND ((a. v). s, x, y, s1).
+VALUE ((a. v). s, a, v)-/.
+VALUE (x. s, a, v)-VALUE (s, a, v).
+ATOM (x. y)-/-FAIL.
+ATOM (x).
+NEQ (x, x)-/-FAIL.
+NEQ (x, y).
+LISP (x)-EVAL ((NIL. NIL). (T. TRUE). (F. NIL),
  x, v)-LIGNE-SORTER (v)-LIGNE.
+FIN.-TTY.
-LISP ((LABEL. REV. (LAMBDA. (L. M. NIL).
(COND. ((ATOM. L. NIL). M. NIL).
  (T. (REV. (CDR. L. NIL).(CONS. (CAR. L. NIL). M.
  NIL). NIL). NIL). NIL). NIL). NIL).
(QUOTE. (A. B. C. NIL). NIL). NIL. NIL).
```

図-2 LISP インタプリタ (KF)
Fig. 2 LISP interpreter (KF).

2. ダイアローグ——ホーン集合と二つの証明手続き

以下では，一階述語論理における resolution 理論の術語や形式をベースにする（ただし，独自の形式も導入していく）．

Kowalski の提案は「ホーン集合」をベースにしている．ホーン集合は，たかだか一個の正リテラルしか含まないクローズからなる集合である．そのようなクローズは下のような形式に書ける．ここで，B や A_i はアトム（素項）である．

（H 1） $B \leftarrow A_1 \,\&\, A_2 \,\&\cdots\&\, A_n$.

（H 2） $B \leftarrow$.

(H 3) $\leftarrow A_1 \& A_2 \& \cdots \& A_n$.

(H 4) $\leftarrow$.

ホーン集合をベースにするのは，それについて自然な「手続き的解釈」が成り立つ (Kowalski[1]) からである．H1 の右辺 $A_1 \& \cdots \& A_n$ は左辺 B が成り立つための条件であるが，それを goal-oriented に解釈する．

μ-Planner と対応づければ，H1 は結論型(CONSE)定理文である．H2 は表明文 (ASSERT)，H3 は目標文 (GOAL) である．H4 は「矛盾」を表し，STOP 文と対応づけられる．また resolution の操作は，パターンによる呼び出しと対応づけられる．

ホーン集合の特徴のひとつは，それについて，効率的で完全な証明手続きが存在することである．代表的なものに，SNL (Selective Negative Linear) と SPU (Selective Positive Unit) の二つがある(Kuehner[6])．

SNL は input linear resolution の一種である．証明図は負クローズの連鎖を幹として，resolve する相手はつねに input クローズである．

SPU は，unit resolution の一種で，resolution の片方のクローズはつねに正の単項である．

Selective とは，resolve すべきリテラルの選択法を固定してよい（たとえば順序づけてよい）ということである．

Input resolution と unit resolution は同じ能力を持つ．すなわち，一方で証明できるときは，他方でも証明できる (Chang[8])．一般の一階述語論理に対して，これらは「完全」ではない．しかし，その部分論理系であるホーン集合に対しては完全である．逆に，これらで証明されるものはホーン集合（に書き直し可能）である[6),7)]．

SNL と SPU による証明過程は，きわめて自然に，普通の計算過程と対応づけられる．図-1 は SNL によるもので，再帰的な計算過程と見立てられる．SPU による証明図を図-3 に示す．これは繰り返し的な計算過程と見立てられる．ここで式(F1)～(F3)を次のように書き直しておこう．

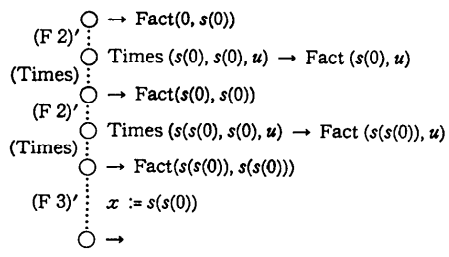

図-3 階乗の計算 (証明)—SPU—

Fig. 3 Factorial computation (proof)—SPU—.

(F1)′ → Fact(0, s(0)).

(F2)′ Fact (x, s) & Times ($s(x)$, s, u) → Fact ($s(x)$, u).

(F3)′ Fact($s(s(0))$, x) →.

ホーン集合に対する証明手続きは，上の二者に限らない．両者の組み合わせ[6)]その他が工夫されている．また，一般の一階述語論理に対して工夫されたものを流用することもできる．たとえば connection graph[11)] を用いた実現例[9)] もある．

SNL と SPU に限るとしてもその証明図の探索法にも各種がある．典型的には，depth-first と breadth-first の二種がある．探索技法の研究では，むしろその両者の組み合わせが課題である．

ホーン集合に限っても，各種の証明手続きがありうるが，ここでは

(i) SNL+depth-first

(ii) SPU+breadth-first

の二種に限ることにし，selection はつねに（書かれた順に）左から右に行われるものとする．PROLOG は (i) がベースになっている．

Kowalski[1)]は，証明手続きの固定化については態度を保留している．とくに，右辺の実行順序については「自由」である方が良いとしている．しかし，その結果，実行順序を制御する補助言語が必要になってくる．一般に，証明システムの研究で，そのような制御用補助言語の必要性がいわれて久しい．にもかかわらず，今に至るも，そのような補助言語について決定的な提案はない．

ということは，「自由」に問題があるのではなかろうか．いわゆるアルゴリズムには，効率的な不自由を求める側面がある．以下の検討のベースを (i) と (ii) に限定してみるのは，証明システムの立場からはきつすぎると見られようが，実在のアルゴリズムがその二つにどのように納まるか，あるいは，はみ出すかを実際に見てみたい，ということなのである．なお，(ii) を入れたのは，いわゆるアルゴリズムでは，繰り返し型の方がむしろ典型的(多くの場合効率的)だからである．

3. カタログ——いくつかのアルゴリズムの分析

3.1 ふたたび Fact——local ホーン構成と tail recursion

goal-oriented な SNL では，(F1)+(F2)が Fact サブルーチンと考えられる．SPU の場合はどうであ

ろうか．(F1)′+(F2)′だけだと証明（計算）の起動のタイミングがはっきりしない．そのままだと，すべての整数についての階乗の表を作ってしまうことになる．起動を goal-oriented にするため，SNL の中に SPU を埋め込むことにする．

(F4) $\mathrm{Fact}(n, m) \leftarrow$

$$\left\{\begin{array}{l} \rightarrow \mathrm{Fact}\,1\,(0, s(0)). \\ \mathrm{Fact}\,1\,(n, m) \rightarrow. \\ \mathrm{Fact}\,1\,(x, v)\ \&\ \mathrm{Time}\,(s(x), v, u) \\ \quad \rightarrow \mathrm{Fact}\,(s(x), u). \end{array}\right\}$$

{ } の中が local に refute されれば，この条件は満足されるものとする．上の式全体はホーンの形にならないが，{ } の中はホーンである．このような構成を local ホーンと呼んでおく．{ } の中が，(この場合) すべての整数について refutable であることが別途保証されていれば (停止性)，これを procedure body と見なすことができる．SPU の動作は，iterative であるから，これを LISP の PROG と関連づけることができよう．ここで，x, v, u は local な loop 変数として現われている．n, m は { } の中では自由変数だが，上のように束縛することにより，procedure 変数となる．

Fact のような簡単なプログラムでは，SNL と SPU の対立は決定的ではない．Fact の代りに ～Fact を独立した述語と考えて書き直し，SNL を用いれば，SPU と同じ動作をする．このとき (F3) に対応するものの変数が自由になる．そこで上のように local ホーンにするのも一法であるが，次のように述語に変数を追加し，述語に情報を運ばせる手がある．

(F5) $\mathrm{Fact}(n, m) \leftarrow \mathrm{Fact}\,2\,(0, s(0), n, m).$
$\mathrm{Fact}\,2\,(n, m, n, m) \leftarrow.$
$\mathrm{Fact}\,2\,(x, v, n, m) \leftarrow \mathrm{Times}\,(s(x), v, u)\ \&$
$\mathrm{Fact}\,2\,(s(x), u, n, m).$

このとき，

(F6) $\mathrm{Fact}\,2\,(x, v, n, m) \leftrightarrow \sim\mathrm{Fact}\,(x, v)\ \&$
$\mathrm{Fact}\,(n, m).$

が成り立っている．これはいわゆる tail recursion の形式であり，recursive procedure で iteration をシミュレートするときよく使われる手法である．しかし，SPU の動作を (tail recursion によって) SNL でつねにシミュレートできるとは限らないし，(F5) の構成の根拠もあらわではない．この場合，(F4) のような SPU による表現の方がより直接的かつ自然であると考えられる．

3.2 フィボナッチ数の計算——新述語の導入，lemma の利用

以下，整数の演算は直接（組み込みで）行われるものとする．定義から，

(Fb 0) $\mathrm{Fb}\,(0, 1) \leftarrow.$
$\mathrm{Fb}\,(1, 1) \leftarrow.$
$\mathrm{Fb}\,(n, v_1+v_2) \leftarrow \mathrm{Fb}\,(n-1, v_1)\ \&$
$\mathrm{Fb}\,(n-2, v_2).$

これを，SNL で処理すれば再帰的な計算になる．SPU (local ホーン) の形式では，

(Fb 1) $\mathrm{Fb}\,(n, v) \leftarrow$

$$\left\{\begin{array}{l} \rightarrow \mathrm{Fb}\,1\,(0, 1). \\ \rightarrow \mathrm{Fb}\,1\,(1, 1). \\ \mathrm{Fb}\,1\,(n, v) \rightarrow. \\ \mathrm{Fb}\,1\,(n_1, v_1)\ \&\ \mathrm{Fb}\,1(n_1+1, v_2) \\ \quad \rightarrow \mathrm{Fb}\,1\,(n_1+2, v_1+v_2). \end{array}\right\}$$

これは iterative に動作するが，tail recursive でないので，unit クローズが蓄積される．また，純 SNL 形式に直接には変換できない．しかし，

$\mathrm{Fb}\,(n_1, v_1)\ \&\ \mathrm{Fb}\,(n_1+1, v_2) \rightarrow \mathrm{Fb}\,(n_1+1, v_2)\ \&$
$\mathrm{Fb}\,(n_1+2, v_1+v_2).$

が成り立つから，新しい述語

$\mathrm{Fb}\,2\,(n_1, v_1, v_2) \equiv \mathrm{Fb}\,(n_1, v_1)\ \&\ \mathrm{Fb}\,(n_1+1, v_2)$

を導入すると，

(Fb 2) $\mathrm{Fb}\,(n, v) \leftarrow$

$$\left\{\begin{array}{l} \rightarrow \mathrm{Fb}\,2\,(0, 1, 1). \\ \mathrm{Fb}\,2\,(n, v, v_1) \rightarrow. \\ \mathrm{Fb}\,2\,(n, v_1, v_2) \rightarrow \mathrm{Fb}\,2\,(n_1+1, v_2, v_1+v_2). \end{array}\right\}$$

と書ける．これは tail recurive であり，SNL にも変換できる．

(Fb 1) のプログラムと同じような動作を SNL で行わせる手段として，associative computation[23] と同じ方法がある．(Fb 0) の第3式の右辺が満足されたとき，左辺の goal を満足させるだけでなく，それに対応する unit assertion を積極的に行っておき，後の証明過程でそれを利用するという方法がある．これは (unit) lemma の利用であり，後の例題でも出てくる有用な方法である．この機能を利用できるようにするため，記号 ⇐ を導入しておく．これは ← と論理的意味は同じであるが，証明動作中の副作用が異なるものである (PROLOG でもこれに相当する機能が用意されている)．

3.3 Reverse プログラム——ガイド，invariant relation

これまでの例題では，SPU の方が好ましいように見える．しかし，次のプログラム (Reverse) を見よう．

(R0) $\mathrm{Rev}(\mathrm{NIL}, \mathrm{NIL}) \leftarrow$.
$\mathrm{Rev}(a+l, m+a) \leftarrow \mathrm{Rev}(l, m)$.

(R1) $\rightarrow \mathrm{Rev}(\mathrm{NIL}, \mathrm{NIL})$.
$\mathrm{Rev}(L, M) \rightarrow$.
$\mathrm{Rev}(l, m) \rightarrow \mathrm{Rev}(a+l, m+a)$.

ここで＋は append の働きをする組み込み演算子，a は1要素のリストを表すものとする．この (R0) は SNL でうまく働く．しかし，これを単純に (R1) と書き換え SPU を用いるのではうまくいかない．なお，ここで二番目の式は目標文で，L, M は入出力変数である．うまくいかないのは，リスト形成の目標が定まらないからである．ところで，述語 $G \equiv \sim\mathrm{Rev}$ による (R2) は，SPU によって，(R0) とまったく同じく，goal-oriented に動作する．これを (R1) と組み合わせて (R3) とする．(R1), (R2) が成り立つから (R3) は成り立つ．

(R2) $\rightarrow G(L, M)$.
$G(\mathrm{NIL}, \mathrm{NIL}) \rightarrow$.
$G(a+l, m+a) \rightarrow G(l, m)$.

(R3) $\rightarrow \mathrm{Rev}(\mathrm{NIL}, \mathrm{NIL}) \,\&\, G(L, M)$.
$\mathrm{Rev}(M, L) \,\&\, G(\mathrm{NIL}, \mathrm{NIL}) \rightarrow$.
$\mathrm{Rev}(l, m) \,\&\, G(a+l', m'+a) \rightarrow \mathrm{Rev}(a+l, m+a) \,\&\, G(l', m')$.

ここで，第3式の両辺において

(R4) $(m'+a)+l = m'+(a+l) \quad (=L)$

が成り立つ．これは，いわゆる invariant relation である．(R4) の条件の下では，Rev と G を合成し，冗長な変数を除去した，

(R5) $\mathrm{R1}(l, l') = \mathrm{Rev}(l, m) \,\&\, G(l', m')$

が導入でき，これによって，繰り返し型の Reverse プログラム

(R6) $\mathrm{Rev}(L, M) \leftarrow \left\{\begin{array}{l}\mathrm{R1}(\mathrm{NIL}, L).\\ \mathrm{R1}(M, \mathrm{NIL}) \rightarrow.\\ \mathrm{R1}(l, a+l') \rightarrow \mathrm{R1}(a+l, l').\end{array}\right\}$

が得られる．この式の中で G は，リスト形成の方向を導くガイドの役割をしている．これについては，後でもっと explicit な例がでてくる．

3.4 パージング・アルゴリズム (1)——トップダウンとボトムアップ

以降，文脈自由文法に対するパージング・アルゴリズムを例題にする．簡単のため，生成規則はチョムスキ標準形 ($A \rightarrow w$ または $A \rightarrow BC$) に限ることにする．いくつかの述語を導入する．$I(w, i, j)$ は，入力系列の中の位置 i と j の間に w の語が存在することを表わす．$D(A, i, j)$ は i と j の間の記号列がカテゴリ A から生成される，あるいは A にパーズされる，ということを表す．P1 と P2 では生成規則の存在，入力記号列の存在に関する表明が別途与えられているものとする．パージングは，目標 $\leftarrow D(s, 0, n)$ を達成する動作である．

(P1) $D(A, i, j) \leftarrow \mathrm{P1}(A, w) \,\&\, I(w, i, j)$.
$\leftarrow \mathrm{P2}(A, B, C) \,\&\, D(B, i, k) \,\&\, D(c, k, j)$.

これを SNL で処理すれば，いわゆるトップダウン・パージングになる (図-4).

(P2) $I(w, i, j) \,\&\, \mathrm{P1}(A, w) \rightarrow D(A, i, j)$.
$D(C, k, j) \,\&\, \mathrm{P2}(A, B, C) \,\&\, D(B, i, k) \rightarrow D(A, i, j)$.

を SPU で処理すればボトムアップ (Cocke-Younger-Kasami) アルゴリズムと同じに動作する．(P1), (P2) は生成規則の定義そのものと見られる．動作上は，

$$\left\{\begin{array}{l}\text{top-down, serial}\\ \text{backtracking}\end{array}\right\} \leftrightarrow \left\{\begin{array}{l}\text{SNL}\\ \text{depth-first}\end{array}\right\}$$

$$\left\{\begin{array}{l}\text{bottom-up, parallel}\\ \text{WFS table}\end{array}\right\} \leftrightarrow \left\{\begin{array}{l}\text{SPU}\\ \text{breadth-first}\end{array}\right\}$$

の対応がある．このように，パージング動作 (アルゴリズム) は，証明手続きの選択による．より複雑な証明手続きは，また別のパージングに対応する．これらはすでに，参考文献 10), 11), 12) などで扱われている．

しかし，より高級な証明手続き (たとえば双方向の証明手続きからは双方向のパージングが得られる) が，具体的にどのパージング・アルゴリズムに対応するかは明らかにされていない．アルゴリズム的な側面が，証明手続きの中にかくされてしまっている．

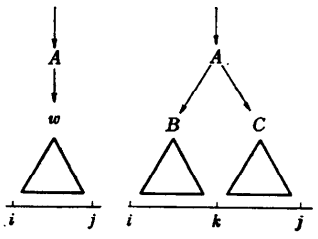

図-4 トップダウン・パージング
Fig. 4 Top-down parsing.

3.5 パージング・アルゴリズム(2)——Earley のアルゴリズム

より高級なアルゴリズムとして，Earley のアルゴリズム[13),14)]を分析してみよう．このアルゴリズムは，"item" の操作によって記述されている．それを（チョムスキ標準形に縮約して）述語形式で書き直す．その結果は，

(P3)

$I(w,i,j)\ \&\ G(w,i)\ \&\ \mathrm{P1}(A,w) \rightarrow D(A,i,j).$

$D(C,k,j)\ \&\ G(C,k)\ \&\ \mathrm{P2}(A,B,C)\ \&\ D(B,i,k) \rightarrow D(A,i,j).$

$D(B,k,i)\ \&\ G(B,k)\ \&\ \mathrm{P2}(A,B,C) \rightarrow G(C,i).$

$\rightarrow G(s,0).$

$G(A,i)\ \&\ \mathrm{P1}(A,w) \rightarrow G(w,i).$

$G(A,i)\ \&\ \mathrm{P2}(A,B,C) \rightarrow G(B,i).$

となる．Earley の item との対応は，

$[A \rightarrow a:\mathrm{Bb},i] \in I_j \equiv G(B,j)\ \&\ D(a,i,j)$

$[A \rightarrow a:,i] \in I_j \equiv D(A,i,j)$

のようにしたが，(P3)自身が Earley のアルゴリズムの本質をよく示していると考えられる．これを(P2)と比べられたい．G が挿入されている点だけが異なる．この G はガイド（あるいはゴール）の役割をしている．G の部分を(P1)と比較すれば，それを簡約化した形になっていて，goal-oriented な動作を（(P1)とまったく同じではなく）近似的に行うことが分かる．(P3)はこのように双方向の探索（パージング）を実現している(図-5)．

しかも，この双方向性は，証明手続きの内部にかくされているのではなく（ここでの証明手続きは単方向の SPU である）表層に explicit に表現されている．ここでは，Gのような補助述語がより高級な証明手続きの代行をしている．

ところで(P3)の正当性は，パージングについての帰納法で証明できる．これを参考文献 15) の扱いと比較されたい．

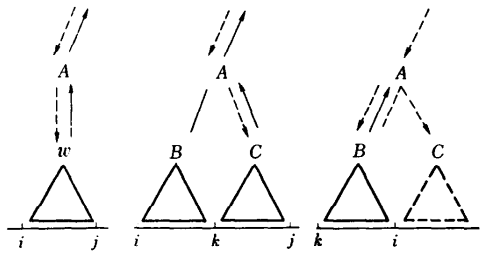

図-5 双方向パージング
Fig. 5 Bidirectional parsing.

Earley のアルゴリズムを改良したものに Pratt のもの[16),17)]がある．(P3)の第4～5式は，入力記号列と独立である．そこで，これを前もって解き，「推移性」$R(A,B)$ のテーブルを作っておいてそれを利用する．また述語Gにそれを生成した規則に関する情報を運ばせる（Earley の item にもその機能がある）．

(P4)

$I(w,i,j)\ \&\ \mathrm{P1}(A,w)\ \&\ \mathrm{G1}(A,i) \rightarrow D(A,i,j).$

$D(C,k,j)\ \&\ G(C,k,A,i) \rightarrow D(A,i,j).$

$D(B,k,i)\ \&\ \mathrm{P2}(A,B,C)\ \&\ G1(A,k) \rightarrow G(C,i,A,k).$

$G1(A,i) \leftarrow G(D,i,E,k)\ \&\ R(D,A).$

3.6 パージング・アルゴリズム(3)——トップダウン WFST パーザ

一般に，ボトムアップ・パージングの速度は n^3 で押えられ，トップダウンの方は指数関数的であるとされる．トップダウン・パージングの改良には，WFS (well formed substring) テーブルの利用が提案されている．これは前述した lemma の利用に相当している．この場合速度では n^3 で押えられる[18)]．

これを，ここでの形式で書けば次のようになる．

(P5)

$D(A,i,j) \Leftarrow I(w,i,j)\ \&\ \mathrm{P1}(A,w).$

$D(A,i,j) \Leftarrow D(C,k,j)\ \&\ \mathrm{P2}(A,B,C)\ \&\ D(B,i,k).$

4. アナログ——他の形式との類似

4.1 SNL と帰納方程式の比較——述語計算と関数計算，shared expression

Fact のようなプログラムは，述語形式で関数値の計算をするものである．LISP などは関数計算をベースにしているが，もともと関数計算と述語計算は本質

$$F(x,y) \leftarrow \begin{bmatrix} x=0\ \&\ y=0 \\ x=0\ \&\ F(x-1,z)\ \&\ T(z,x,y) \end{bmatrix}$$

$$F(x,y) \equiv y=f(x)$$

$$T(z,x,y) \equiv y=z*x$$

$$y=f(x) \leftarrow \begin{bmatrix} x=0\ \&\ y=0 \\ x=0\ \&\ z=f(x-1)\ \&\ y=z*x \end{bmatrix}$$

$$y=f(x) \leftarrow \begin{bmatrix} x=0\ \&\ y=0 \\ x=0\ \&\ y=f(x-1)*x \end{bmatrix}$$

$$f(x) = \begin{bmatrix} x=0:\ 0 \\ x=0:\ f(x-1)*x \end{bmatrix}$$

図-6 述語計算から関数計算への移行
Fig. 6 From predicate calculus to functional calculus.

的に異なるものではない．それを示す一例が**図-6** である．

ここで，形式を似せるためと，後での有用性のために，alternative を表わす［ ］を導入する．ここでは，順序づけ（リテラルの順序，式の順序）を前提としているから，そのような shared expression は自然に導入され，表現上だけでなく，処理上も有用である．また，関数表現に移行するため，等号＝を導入する．二番目の変換では等号の性質を用いている．最後は LISP（帰納方程式）と同形の表現となる．

関数表現が与えられたとき，それを述語表現に変換することも同様にできる．

4.2 結合グラフと流れ図式

クローズ相互間の関連を示すのに結合グラフ (connection graph[11], clause interconnectivity graph[20]) が提案されている．これは，証明手続きを，グラフ操作に還元しようとするものである．関連リンクには，mgu による代入リストが附随させられる．**図-7** はその一例である．

図-7 の左に，階乗計算の流れ図を重ねて見られたい．この例は流れ図式理論と述語論理的プログラミングの関連を示唆するとともに，述語論理プログラムの「コンパイリング」を示唆していると思われる．クローズは流れ図のノード（ラベル）に，リンクはアクションに対応し，双対なグラフになっていると見られる．

5. モノローグ——EPILOG の提案，問題提起

これまでの検討から，プログラミング（アルゴリズム表現）に適した述語論理形式として以下のようなものが浮び上がってくると思われる．

（1） リテラルの配列，クローズの配列について，「順序づけ」を前提とする．

順序づけは不自由に見えるが，逆に，最適化に関する知識を固定する役割も果たす．しかし，順序づけによって，論理的意味は変化しないことに注意しよう．

（2） 順序づけをベースに shared expression を許す．

そのひとつは alternative［ ］である．また，結論部に & による結合を許す（これによってホーンの性質は乱されない）．

（3） SNLと，local ホーンな SPU の二つをベースにする．

（4） 各種の implication operator を導入する．

考えられるものは←, →, ↔, ⇐, ⇒, ⇔ である．後三者は lemma の利用を許すものとする．これらは論理的意味を変えるものではない．

（5） 組み込み述語，組み込み関数を用意する．

これらは基本的操作を組み込むものであるが，さらに拡張機能を (PROLOG のように) 実現するてだてともなる．拡張機能には，データベース機能，バックトラックなどの制御機能，ancester 操作などの（ホーン集合からの）拡張機能，高階操作機能が考えられる．(PROLOG にもいくつか用意されている．)

（6） タイプ述語，delay 機能[19]，関数定義も考えられる．

（7） syntactic sugar を用意する．Operator 表現 (PROLOG) など，表層構造上のサービスを考える．

以上のような論理形式はどうであろうか．これを実現した形式をここでかりに EPILOG と名付けておく．

EPILOG を実際に実現し，またプログラミングの実際に用いるには，まだ検討すべき問題が沢山ある．一つは拡張上の問題である．述語論理的プログラミン

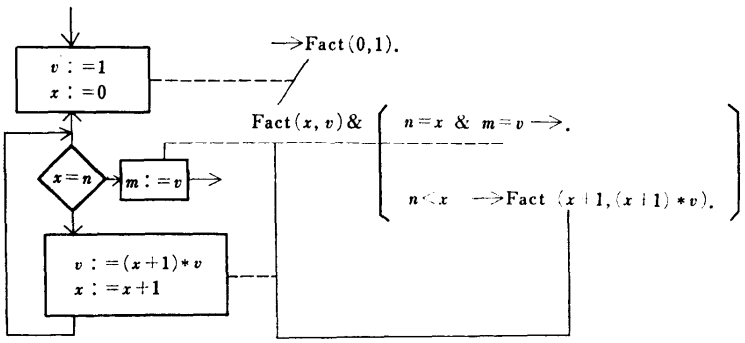

図-7 （流れ図と）結合グラフ
Fig. 7 (Flow charts and) connection graphs.

グの利点のひとつは，semantics の良さにある．ホーン集合での SNL, SPU はアルゴリズム表現に適した性質をもっている．これらの利点を保存しつつ拡張していくにはどうしたらよいだろうか．拡張のひとつは，一般の一階述語論理への拡張である．それに対して工夫された数々のストラテジーを explicit に組み込むにはどうしたらよいか．そのとき理論的にスマートであることが望ましい．拡張のもう一つは高階論理への方向である．そのとき，たとえば，高階のホーン集合は理論的にどんな性質をもっているのだろうか．また，様相論理，直観主義論理との関連はどうなるだろうか．

処理系作製上は，上の検討である程度明らかなように，LISP 技術で間に合いそうである．tail recursion (loop) の処理には注意が要るだろう．array の導入はどうしたらよいであろうか．

プログラミングの実際に用いるには，他にどのような要素が必要であろうか．通常のプログラミング言語との比較をもっと進める必要がある．また，いわゆる「プログラム理論」との関連はどうなるだろうか．また，プログラムの自動合成（あるいは自動改良）にどのように役立つであろうか．また，ガイドのメカニズムやサブセオリーはどのように発見したらよいであろうか．

6. エピローグ

個人的ないきさつを書く．Kowalski[1]の提案は以前から知っていたが，とくに興味をひかれたわけではなかった．その後，必要があって Early/Pratt のアルゴリズムを分析する機会があった．それが論理プログラミングを有望なアプローチと感じるようになったきっかけである．(もっとも Kowalski[1]と逆の主張もある．)ここに述べた事項は，検討の時間的順序と逆の配列になっている．PROLOG については，ごく最近その詳細を知りえた．良くできたシステムと思うとともに，それまで検討していたことにまだ意義があるとも感じた．

検討すべき問題は，上述のように，まだ多いが，論理プログラミングは有望なアプローチのひとつと思われる．諸賢のご協力をあおぎたい．

最後に，PROLOG に関する詳細な情報をもたらした古川康一君に感謝する．彼は討論の良き相手であり，ここでの検討を激励された．また，同じ意味で筆者の推論機構研究室の諸君に感謝する．

参考文献

1) Kowalski, R.: A Predicate Logic as Programming Language, Proc. IFIPC, pp. 568-574 (1974).
2) Van Emden, M. H. and Kowalski, R. A.: The Semantics of Predicate Logic as a Programming Language, J. ACM, 23-4, pp. 737-742 (Oct. 1976).
3) Battani, G. and Moloni, H.: Interpreteur du Language de Programmation PROLOG, R. Groupe d'intelligence Artificielle, Univ. d'Aix-Marseille.
4) Warren, D. H. D.: WARPLAN: A System for Generating Plans, DCL Memo 76, Dept. A. I., Univ. of Edinburgh (1974).
5) Kanoui, H.: Some Aspects of Symbolic Integration via Predicate Logic Programming, ACM SIGSAM pp. 29-42 (Nov. 1976).
6) Kuehner, D.: Some Special Purpose Resolution Systems, in Machine Intelligence, Vol. 7, pp. 117-126 (1970).
7) Henschen, L. and Wos. L.: Unit Refutation and Horn Sets, J. ACM, 21-4, pp. 590-605 (Oct. 1974).
8) Chang, C. L.: The Unit Proof and the Input Proof in Theorem Proving, J. ACM. 17-4. pp. 698-707 (Oct. 1970).
9) Tarnlund, S. A.: An Interpreter for the Programming Language Predicate Logic, Proc. IJCAI 4, pp. 601-608 (1975).
10) Vander Brug, G. J. and Minker, J.: State Space, Problem Reduction and Theorem Proving-Some Relationships, C. ACM, 18-2, pp. 107-115 (Feb. 1975).
11) Kowalski, R.: A Proof Procedure Using Connection Graphs, J. ACM, 22-4, pp. 572-595 (Oct. 1975).
12) 西岡，打浪，手塚：導出原理を用いた構文解析システム—構文解析手続きの公理化，信学会研究会資料 AL 76-64, pp. 57-66 (1976).
13) Earley, J.: An Efficient Context-Free Parsing Algorithm, C. ACM, 13-2, pp. 94-102 (1970).
14) Ullman, J. D.: Application of Language Theory to Compiler Design, in Currents in the Theory of Computing, pp. 173-218 (1973).
15) Jones, C. B.: Formal Development of Correct Algorithms: An Example Based on Earley's Recognizer, Proc. ACM Conf. on Proving Assertions about Programs, pp. 150-169 (Jan. 1972).
16) Pratt, V. R.: LINGOL-A Progress Report, Proc. IJCAI 4, pp. 422-428 (1975).
17) Pratt, V. R.: The LINGOL System, Rep. AI

Lab. MIT (Feb. 1975).
18) Grishman, R.: A Survey of Syntactic Analysis Procedures for Natural Language, AJCL Microfiche 47 (1976).
19) Hewitt, C.: Viewing Control Structure as Patterns of Passing Messages, WP 92, AI Lab. MIT (1976).
20) Sickel, S.: A Search Technique for Clause Interconnectivity Graphs, IEEE Tr. on Computers, 25-8, pp. 823-835 (Aug. 1976).
21) Ashcroft, E. A. and Wadge, W. W.: Lucid, a Nonprocedural Language with Iteration, CACM Vol. 20, No. 7, pp. 519-526 (July 1977).
22) Sussman, G. J. et al.: Micro-Planner Reference Manual, AI Memo 203 a,AI Lab. MIT (Dec. 1971).
23) Goto, E. and Kanada, Y.: Hashing Lemmas on Time Complexities with Applications to Formula Manipulation, Proc. SYMSAC (1976).

(昭和60年7月1日受付)