総　論

Prolog 総 論†

古川　康一††

1. はじめに

プログラミング言語 Prolog がその産声をあげたのは，今から10年以上も前，1971年である．場所は，フランス南部の港マルセーユに隣接したルミニという所にあるマルセーユ大学である．当時カナダでの翻訳プロジェクトに加わっていた Alan Colmerauer は，フランスに戻ると，構文解析の研究を継続して行い，試行錯誤過程を組み込んだ巧妙な構文解析プログラムを開発した．それは SYSTEM Q と名付けられたが，その動作が述語論理における証明過程として説明できることから，後に Prolog と改名された．その名前の命名者は，Colmerauer 自身という説と，彼の同僚であった Philippe Roussel という説，あるいは Roussel 夫人という説もあり，定かではない[1]．風説によれば，パーティで議論しているうちに作られたということである．

Colmerauer と並んで Prolog を語るときに欠かすことができないのが Robert Kowalski である．彼は Horn 節に限定したときの input resolution による証明過程と通常の計算機言語における計算過程を対比させ，Prolog の計算機言語としての明解な解釈を与えた[2]．この解釈により，Prolog の2通りの読み方，すなわち宣言的読み方と手続き的読み方が確立した．

Prolog がリスト処理の高級言語であることは，今では良く知られているが，Prolog が生まれてからリスト処理のごく基本的プログラムである append プログラムが発見されるまで，3カ月を要したと聞いている．実は Prolog による構文解析においては，通常のリストよりも巧妙な difference list (dlist, 重リスト) が用いられており，その中に Prolog にとって最も重要な概念である incomplete data structure (不完全データ構造) の考えが使われている．そして，この概念の重要性が認識されるまで，さらに数年を要している．正直に言って，筆者自身がその真の重要性を認識したのは，つい最近 (約1年前) のことである．LISP と比べて Prolog がリスト処理の能力で真により優れている点は，不完全データ構造が扱える点である．

Prolog は，このようにそれ自身が秘めている潜在能力に比べて，これまで過小評価されつづけてきた言語であると言えよう．Prolog が LISP に取って替わり得るリスト処理言語であるとの認識は，わが国での第5世代コンピュータ・プロジェクト以前にはまったくなかったものである．

Prolog の価値を高めた仕事として忘れてはならないものに，David Warren 等による処理系の実用化の研究がある[3]．彼等の研究により，マルセーユの処理系と比べて処理系の速度が15～20倍，必要とする作業領域がほぼ10分の1という驚くべき改善を見た．速度の向上は，ユニフィケーションのコンパイルによっている．Horn 節の頭部にあたる手続き宣言部のパターンに応じて，それが呼び出されるときにどのようなユニフィケーションがなされるかを解析し，コンパイル化を図った．作業領域の縮小は，スタックの管理およびテール・リカージョン呼出しの最適化による．Warren は変数によってはすぐに不要となるものが存在することに目をつけ，ローカル用，グローバル用の2つのスタックを用意し，早くたたんでよいものとそうしてはいけないものを別々に管理した．処理系の改良は，実用的観点から見ると大変重要である．処理速度および所要作業領域量の向上によって初めて Prolog が実用プログラミング言語となったといえよう．

Prolog に対する過小評価は，アメリカ，とくに MIT の Artificial Intelligence Laboratory (AI ラボ) において顕著であった．それは主に誤解に基づくものであった．MIT の AI ラボでは，Carl Hewitt が，Prolog ができる直前に述語論理の手続き的解釈に基づく言語 PLANNER を開発している[4]．PLANNER サブセットである μPLANNER の処理系が LISP 上

† Overview of Prolog by Koichi FURUKAWA (Institute for New Generation Computer Technology).

†† (財)新世代コンピュータ技術開発機構

に作られ，それは Winograd のつみきの世界についての自然言語理解システムを開発するのに使われた．μPLANNER 自身は問題解決機として使われ，細かいリスト処理は親言語の LISP にまかされた．このように，MIT の研究者から見れば，少なくとも Kowalski の仕事は彼等の仕事の二番煎じと映ったわけである．そのこと自身は基本的には正しいかもしれないが，Prolog は μPLANNER と異なり，それ自身で閉じたリスト処理言語であるという認識が MIT の研究者たちにはなかったと思われる．もう1つの不幸は，アメリカの AI 研究者全般に見られる論理学に対する奇妙な不信，偏見である．それは，彼等が AI 研究の初期に経験した失望と深く係り合っている．AI 研究の初期には，汎用の問題解決機を開発したいという強い願望があって，その有力な候補に一階述語論理の定理証明機が考えられた．しかし，それは問題が少しでも複雑になるとまったく役立たないことが判明し，「述語論理は AI では役に立たない」という迷信ができあがってしまったようである．

最近になって，この誤解は徐々に解けてきたようである．今年2月にアメリカ東海岸のアトランティック・シティで開かれた "1984 International Symposium on Logic Programming" には，約400名の参加者があったとのことである．

上記のシンポジウムでもそうであるが，近年，アーキテクチャの研究者が論理プログラミングに大きな関心を寄せている．それは，並列アーキテクチャの可能性を追求するためである．

以下，本稿では，**2.** でリスト処理言語としての Prolog，とくに不完全データ構造の役割について述べ，**3.** では，Prolog における Logic の役割について述べたい．**4.** では，Prolog の拡張として並列性の導入の重要性を指摘したい．おわりに，今後の研究方向について述べてみたい．

2. リスト処理言語としての Prolog

Prolog によるリスト処理の詳細は，本特集号の他稿に譲るが，本章では，単純な例によって，不完全データ構造の重要性を指摘したい．例は，よく知られたリストの平坦化プログラム flatten (L, F) である．たとえば，

```
?-flatten ([[a, b], c, [d], e], F).
```

とすると，F=[a, b, c, d, e] となる．

重リストを使った Prolog プログラムを d-flatten とすると，flatten, d-flatten の定義は，つぎのようになる（以下のプログラムは，出力中に不要な空リスト [ ] が入ってしまうが，話を単純にするために，ここではそれを抑制するプログラム部分を省略した）．

```
flatten (L, F) :-
        d-flatten (L, d(F, [ ])).

d-flatten ([X|Y], d(U 1, U 3)) :-
    d-flatten (X, d(U 1, U 2)),
    d-flatten (Y, d(U 2, U 3)).
d-flatten (X, d([X|U], U)).
```

ここで，d(U, V) が重リストである．重リストは，一本のリストを2つのポインタで指し，その2つのポインタにはさまれた差分 (difference) から成るリストを表しているものと解釈する．そうすると，たとえば，d([a, b, c|U], U) は，たとえ U が未知の変数であっても，重リストとしては，リスト [a, b, c] を表していることになる．

さて，d-flatten プログラムは，[X|Y] というリストを平坦化するのに，まず X の部分を平坦化して重リスト d(U 1, U 2) を作る．例題に適用した場合，これは d([a, b|U 2], U 2) となる．ついで，Y の部分を平坦化し，重リスト d(U 2, U 3) を得る．ここで，U 2=[c, d, e|U 3] のようになるが，このとき，U[1=[a, b|U 2]=[a, b, c, d, e|U 3] となり，重リスト d(U 1, U 3) は結局 [a, b, c, d, e] となる．

さて，このプログラムの計算量を求めると，それはデータ数 n に比例していることがすぐに分かる．

一方，上のプログラムに匹敵するような LISP プログラムは，つぎのようなものであろう．

```
(DEFUN FLATTEN (X)
  (COND ((ATOM X) (LIST X))
        (T(APPEND (FLATTEN (CAR X))
                  (FLATTEN (CDR X))))))
```

このプログラムは，リストの先頭と残りを平坦化して，後でその2つのリストをつなげている．この最後の結合操作のために，計算量が $O(n \log n)$ になってしまう．

実は LISP でも $O(n)$ の平坦化プログラムを書くことができる．それはつぎのようなものである*．

```
(DEFUN FLATTEN (X)
        (FLAT X NIL))
```

* 本プログラムは，NEC の上田和紀氏による．

```
(DEFUN FLAT (X REST)
  (COND ((ATOM X) (CONS X REST))
        (T (FLAT (CAR X)
                 (FLAT (CDR X) REST))))))
```

Prolog の d-flatten に相当するのが FLAT プログラムであるが，この2つのプログラムは，動き方がまったく異なる．d-flatten は，平坦化をリストの先頭から進めて行くが，FLAT の方は，リストの右端から平坦化を始める．FLAT プログラムを解読するのは大変困難である．読者は試みられよ．

d_flatten プログラムの簡潔性は重リストの使用に負っている．d_flatten プログラムで，リスト [X|Y] の先頭部分 X を平坦化する部分に注目すると，そこでは，重リスト d(U1, U2) を作成している．今の場合，U1=[a, b|U2] である．ところで，リスト U2 は，実はこの時点では求められていない．U2 は，リストの残りの平坦化によって初めて求められる．すなわち，プログラムの前半では，U2 が未知のままの不完全データ構造d (U1, U2) を作ったことになる．そして，後で U2 が作られるわけである．もしも，扱うデータ構造が未知の変数を含むことができなければ，データ構造の作成手順は，その構造によって決められてしまう．その結果が FLAT プログラムである．このように，不完全データ構造の導入は，データ構造の作成手順に大きな自由度を与えるものといえる．

この機能をうまく使っているのが Definite Clause Grammer による構文解析である．構文解析は，平坦化の逆で，平坦な入力 (語の列) を構造化するものであるが，構文解析木を作る際に不完全データ構造を使っている．

その他にも，いくつか例がある．Prolog による待ち行列の実現，木の幅優先探索プログラムなどである．この機能に着目したより高度なプログラミング技法も，いずれ出現するであろう．

不完全データ構造は，lazy evaluation の考えとも関連している．lazy evaluation が時間軸上でデータの加工の柔軟性を獲得しているのに比べて，不完全データ構造は，空間表現 (d(U1, U2) という記述) によって，この柔軟性を得ている．

3. Prolog における Logic の役割

Prolog における Logic の役割は，2つに分けて論じることができると思われる．第1は，計算過程が証明過程になっていることに由来するもので，第2はユニフィケーションに由来するものである．以下，その2つの役割について別々に論じよう．

3.1 入力導出証明機としての役割

Prolog の実行過程がホーン節の証明過程となっているからといって，プログラムの検証が不要であることにはならないが，少なくとも停止性以外の部分正当性は，述語論理の枠組での証明が比較的容易である．その他にも，プログラミング・システムの能力を向上させる種々の工夫が可能である．それらは，アルゴリズミック・デバッギング[5]，プログラム変換に基づくプログラムの最適化などである[6),7)]．

証明機としての Prolog の特質を最大限に生かしているのが，データベースへの応用である．データベースへの検索が証明過程そのものになっていることはいうまでもない．その他，データベースの一貫性管理に関する興味ある研究が，最近随所で行われているが，それは，一貫性維持のための制約条件をホーン節で表すことができれば，Prolog インタプリタを直接制約条件検証のために利用できることに依存している[8)-10)]．

証明機そのものではないが，ロジックとの関連で興味ある研究に，帰納的推論に基づいたモデル推論システムの研究がある[5)]．それは Prolog プログラムに適用することによって，プログラム合成，文法推定および概念形成がまったく同じ枠組で達成されることが示されている．さらにその枠組と一貫性管理の枠組を組み合わせることによって，信念と呼ばれる不確かな知識を含む系において正しい入力にともなって発生する信念の翻意を含む推論根拠管理システムが実現できることが最近分かってきた[11)]．

Prolog の処理系がホーン節の定理証明機となっていることは，以上に述べたように，多くの利点を生んでいる．Prolog に対して，実用的な観点などからいろいろな拡張が提案されているが，その中からどの案を採用するかを判断するための基準として最も大切なものはそれがロジックとしての性質を失わないことである．すなわち，プログラムの実行結果は，プログラムを公理の集合と考えたときの論理的な帰結となっていることが必要であり，それは言語を拡張しても成り立っていなければならない重要な性質である．

3.2 ユニフィケーションの役割

ユニフィケーションは，2. で述べたリスト処理を実現するための基本的な仕掛けである．Prolog では，LISP のようなリスト処理基本演算群(CONS, CAR, CDR) を必要としない．それらはすべてリスト型の引

数間のユニフィケーションによって達成される．また，ユニフィケーションが，手続きの呼出し元と呼出し先で引数を対等に扱っているので，呼出し元のデータ構造に変数が含まれていてもよく，その結果，2.で述べた不完全データ構造の扱いが可能となっている．

ユニフィケーション操作は，このようにリスト処理における基本操作となっていることが分かる．これは数値計算における四則演算に相当するものである．すなわち，リスト処理に適したコンピュータを開発する場合，ユニフィケーションは，アーキテクチャの設計のための仕様となると考えてよかろう．この意味で，ロジックは，むしろアーキテクチャ研究にとっての指導原理を与えているといえる．近年，論理型プログラミング言語のための並列コンピュータのアーキテクチャ研究が，国際的にも非常に盛んになってきていることがこのことを物語っている．

4. 並行 Prolog

Prolog はロジックを基にしていることから，データベースの検索などの，変化しない情報を扱うことは得意であるが，状態の変化の取扱いは苦手である．とくに，オブジェクト指向プログラミングに見られるような独立して変化するオブジェクトが複数存在するような系を表現することは大変難しい．

ロジックの枠組をこわさずにこのような機能を実現するためには，ロジックの枠内で並行プログラミング機能を実現すればよいことが知られている．そして，実際にその性質を満足する言語として，Relational Language[12]，Parlog[13]，Concurrent Prolog[14] などが提案されている．本稿では，これらの言語を総称してストリーム並列 Prolog と呼ぶことにしよう．

これらの言語の詳細については，本特集号の他稿を参照していただくこととし，本稿ではストリーム並列 Prolog のプログラミングにおける基本的考え方を述べたい．

ストリーム並列 Prolog での並列プロセスは，Prolog のゴールに現れる複数リテラルの（擬似）並列実行が対応していると考えてよい．そして，これらの複数プロセス同士は，それらの中に同時に出現する論理変数（共有論理変数）によって関連づけられている．すなわち，共有論理変数をプロセス間の通信を実現するために利用するわけである．ここで，連続通信をどう実現するか，信号の送り手，受け手をどのようにして指定するか，などの問題を解決しなければならない．連続通信は，信号のストリーム化，すなわち，現在から未来にいたるまでの信号の時系列を1つの変数によって表すことによって達成される．そのような方法による通信法をストリーム通信と呼ぶ．信号の送り手は，信号の生産者とも呼ばれ，受け手は消費者とも呼ばれる．生産者，消費者の区別を行う方法は各言語によって異なるが，区別をすることには変わりがない．

ストリーム通信によって送受する信号には，大きく分けてデータと指令とがある．データをストリームとするプログラムは，リスト処理のパイプライン的な並行処理が可能であり，プロセッサが実際に複数個存在すれば並列処理も可能である．

一方，指令の列をストリームとして送るプログラムは，オブジェクト指向プログラムとなる．すなわち，指令（メッセージ）は他プロセス（オブジェクト）に対する処理の要求であり，メッセージを受け取ったオブジェクトは，自分のところに定義されたメッセージの処理手順に従って要求された処理を行う．

オブジェクト指向プログラミングをストリーム並列 Prolog で実現するときの利点の1つは，不完全データ構造をメッセージとして用いることによって，要求に対する返答を簡潔に表現できる点である．すなわち，要求メッセージの一部分を返答用の変数とするわけである．

もう1つの利点は，処理の並列化である．もし，ストリーム並列 Prolog を並列実行するコンピュータが作られれば，その上でオブジェクト指向プログラムが直ちに並列に実行されることになる．オブジェクト指向プログラミングは，問題解決のレベルで処理を並列に動作し得るオブジェクト群に分配するのに適した方法論となっており，それをリスト処理のレベルでの並列化と組み合わせることによって，応用プログラムからより多くの並列性を引き出すことが可能となる．そのため，この両者の自然な結合の重要性は大変大きいと考えられる．

ストリーム並列 Prolog は，ロジックの点から見ると，Prolog よりもさらに制約が強く，その有用性が大きく損なわれているのではないかという批判がある．失った機能の1つは，プログラムの双方向的な動きである．Prolog では変数は入出力両用が可能であり，その役割を逆転させることによってまったく異なる働きをすることになる．たとえば，日英機械翻訳プログラムを双方向性を失わないように作る（カットを

用いないで）と，それは英日機械翻訳プログラムとしても働くわけである．しかしながら，そのようなプログラムは例外的なものであり，むしろ，大規模な実用プログラムでは，このような性質はすでに失われている例がほとんどであると思われ，この機能の喪失は，実用的な観点からは，あまり問題とならないであろう．むしろ，並列プログラミング機能を獲得することによる表現力の増大は，そのような損失を補って余りあるといえよう．

並列実行環境においては，純粋 Prolog とストリーム並列 Prolog は，その役割が異なってくる．純粋 Prolog (Prolog から，カットおよびデータベース操作述語 (assert, retract など) を除いたもの) は，データベース検索言語と考えられる．

一方，ストリーム並列 Prolog は，そのまま並列処理が可能であり，すでに述べたようにリスト処理やオブジェクト指向プログラミングなどが可能な汎用のアルゴリズム記述言語といえる．すなわち，並列実行環境では，むしろ，ストリーム並列 Prolog の方がより基本的なプログラミング言語となると考えられる．

5. おわりに

前章でも述べたように，Prolog の拡張方向の1つは並行処理の導入である．本章では，それ以外の有望な拡張の方向を述べてみたい．

第1の方向は Prolog Ⅱ[15] が進んでいる方向である．それは，逐次実行の範囲で Prolog をより純粋 Prolog に近づける努力である．Prolog Ⅱ では，ユニフィケーション時の occur check の問題，すなわち，たとえば p(X, X) と p(Y, f(Y)) を同一化しようとするときに，X と f(X) の同一化を禁止するために構造 f の中に出現する変数を調べなければならない問題を，そのような同一化に逆に意味をもたせて回避している．X と f(X) の同一化から生ずる結果は，f(f(f(…(X)…))) の形の無限構造であると考える．そして，そのような構造をプログラミングに役立てることを考えている．Prolog Ⅱ では，この他に引数の値が決まるまで実行を遅らせる freeze 機能と，それを利用した不等号の処理を導入しており，それによって制御構造を豊富にし，カットを除去する試みをしている．

第2の方向は，等号の導入である．最近，同一化と項書替えの2つをうまく融合した演算を定義して，論理型言語と関数型言語の両方の長所を兼ね備えた言語を設計する試みがいくつか発表されている．この融合により，言語の記述力は飛躍的に向上し，数式処理の単純な公式程度ならそのままプログラムとして記述できる．ただし，この種の言語で書かれたプログラムは，十分効率よく実行できるかどうかは確かではない．

第3の方向は，プログラムの構造化を含んだオブジェクト指向プログラミングのためのサポート機能である．これは，前章で述べたようにストリーム並列 Prolog とも関連している．プログラムの構造化に着目して Prolog にオブジェクト指向プログラミング機能を付加した言語 ESP[16] (Extended Selfcontained Prolog) が ICOT で開発されている．ESP では，状態の変化はデータベースの更新機能を用いて実現している点がストリーム並列 Prolog によるオブジェクト指向化と異なる．ストリーム並列 Prolog にプログラムの構造化をもち込んだ言語 Mandala が，同じく ICOT で開発中である．Mandala は，ESP の並列版であると同時に知識プログラミング・システムとしての強力かつ柔軟な表現力をもたせることもねらっている．

第4の方向は，処理の高速化，効率化のための工夫である．データ構造に関しては，論理的性質を保ったままで効率よく操作できる配列の実現法が検討されている．制御構造に関しては，ユーザ定義のインタプリタ内で制御情報を解釈する方式や，ホーン節の探索順序にもっと自由度をもたせ，深さ優先以外に幅優先，最適探索などが可能となるような処理系の方式も提案されている．

Prolog は万能ではないが，一般に認識されているよりもずっと強力なプログラミング言語であると言えよう．現在では処理系の整備も徐々に進んでゆき，国内の汎用大型機の上で走る処理系の実現もそう遠くないと思われる．また ICOT で開発中の Prolog マシン PSI[17),18)] も，数年のうちには一般の利用に供し得るようになる可能性もある．さらに，PC 9800 上で高速で走る Prolog の処理系 (Prolog/KABA) も開発されている．プログラミング言語の普及は，多くの人が使ってみることによってはじめて達成される．言語は文化そのものである．Prolog あるいは，その拡張であるストリーム並列 Prolog のユーザが増加し，その人々によって独自の文化が育成されていくことを期待したい．

参 考 文 献

1) Robinson, J. A.: Logic Programming—Past,

Present and Future, New Generation Computing, Vol. 1. No. 2 (1983).
2) Kowalski, R. A.: Predicate Logic as Programming Language, Proc. IFIP-74 Congr., North-Holland, Amsterdam (1974).
3) Warren, D. H. D.: Implementing Prolog, Technical Report D. A. I., Research Report No. 39, Univ. of Edinburgh (1977).
4) Hewitt, C.: Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, MIT, AI Lab. Rep., AI-TR-258 (1972).
5) Shapiro, E. Y.: Algorithmic Program Debugging, An ACM Distinguished Dessertation 1982, The MIT Press.
6) Tamaki, H. and Sato, T.: Unfold/fold Transformation of Logic Programs, Proc. of the Second International Logic Programming Conf., Uppsala Univ., Uppsala, Sweden (1984).
7) Sato, T. and Tamaki, H.: Transformational Logic Program Synthesis, Proc. International Conf. on Fifth Generation Computer Systems (1984).
8) Kitakami, H. et al.: A Methodology for Implementation of a Knowledge Acquisition System, Proc. 1984 International Symposium on Logic Programming, Atrantic City (1984).
9) Miyachi, T. et al.: A Knowledge Assimilation Method for Logic Databases, to appear in New Generation Computing.
10) 國藤他：Prolog による対象知識とメタ知識の融合とその応用，情報処理学会知識工学と人工知能研究会資料 30-1 (1983).
11) 北上他：大規模な知識ベース管理システムのアーキテクチャ，ICOT Technical Memo, TM-0070 (1984).
12) Clark, K. and Gregory, S.: Relational Language for Parallel Programming, Proc. Conf. on Functional Programming Languages and Computer Architecture, ACM (1981).
13) Clark, K. and Gregory, S.: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Imperial College (1984).
14) Shapiro, E.: A Subset of Concurrent Prolog and Its Interpreter, ICOT TR-003 (1982).
15) Colmerauer, A.: Prolog and Infinite Trees, in Logic Programming, Clark, K. L. and Tarnlund, S. A. (eds.), Academic Press (1982).
16) Chikayama, T.: Unique Features of ESP, Proc. International Conf. on Fifth Generation Computer Systems (1984).
17) Taki, K. et al.: Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI), ibid.
18) Yokota, M. et al.: A Microprogrammed Interpreter of the Personal Sequential Inference Machine, ibid.

(昭和 59 年 9 月 27 日受付)