解　説

Prolog 処理系†

中村　克彦††

1. まえがき

マルセーユ大学の Colmerauer ら[8] によって最初の Prolog インタプリタが作成されて以来，数多くの Prolog 処理系がつくられ，各種機能の組込み，および処理速度と記憶効率の向上などの面で多くの改良がなされてきた．これらの処理系は導出原理 (J. A. Robinson[19] 1965) にもとづく自動定理証明法の研究にくわえて人工知能研究において開発された各種プログラミング技術にその基礎をもっている．

Prolog の処理系は他の多くの言語の場合にはみられないつぎのような処理技術や機能を含んでいる．

（1） データベースの操作
（2） パターンマッチングによる規則の適用
（3） 非決定性プログラミング

これらはいずれも Planner[11], Conniver[13] などの人工知能用言語や多くのプロダクション・システム[20] にも使われているが，Prolog ではこれらの処理技術がきわめて簡単な文法の言語のなかに組込まれていることに特長がある．特に Prolog のパターンマッチングは統合 (unification) と呼ばれる基本演算であり，これによって変数への代入，新しいデータの生成，分解などが行われる．

この論文では Prolog の処理系の基本的構成と各種の処理方式について述べる．論理プログラミングは新世代の並列計算機のアーキテクチャの基礎となることが期待されているため，Prolog の並列処理のためのハードウェアもさかんに研究されているが，これは本特集に別の解説があるので，ここでは現在の計算機のための直列的な処理方法のみを扱う．プログラム例はエディンバラ版 Prolog[7] の構文に従う．

2. 節と項の表現

Prolog の文法において節 (Horn clause) を構成する基本的要素は項 (term) である．特にエディンバラ版 Prolog[7] では，節の頭部と副目標 (subgoal) ばかりでなく，節全体や節の条件部なども項の形式をもつ．原始プログラムまたはデータベース中の節に含まれる項 (原始項, source term) は統合によってそれに含まれる変数に代入がなされると新しい項 (instance) に変化する．したがって，処理系の設計においてはデータベース中の節と instance のデータ構造がまず大きな問題点となる．

2.1 データベースのデータ構造

データベースにおいて節を表すためにつぎのふたつの基本データ構造が使われている．

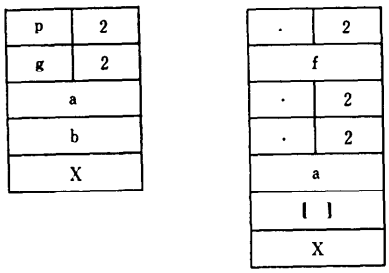
（1） レコード形式
（2） 2進リスト形式

図-1 にこのふたつの形式のデータ構造の例を示す．レコード方式はひとつの項を連続した記憶領域にポーランド記法の形式で格納する方法であり，エディンバラ版 Prolog を実現した多くのシステムが採用している．この形式では，リストは特別な項によって表す (図-1(a))．

リスト方式は Lisp の場合とほぼ同じデータ構造を用いる．述語項や関数項などの項はリストの特別な場合になる．

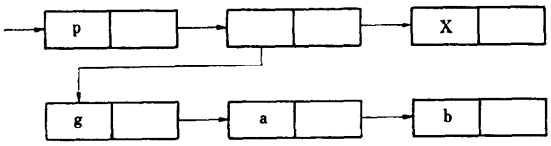
ふたつの方式の違いは当然，言語（文法）にも反映する．たとえば，リスト方式ではふたつの項，$f(a|X)$ (これは $[f, a|X]$ と同じ) と $f(a, b, c)$ を統合して変数 X にリスト $[b, c]$ を代入させることができるが，レコード方式ではこれは直接には不可能である．また，レコード方式のシステムではリストと項を相互に変換する組込み述語が用意されているが，リスト方式ではこれは不要である．このような2，3の点をのぞけば，入力プログラムを内部構造に変換する適当な入力ルーチンを用いることによって構文上の違いはそれほど大きくないようにすることができる．ふたつの方式の言語上の相違について Campbell[5] による詳細な比較がある．

† Prolog Systems by Katsuhiko NAKAMURA (Dept. of Systems Engineering, Tokyo Denki University).
†† 東京電機大学理工学部経営工学科



項 $p(g(a, b), X)$　　リスト $[f, [a]|X]$

(a) レコード形式



(b) リスト形式の項 $p(g(a, b), X)$

図-1 データベースのデータ構造

記憶の効率に関しては，ポインタがないだけレコード方式が優れている．一方，使用済の領域の再使用，項の instance を複写する際の効率，および処理系の記述の簡潔さの面ではリスト方式が有利である．

Nakamura[16] はデータベース中の節の変数を含まない部分リストを Goto[10] によって導入された mono-copy list (同一のリスト構造は同じポインタをもつ) の形式でハッシュ記憶に格納して，記憶領域の節約と統合の高速化を計っている．

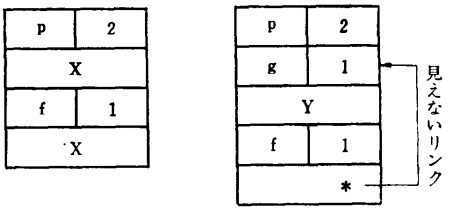
2.2 instance の表現

Prolog プログラムの節がある目標に適用されると，節の頭部と目標の統合によって変数に代入が行われ，つぎにこの節の副目標と始めの目標の後につづく目標の instance が評価される．したがって，instance の表現法が処理の効率に大きく影響する．この表現法にはつぎのようなふたつの方式がある．

2.2.1 構造複写 (copy) または非構造共有 (non-structure-sharing) 法

この方式では，副目標の instance は元の原始節のもの (変数はその値) をコピーすることによって作業記憶内に構成される．コピーの量を最小にして後にこの instance 中の変数への代入を行えるようにするため，構成される instance 中で複数回現れた同一変数は“見えないリンク”によってひとつの記憶場所へ接続される．

例　図-2 に項 $p(X, f(X))$ の変数 X に $g(Y)$ が代入された場合に複写によってつくられる instance の例をしめす．



原始項 $p(X, f(X))$　　複写された instance $p(g(Y), f(g(Y)))$

図-2 複写の例 (X=g(Y))

2.2.2 構造共有 (structure sharing) 法[1]

これは実際には instance を構成せずに，原始項と各変数に対してその値を与える環境とによって間接的に instance を表す方式である．すなわち，ある instance を参照するには，まずその原始項から出発して変数が現れると，環境からその値を求める．値となる項の instance も原始項と環境 (へのポインタ) によって表されているので，この過程は再帰的にくりかえされる．

図-3 に基本データ構造としてリストを用いた場合の統合プログラムを示す．関数 unify は構造共有方式で表されたふたつの項の instance の統合を試み，成功したとき値 TRUE を，失敗のとき FALSE を返す．このプログラムはCに類似した言語で書かれており，リストは Lisp と同様に car, cdr などの関数によって操作できることが仮定されている．

3. プログラム実行の制御

一般に，分岐の時点ではどちらに分岐するかを決定できないような非決定性分岐を含むプログラミング法を非決定性プログラミングと呼ぶ (Floyd[9])．Prolog ではある目標に適用できる節を選択する位置が非決定性分岐となる．Prolog の宣言的な側面は実行が非決定的であることと関係が深い．

非決定性プログラムのひとつの処理法は深さ優先探索によるものである．この探索戦略では，処理中にそれ以上実行を続けられない失敗の状態が検出されたとき，最後の非決定性分岐の位置に後もどり (バックトラック) してつぎの分岐先の実行が試みられる．ほとんどの Prolog システムは深さ優先探索を用いており，失敗が検出されると前の目標を再評価するためバックトラックする．深さ優先探索を採用した最大の利

```
unify(g,eg,h,eh)
    list  g, h;
    environment eg, eh;
     {
        if variable(h) then
              if unbound(h,eh) then
                     {
                      store(h,eh,g,eg);
                      return(TRUE);
                      }
              else return(unify(g,eg, getterm(h,eh),getenv(h,eh)));
        if variable(g) then
              if unbound(g,eg) then
                     {
                      store(g,eg,h,eh);
                      return(TRUE);
                      }
              else return(unify(getterm(g,eg),getenv(g,eg), h,eh));
        if atom(g) then
              {
                if g == h then return(TRUE);
                else return(FALSE);
              }
        if atom(h) return(FALSE);
        if unify(car(g),eg,car(h),eh) then
              return(unify(cdr(g),eg,cdr(h),eh);
        return(FALSE);
    }
```

unify(e,eg,h,eh): gとeg，hとehによって表されたふたつの項を統合し成功すればTRUE，失敗すればFALSE を返す。

variable(v): vが変数ならばTRUE.

unbound(v,e): 変数v が環境eにおいて未代入ならばTRUE.

getterm(v,e): 環境eの変数vの値（原始項）を返す。

getenv(v,e): 環境eの変数vの値（環境）を返す。

store(v,e,t,et): 環境eの変数vに値(t,et)を代入する。

図-3 構造共有法にもとづく統合のためのプログラム

点は各変数に対して環境は一意的な値を与えればよいのでシステムの構成が簡単になることである．これに対して，深さ優先探索以外の探索法では一般に多重環境を必要とする．

ある目標の評価が非決定的であるとは，バックトラック時にこの目標が再評価されたときふたたび成功する可能性をもつことである．この性質は目標とその評価のために適用される節，さらにこの節の副目標に対する節の適用にも依存するので，一般に静的には決まらない．ある節の適用中の副目標の評価がすべて決定的であるとき，この適用は決定的であるという．節の中のカット (cut) はこの節の適用を決定性に変える働きをもつ．

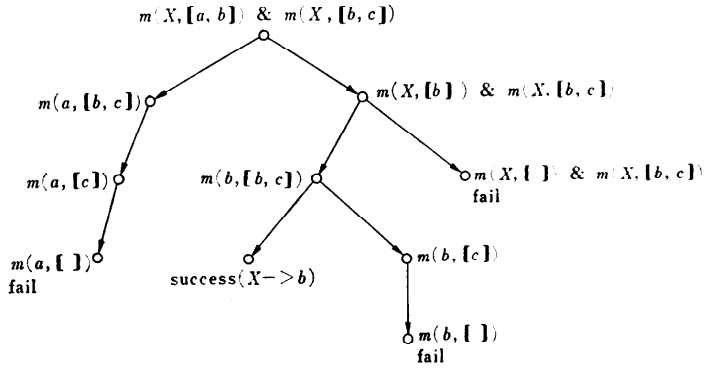
一般に，非決定性プログラムのある実行の様子は分岐を節点とする探索木によって表される．多くの Prolog システムは線形入力導出[15]を採用しているので，節点には目標の系列が対応する．図-4 にふたつのリストの共通の要素を求める問題に対する探索木の例を示す．

4. 作業領域の管理

Prolog システムにおいても Pascal のような手続き形言語の場合と同様に，ある変数にはひとつの記憶場所が対応し，節の適用に際してスタック上に連続した作業領域（フレーム）がとられる．しかし，つぎの諸点は Prolog 独特である．

（1） 節の非決定的な適用の場合は適用が終了しても変数の記憶場所を解放できない．

（2） 変数は未代入の状態から統合によって代入されてある値をもつが，バックトラックにより未代入の



プログラム $m(X,[X|\_])$.
$m(X,[Y|Z]):-m(X,Z)$.

図-4 探索木の例

状態に戻ることがある.

(3) 決定的な節の適用終了時に記憶場所を解放できる変数(局所変数)とできない変数(大域変数)がある.

Prolog プログラムの処理は上の性質(1)のために一般に多くの記憶領域を必要とする. つぎの各節に述べる処理方式は, 非決定性を含まない部分の処理に必要な記憶量が Lisp などの他の言語による同様な処理の場合と同程度にすることを目的としている. これらはいずれも Warren[21] によって開発され, Dec-10 Prolog システムに採用された.

4.1 多重スタック

ほとんどの Prolog システムではスタックに環境と制御情報を格納している. Dec-10 Prolog ではつぎのような3個のスタックを用いている. どのスタックもある節の適用開始時にその上に作業領域が確保される.

(1) 局所 (local) スタック
(2) 大域 (global) スタック
(3) trail スタック

どのスタックの領域も節の適用が失敗したときに解放されるが, 特に局所スタック上の領域は決定的な節の適用終了時にも解放される.

制御情報は主として局所スタックに, また変数の環境はつぎの **4.2** 節でのべるように局所および大域スタックに分けて格納される. trail スタックはバックトラックの際に未代入に戻す必要のある変数の記憶場所の情報を格納する.

同時に3個以上のスタックを用いる場合には, あふれたスタックを再配置する必要がある. 仮想記憶を用いる場合にはこの問題は重要ではない.

4.2 局所変数と大域変数

まず, 局所変数と大域変数の動的な区別を例によって説明する. つぎのようにある目標に節が適用されると仮定する.

目標 : $p(f(p), Q, h(f(a)))$
節 : $p(X, g(Y), h(Z)) :- q(X, Y, Z, e(U))$.

目標と節の頭部の統合によって変数 X, Q, Z につぎのような代入が起こる.

$X \rightarrow f(p)$
$Q \rightarrow g(Y)$
$Z \rightarrow f(a)$

この節の適用が決定的であれば(すなわち, 副目標 $q(f(p), Y, f(a), e(U))$ の評価が決定的ならば), この適用の終了後は変数 X と Z は参照されることはないので, X と Z の記憶場所を解放することができる. 一方, 変数 Y は目標の変数 Q の値に含まれているので, 節の適用終了後も残しておく必要がある. このとき, X および Z を局所変数, Y を大域変数と呼ぶ. さらに, 大域変数に後で代入される項に含まれる未代入変数も大域変数となる. すなわち, 上の例において副目標中の関数項 $e(U)$ がある大域変数に代入されると, これに含まれる変数 U も大域変数になる.

Dec-10 Prolog システムでは, 局所および大域変数の識別を簡単にするため, つぎのような静的区別法を採用している.

局所変数 : 常に節中の top level に現れている変数
大域変数 : 節の頭部および副目標の引数の関数項中に現れている変数

この区別法はきわめて簡単なので, プログラムの入力時にこのふたつを識別して局所および大域スタック上のフレーム内の記憶場所を割り当てることができる. これは特にインタプリタの場合きわめて有効な方法である.

静的な区別法によれば, 上の例の変数 Y だけではなく Z と U も大域変数となる. このように実際には局所変数として扱える記憶場所を大域変数のものとして扱うので, 大域スタックの garbage collection が必要になる[3),21)].

図-5 は構造共有法を採用したシステムにおける処理中の作業記憶の様子を表している. 局所および大域スタック上で各変数の値は環境(フレーム)およびヒープ内の原始項を指すふたつのポインタによって表されている. プログラムおよび質問はつぎの通りである.

```
append([ ], X, X).
append([A|B], C, [A|D]) :-append(B, C, D).
?-append([a, b, c], [ ], R).
```

Mellish[14] は構造複写法にもとづいた局所および大域変数の操作法を示し, 構造共有法と比較している. この方法では, 変数の記憶場所の配列が局所スタックに置かれ, instance が大域スタック上にコピーされる. 大域変数の実際の記憶場所はコピーされた項に含まれるので, 構造共用法の場合と同様に大域スタック上に置かれる. **図-6** はこの方法のシステムによって図-5 と同じプログラムを処理する場合の作業記憶の状態を表している.

4.3 再帰呼び出しの最適化[22]

Prolog では通常のプログラムのループもすべて再

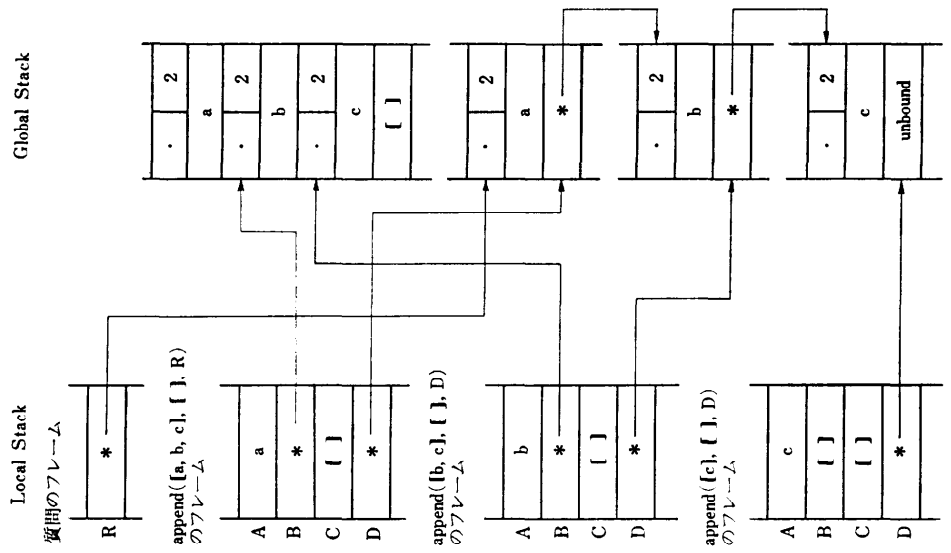


図-6 構造複写法の作業記憶 (Mellish[1] にもとづく)

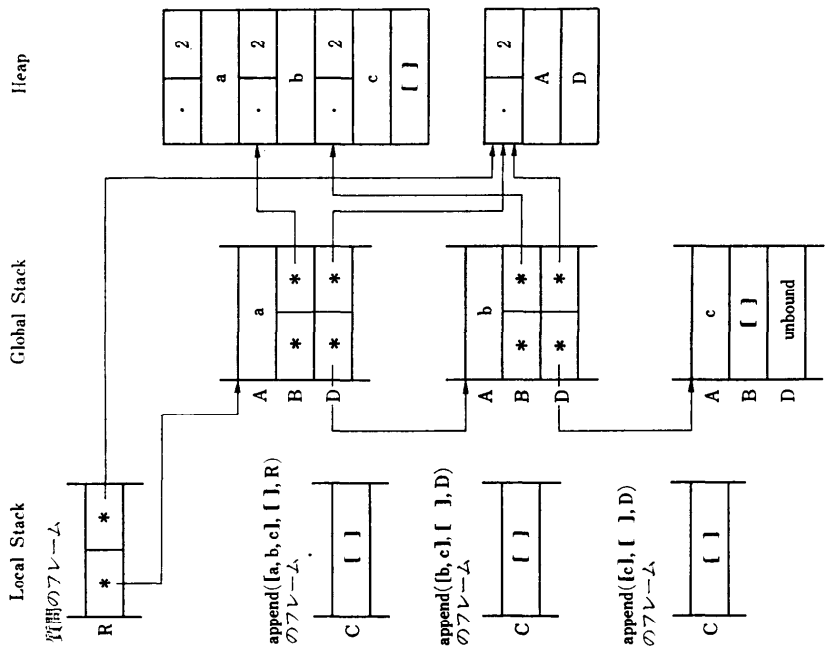


図-5 構造共有法の作業記憶 (Mellish[1] にもとづく)

帰によって表す．このため，ループで表せる再帰的表現はスタック領域を浪費することなく処理できる方式が重要になる．これは一般に末端再帰の最適化(end recursion optimization)と呼ばれているが，実際には決定的な節の適用の最後に他の任意の節が呼びだされる場合にも使われるため，最終呼び出しの最適化(last call optimization)ともなっている．

最適化が行われるのは，決定性の節の適用において節の最後の副目標とこれに適用される別の節の頭部との統合が成功したときである．すなわちこの後，第2の節の副目標の評価に移るときには，始めの節の適用のため局所スタック上のフレームは不用になるので，これが解放される．

Lisp に対しても同様の問題が議論されているが，Prolog の方がこの最適化を適要できる場合が多い．

5. Prolog システム

Prolog システムはインタプリタ，またはインタプリタとコンパイラを中心に，入出力ルーチン，組込み述語の集合，専用エディタなどによって構成される．

5.1 インタプリタとコンパイラ

現在の多くのシステムはインタプリタのみによってプログラムを処理するが，いくつかのシステムは高速処理のためにコンパイラももっている．コンパイラを用いることによりインタプリタの場合の数倍から数10倍の処理速度を得られることが報告されている．Warren[21]は Prolog コンパイラの処理速度が同じ問題に対する Lisp プログラムの Lisp コンパイラによる処理速度と同程度であることを示した．

コンパイラは入力プログラムの各節に対して，目標が与えられて初めて決定できる部分以外の処理を最適化または部分評価して目標の評価のための機械語コードをつくりだす．節の頭部はこれと統合が試みられる目標の各部分項を調べて変数への代入を行うための一連の命令コードに翻訳される．また，節の各副目標に対して，その引数を用意してその副目標の述語のためのサブルーチンを呼び出すような機械語コードがつくられる．

目的プログラムの処理効率は，節を適用する目標の引数の種類（定数項＝入力用，または未代入変数＝出力用）を制限することによって大幅に改善できる．Dec-10 Prolog では，モード宣言と呼ばれる特別なユニット節を用いてこの情報をコンパイラに与える．コンパイラ開発を容易にし，移植可能性を高めるため，多くの Prolog コンパイラは Prolog 自身で記述され，特別の中間言語または汎用高水準言語を目的言語としている[18),21)]．

5.2 組込み述語

Prolog では，通常の Horn 節によっては定義することが不可能な，入出力，ファイルの操作，算術演算，データベースへの節の追加と取り消しなどは組込み述語 (built-in predicate) を呼び出すことによって行

表-1 いくつかの Prolog システムの処理方式の分類

システム（作成者）	対象計算機言語	データ構造	instance の表現	local global の区別	compiler
Prolog (Colmerauer et al.)	Fortran	レコード	SS	●	●
Dec-10 Prolog (Warren et al.)	DECsystem 10 アセンブリ	レコード	SS	○	○
RT-11 Prolog (Mellish)	PDP-11 アセンブリ	レコード	NSS	○	●
Prolog/KR (中島)	LISP	リスト	SS	●	●
H-Prolog (中村)	C	リスト	SS	○	●
K-Prolog (紀，大西)	C	レコード	NSS	○	●
C-Prolog (Pereira et al.)	VAX-11 C	レコード	SS	○	●
LM-Prolog (Kahn et al.)	LISP machine LISP	リスト	NSS	●	○

SS：構造共有　NSS：構造複写　○：あり　●：なし

う．組込み述語のなかには再評価されたとき，つぎの操作を行うような非決定的なものがある．あるシステムの使いやすさや異なるシステムとの互換性は使用できる組込み述語の種類に依存する．

5.3 システム記述言語

これまでに公表された Prolog システムの記述言語はつぎの3種類に分類される．

（1） アセンブリ言語

（2） Lisp

（3） Pascal, C などの汎用高水準言語

作成されるシステムの処理方式の選択，処理効率，組込み述語のもつ機能などは記述言語およびオペレーティング・システムにかなり依存する．

6. むすび

この論文では現在の計算機に Prolog をインプリメントするための主要な技術について述べた．**表-1** にいくつかの Prolog システムの採用している処理方式をまとめておく．

Prolog の歴史は新しいので，つぎのような分野に将来の発展が期待できる興味深い研究テーマが数多く残されている．

（1） 処理アルゴリズム（特にコンパイラ）

（2） データベースの構成，および他のデータベース・システムとの接続

（3） 探索方式

Prolog 処理方式についての多くの研究テーマは Prolog 言語および論理プログラミングの問題点に深く関係しており，また新しいアーキテクチャの開発とも密接な関連がある．

謝辞 原稿の作成に協力いただいた本学塩谷勇助手に感謝する．

参考文献

1) Boyer, R. S. and Moore, J. S.: The Sharing of Structure in Theorem Proving, in Machine Intelligence 7 (eds. Melzer, B. and Michie, D.), Edinburgh University Press (1972).
2) Bruynooghe, M.: The Memory Management of Prolog Implementation, in Logic Programming (eds. Clark, K. L. and Tarnlund, S. A.), Academic Press (1982).
3) Bruynooghe, M.: Garbage Collection in Prolog Interpreters, in [Campbell 1984].
4) Campbell, J. A.: Implementations of PROLOG, p. 384 Ellis Horwood, (1984).
5) Campbell, J. A. and Hardy, S.: Should Prolog Be List or Record Oriented?, in [Campbell 1984].
6) Clark, K. L. and McCabe, F. G.: Micro-PROLOG: Programming in Logic, p. 401 Prentice-Hall, (1984).
7) Clocksin, W. F. and Mellish, C. S.: Programming in Prolog, p. 274 Springer-Verlag, (1981), 中村訳 Prolog プログラミング，マイクロソフトウェア (1983).
8) Colmerauer, A.: Metamorphosis Grammars, in Lecture Notes in Computer Science 63, Springer-Verlag, pp. 133-190 (1978).
9) Floyd, R. W.: Nondeterministic Algorithms, J. ACM, Vol. 14, No. 4, pp. 636-644 (1967).
10) Goto, E.: Monocopy and Associative Algorithms in Extended LISP, Technical Report of Information Science Laboratory, University of Tokyo (1974).
11) Hewitt, C.: Planner: A Language for Proving Theorem in Robot, Proc. of IJCAI (1969).
12) Kahn, K. M. and Carlsson, M.: How to Implement Prolog on a LISP Machine, in [Campbell 1984].
13) McDermott, D. and Sussmann, G.: The Conniver Reference Mannual, MIT AI LAB MEMO 259 A (1974).
14) Mellish, C. S.: An Alternative to Structure Sharing in the Implementation of Prolog Programs, Dept. of Artificial Intelligence Research Report No. 150, University of Edinburgh (1981).
15) Kowalski, R.: Logic for Problem Solving, Elsevier North-Holland (1979).
16) Nakamura, K.: Associative Evaluation of Prolog Programs, in [Campbell 1984].
17) Nakashima, H.: Prolog/KR—Language Features, Proc. of 1st International Logic Programming Conference (1982).
18) Nilson, J. F.: On the Compilation of a Domain-based Prolog, Proc. of IFIP, pp. 293-298 (1983).
19) Robinson, J. A.: A Machine Oriented Logic Based on Resolution Principle, J. ACM, Vol. 12, No. 1, pp. 23-44 (1965).
20) 田中編：知識工学，p. 411，朝倉書店 (1984).
21) Warren, D. H. D.: Implementing PROLOG—Compling Predicate Logic Programs Vol. 1 and 2, Dept. of Artificial Intelligence Research Report No. 39 and 40, University of Edinburgh (1977).
22) Warren, D. H. D.: An Improved Prolog Implementation which Optimises Tail Recursion, Proceedings of Logic Programming Workshop, pp. 1-11 (1980).

（昭和59年7月31日受付）