記 号 処 理 25－1
（１９８３．１０．３）

手続きと論理を融合した知識処理言語 ＰＲＥＳＴＯとその支援系

石田　亨　　串間　和彦　　東田　正信　　和佐野哲男
（日本電信電話公社横須賀電気通信研究所）

１．まえがき

論理型言語Prologは、ユニフィケーション（unification)とバックトラックという２大機能を有し、従来のプログラミング言語に比べ、自然言語処理やエキスパートシステム等の知識処理を容易に記述できる言語として期待されている。しかし一方では、Prologの欠点が指摘され[1]、改良、拡張の検討が行われている。本資料では、Prologの実用性を向上させる第１歩として、以下の問題点を取り上げ解決策を提案する。

（１）手続きと論理の記述に関する問題点

知識処理には探索（即ちバックトラック）を要する処理と、探索を要しない処理（手続き的な処理と呼ぶ）が混在して含まれている。知識処理システムの開発には、双方の処理を適切に記述できる言語機能と、デバッグできる支援系が必要である。しかし現状のPrologには、以下の問題がある。

①手続き的制御記述能力の欠如

探索を要する処理の記述を目的とした言語機能は存在するが、手続き的な処理の記述を目的とした言語機能は存在しない。

②論理的デバッグ機能の欠如

一方、デバッグ機能としては、Prolog処理系（論理を手続き的に解釈実行する）の実行経路をトレースする機能が用意されているだけである。プログラマはプログラムを論理のレベルで記述したにもかかわらず、処理系の実行過程を手続き的に追うことを強いられるため，デバッグは容易でない。

（２）実用上の種々の問題点

①モジュール化機能の欠如

多数のプログラマが共同でシステムの開発を行う場合には、モジュール化機能が必須である。現状のPrologで共同開発を行うためには、システム中で用いられる名標を全て管理する必要がある。

②日本語処理機能の欠如

多様化している文字データを扱うためには１、２Byte系の混在したデータに対応できる機能が必要である。

③言語間リンク機能の欠如

Prologと適用領域を異にする他言語プログラムとの双方向リンク機能が必要である。また、既存の膨大なソフト資産との有機的結合を実現するためにも、言語間リンク機能は必要である。

④保守の困難さ

Prologは論理のレベルでプログラムを記述するものであるため、読解性は良い。しかしプログラム開発者以外の人間には、述語の意味が容易には理解できないため、この長所が十分生きない場合がある。

⑤流通性の欠如

現段階ではPrologの標準化は予定されていないが、実用言語としては、処理系、及びプログラムの流通性が要求される。

上記問題点に対して、既にいくつかの提案がなされている。手続き的制御記述に関しては、Prolog K/R[2]がいくつかの高階の制御記述を導入している。モジュール化機能はMPROLOG[3]により導入が行われている。日本語処理機能は漢字Prolog[3]により導入が行われている。筆者らは実用あるいは実用を前提としたプロトタイプ開発に適する環境を実現する第１歩として、Prologとその支援系に上記の問題点に関する改良を加

えた知識処理言語PRESTO(PRogramming in Efficient and STructured lOgic) の開発に着手した。PRESTOの当面の目標は以下のとおりである。

（１）手続き的な処理の記述、論理のレベルでのデバッグを可能とし、手続きと論理の融合を実現する。

①Prolog K/Rの考え方をさらに進め、手続き的な処理の記述に必要な制御述語を網羅的に導入する。また、プログラム上の表現形式として、従来手続き型言語で用いられてきた表現形式を許す。

②論理のレベルでのデバッグを可能とする論理トレース機能を導入する。

（２）実用上の種々の問題点を解決することにより、実用あるいは実用を前提としたプロトタイプ開発に適した知識処理プログラミング環境を実現する。

①モジュール化機能を導入する。

②日本語処理機能を導入する。

③FORTRAN 、COBOL 、Ada 等の他言語との双方向リンクを実現する。

④述語の意味をあらかじめ登録することによって、論理の意味を表示する日本語ドキュメンタを導入する。

⑤処理系の流通性は、流通性の高い言語で記述することによって保証する。また、PRESTOプログラムの流通性は、コンパイラがオブジェクト生成過程で流通性の高い言語のソースプログラムを出力することにより、保証する。

２．手続き的制御の記述

２．１　基本的な考え方

従来のPrologでの手続き的制御の実現手段と問題点を以下に示す。

①　逐次的な制御は、本来探索を要しない処理であるにもかかわらず、探索を前提とした言語機能である論理式の形で表現される。この結果、プログラム上で探索を要する処理と、要しない処理の判別がつきにくい。

②　条件分岐による制御は、カットオペレータを用いてPrologにより行われる全域的なバックトラック経路を物理的に変更することにより実現される。プリミティブな制御機能であるカットオペレータのみを組合わせて複雑な制御構造を実現するため、プログラム開発が困難なばかりでなく、プログラム中のカットオペレータ相互の関連が把握しにくい。

③　ループ制御はバックトラック（repeat述語）を用いる方法と再帰呼出しを用いる方法がある[5] 。バックトラックを用いる方法は動作が難解であり、余分なバックトラックが行われて思わぬ結果が導びかれることがある。一方、全てのループを再帰呼出しを用いて実現することは効率上現実的でない。

PRESTOでは、探索を要する処理と手続き的な処理のそれぞれを適切に表現できるよう、Prologに以下の拡張を加える。

①　手続き的制御述語として、逐次構造、IF構造、CASE構造、LOOP構造を表す述語を用意する。各述語を組合わせることにより、任意の制御構造を実現することができる。

②　手続き的制御述語のプログラム上の外部表現形式として、従来の述語形式の他に、手続き型言語で用いられてきた表現形式（手続き的制御記述と呼ぶ）を許す。

③　LOOP構造では、同一の変数に何度も繰り返し異なる値をbindする必要が生じる。これを可能とするため、変数の値をunbindする機能を導入する。

２．２　手続き的制御述語とその外部表現

手続き的制御述語の基本的な機能は以下のとおりである。

①　手続き的制御述語はパラメータとして複数のゴール式（ゴールをAND 、ORで結んだもの）をとる。手続き的制御述語はこれらのゴール式の導出順を手続き的に制御する。

②　探索は１個のゴール式の範囲で実行される。複数のゴール式に渡るバックトラックは行われない。

従って、プログラム中の全てのゴール式が１個のゴールで構成され、それぞれのゴールの処理にオルタナティブが存在しない場合には、そのプログラムは完全に手続き的な処理を表現している。逆にプログラム中で手続き的制御述語が用いられていなければ、そのプログラムは手続き的な要素の全く無い探索処理を表現している。

手続き的制御述語は複雑な述語であるため、従来の述語形式で記述すると読解性が悪い。そこで手続き型言語と同様の外部表現を許すこととする。PRESTOの手続き的制御記述の構文定義を図１に示す。また、既存の代表的なPrologであるDEC10-Prolog[6] とPRESTOにより手続き的な処理を記述した場合の比較を図２に示す。逐次構造、IF構造、CASE構造、LOOP構造の実行は以下のように制御される。

① 逐次構造は、その構成要素を逐次実行する。ある構成要素がfailした時点で逐次構造がfailする。

② IF、CASE構造は、条件部の実行結果（success 、fail）に従って逐次構造を選択実行する。

③ LOOP構造は、WHILE 条件部がsuccess する間、ループ内の逐次構造を実行する。EXITが実行されればループを脱出する。

２．３ 具体例

実際にDEC10-Prologで記述された具体例を用いて、手続き的制御記述の導入効果を説明する。図３は同一プログラムをDEC10-PrologとPRESTOのCASE構造により記述したものである。このプログラムには探索を要する処理は含まれていない。それにもかかわらずDEC10-Prologでは、常に探索が前提とされるため、カットオペレータを用いてバックトラック経路を変更する必要が生じている。これに比べPRESTOでは、探索はゴール式に局所化され、それらのゴール式の導出順はCASE構造によって制御されるため、カットオペレータは不要となり読解性の良い記述となっている。

図４はIF構造の使用例を示している。DEC10-Prologによる記述では、探索を要する箇所が不明確であったが、PRESTOではIF構造の条件部でのみ使用していることが明示されている。また、DEC10-Prologでは条件成立時の処理と、不成立時の処理とを別個の節を用いて表現せざるを得なかったが、PRESTOでは１個の節内に記述可能となっている。

図５は再帰呼出しを用いて構成されたループをLOOP構造で記述しなおしたものである。ループの脱出条件が別個の節となっていたものが、１個の節内で記述可能となっている。ループを実現するには同一変数に何度も異なる値を設定する機能を導入しなければならない。そこで、PRESTOでは、unification が行われる直前の値をunbindする機能を導入することによって、この問題を解決している。unbindされるべき変数は先頭に＊を付加して表現する。例えば、

```
*Read=Seen
```

は、ループを回る毎にReadに値が再bindされることを示している。なお、unbindされた値はバックトラックにより復元される。

３．論理トレース

従来のPrologでは、手続き的な処理の記述能力が欠けている反面、デバッグ時には論理レベルのデバッグが行えず、インタプリタの処理手順をトレースしなければならないという欠点があった。PRESTOでは、論理レベルのデバッグが可能な論理トレース機能を導入することで、従来のPrologの欠点を克服する。論理トレース機能は以下の特徴を有している。

①プログラマは実行結果を節単位にトップダウンに、デバッグに必要な最小限の情報のみを表示させることができる。

②プログラマは誤りを含む節を以下のデバッグアルゴリズムに従って検出できる。

デバッグの基本的な考え方とアルゴリズムは次のとおりである。

> 節本体のゴールが全て正しく計算され、かつ節頭部が誤っている場合には、節の定義に誤りがある（但し、逆は成立しない）。

たとえば、ａ＆ｂ＆ｃ－＞ｄという論理でａ、ｂ、ｃが真であるにもかかわらずｄが偽であれば、その論理自体が誤っているといえる。

step1 ： システムはまず最初に起動された節の計算結果を表示する。プログラマはその節頭部(clause-head) の正しさを確認する。

正しい－－＞プログラムの実行は正しい（テスト完了）。

誤り－－＞step2 へ。

step2 ： 誤りを含む節を見つけるまで以下の処理を繰り返す。

2.1 プログラマは節本体に含まれるゴールを全て調べ、計算結果に誤りのあるゴールを見つける。

誤ったゴールが無い－－＞節本体に含まれるゴールが全て正しいのに、節頭部に誤りがあるから、この節の論理に誤りがある（バグ発見）。

誤ったゴールがある－－＞ 2.2へ。

2.2 システムに誤ったゴールを節頭部とする節の計算結果を表示させる。

図６に論理トレースを用いたデバッグ例を示す。論理トレース機能は、①プログラムが途中でfailして実行が中断された場合、及び②プログラムの実行は終了したが値が誤っている場合の双方に有効である。failしたゴールには先頭に＊を付加することによって、バックトラックの模様がひとめで分るようになっている。例では、shubu を定義した２番目の節に誤りがあるためバグが生じている。節shubu に誤りがあることは、節shubu を構成するゴールの計算結果が全て正しい（failした場合を含めて）にもかかわらず、shubu 述語の計算結果が誤っていることにより容易に検出できる。プログラムの規模が増大した場合には、従来のトレースリストを追うのは容易ではない。論理トレースを用いると、探索を要する処理のデバッグを容易に行うことができる。

PRESTOでは論理トレースを基本とし、以下に示すさまざまな工夫を加えることにより、手続き的な処理と探索を要する処理の混在したプログラムに対する総合的な会話型デバッグ環境を実現する。

①内部にループを含む節のトレースでは、ループの過程を順に追う逐次型のトレース機能を併用できる。

②節の計算結果の表示にあたっては、入力パラメータと出力パラメータを区別して表示することができる。

③ソースプログラムとトレース結果を対応を取って表示することができる。

４．その他の特徴的機能

PRESTOは実用的な知識処理プログラミング環境を実現するために、以下の特徴的機能を備える。

４．１　モジュール化機能

多くのプログラマが分担してプログラムを開発する環境では、モジュール化機能が必要である。PRESTOは以下のモジュール化機能を有する。

①関連する節をまとめて１モジュールとして定義する機能。

②他モジュールに使用を許可する節を宣言(export)する機能。

③モジュール内で使用する他モジュールの節を宣言(import)する機能。

４．２　日本語処理機能

自然言語理解、文書処理等を行うためには、漢字、ひらがなを使用できる必要がある。PRESTOは以下の日本語処理機能を有している。

①ＪＩＳ２Byteコード、１Byteコードが混在した文字データを扱える。

②プログラム内で漢字、ひらがなを用いた名標が扱える。

4．3　言語間リンク機能

大規模システムの一部に知識処理を導入する場合には、既存ソフトウェア資産とPRESTOプログラムとの有機的な結合を図る必要がある。PRESTOでは、他言語で記述されたソフトウェアとの双方向リンクを以下のように実現する。

（1）他言語プログラムからPRESTOプログラムの呼出し

①他言語プログラムはPRESTOインタプリタそれ自身をサブルーチンとして呼出すことができる。

②PRESTOコンパイラはリンケージ可能なオブジェクトを生成する。他言語プログラムはインタプリタ同様、オブジェクトをサブルーチンとして呼出すことができる。

（2）PRESTOから他言語プログラムの呼出し

PRESTOから他言語プログラムを述語として呼出すことができる。

4．4　日本語ドキュメンタ

論理型言語はプログラムを論理のレベルで表現するため、読解性が良いと考えられている。しかし、実際には各々の述語の意味が理解しにくいため、開発者以外の人間がプログラムを保守するのは容易ではない。

PRESTOでは論理の意味を日本語で表示する日本語ドキュメンタを備えることによって、保守性の向上を図る。日本語ドキュメンタの出力イメージを図7に示す。日本語ドキュメンタを使用するために、プログラマはまず各々の述語の意味を日本語で登録する。システムは登録された述語の意味を組合わせることによって、論理の意味を表示することができる。

4．5　プログラムの流通性

論理型言語は当面の間、言語仕様をリファインする時期が続くと考えられる。PRESTOではプログラムの流通性を保証するため以下の方針をとる。

①PRESTO処理系（インタプリタ、コンパイラ）は流通性の高いシステム記述言語（Ada 、C 等）で記述する。

②PRESTOコンパイラはオブジェクトを生成する過程で中間情報としてシステム記述言語のソースを出力する。

従って、PRESTO処理系、及びPRESTOプログラムの流通性はシステム記述言語の流通性によって保証される。

この方針はまた、処理系の作成を容易とする。即ち、PRESTOコンパイラは論理から手続きへの変換、最適化を重点的に行えばよく、オブジェクトレベルの最適化、マルチターゲット化等をシステム記述言語のコンパイラに行わすことができる。

5　むすび

知識処理言語PRESTOの言語機能と支援系について述べた。PRESTOは、①手続き的制御記述と論理トレース機能の導入による手続きと論理の融合、②実用上の種々の問題点の解決による実用的な知識処理プログラミング環境の実現を目標としている。今後の開発、適用を通じて得られる研究成果は別途報告する予定である。

参考文献

[1] Kurokawa,T. : “Logic Programming--What does it bring to the software engineering?“, First International Logic Programming Conference(1982).

[2] 中島　：”Prolog/KR の概要”，記号処理，18-5(1982).

[3] Bendl,J.,et.al. ：”The MPROLOG System”，Logic Programming Workshop(1980).

[4] 太細他　：”漢字PROLOG”，情報処理第25回全国大会(1982).

[5] 中島　：”PROLOGとその処理系”，情報処理，23，11(1982).

[6] Bowen,D,L.：”DECsystem-10 PROLOG USER'S MANUAL ”(1981).

[7] 宮地他　：“論理データベース向きの知識同化方式の一提案“，ICOT，(1983).

```
goal-expression ::=
     goal { # | & } goal-expression
sequence-structure ::=
     { goal-expression
       if-structure
       case-structure
       loop-structure
       exit-predicate } ; 。。。
if-structure ::=
     if goal-expression
          then sequence-structure
          [ else sequence-structure ]
     endif
case-structure ::=
     case term is
        { when term => sequence-structure } 。。。
        [ when others => sequence-structure ]
     endcase
loop-structure ::=
     [ while goal-expression ]
     loop sequence-structure
     endloop
exit-predicate::=
     exit [ when goal-expression ]
```

（注） {a};。。。は；で区切ってaを任意回繰り返すことを示す。

図１　手続き的制御記述の構文定義

(DEC10-Prolog)

```
a :- a1.
a :- a2.
a1 :- b,!,c,!,d.
a2 :- e,!,f,!,g.
```

（手続き的制御記述導入後）

逐次構造

```
a :- b; c; d.
a :- e; f; g.
```

IF構造

```
a :- b ,!,c.
a :- d.
```

```
a :-if b then c else d.
```

CASE構造

```
a(b1) :- !,c1.
a(b2) :- !,c2.
a(X) :- c3.
```

```
a(X) :-case X is
        when  b1 => c1
        when  b2 => c2
        when  others => c3
       endcase.
```

LOOP構造

```
repeat.
repeat :- repeat.
a :- repeat,(b;c,fail).
c :- d,!.
```

```
a :-loop
     if b then exit else c
   endloop.
c :- d.
```

（注）DEC10-Prologでは';' はORを表わすが、本記法では逐次制御を表わす。

図２　手続き的制御記述導入前後の比較

```
check-args([+ | Rest],[A | More]) :- !,
              nonvar(A),!,
              check-args(Rest,More).
check-args([- | Rest],[A | More]) :- !,
              var(A),
              check-args(Rest,More).
check-args([? | Rest],[A | More]) :- !,
              check-args(Rest,More).
```

＜DEC10-Prologによる記述＞

```
check-args([Argmode | Rest],[A | More]) :-
    case Argmode is
      when  + => nonvar(A);
                 check-args(Rest,More)
      when  - => var(A);
                 check-args(Rest,More)
      when  ? => check-args(Rest,More)
    endcase.
```

＜PRESTOによる記述＞

（注）上記プログラムはモードチェックプログラムの一部である。

図３　CASE構造の使用例

```
assim(Input,View):-
      assert(current-db(Input)),
      current-db(check-db(Input,IC,Message,ViewX)),
      co-owner(View,ViewX),
      ic-trans(IC,ICR),
      Check-IC=..[demo,ICR,View],
      Check-IC,ttyn1,ttyn1,
      display('--- Input conflicts with '),
      display('the Integrity constraint !!'),
      ttyn1,ttyn1,display('    '),
      display(Massage),
      ttyn1,ttyn1,ttyn1,
      retract(current-db(Input)).
assim(Input,View):-retract(current-db(Input)),fail.
```

（注）上記プログラムは知識同化
プログラム[7] の一部である。

＜DEC10-Prologによる記述＞

```
assim(Input,View):-
    assert(current-db(Input));
    if current-db(check-db(Input,IC,Message,ViewX))
       & co-owner(View,ViewX)
       & ic-trans(IC,ICR)
       & Check-IC=..[demo,ICR,View]
       & Check-IC
    then ttyn1;ttyn1;
         display('--- Input conflicts with ');
         display('the Integrity constraint !!');
         ttyn1;ttyn1;display('    ');
         display(Massage);
         ttyn1;ttyn1;ttyn1;
         retract(current-db(Input))
    else retract(current-db(Input));
         fail
    endif.
```

＜PRESTOによる記述＞

図４　IF構造の使用例

```
rest(end-of-file,Read) :- !.
rest(Other,Read):-
        expand-term(Other,Term),
        handle(Term,Read,Seen),
        read(Next),!,
        rest(Next,Seen).
```

（注）上記プログラムはモードチェックプログラム
の一部である。

＜DEC10-Prologによる記述＞

```
rest(Term1,Read) :-
    while Term1≠end-of-file
    loop
       expand-term(Term1,*Term2);
       handle(Term2,Read,*Seen);
       read(*Term1);
       *Read=Seen
    endloop.
```

＜PRESTOによる記述＞

図５　LOOP構造の使用例

```
<プログラム>  [1]  bun(X,Y,Z) :- shubu(X,X1,Z1) & jutsubu(X1,Y,Z2) & Z=[bun,Z1,Z2].
              [2]  shubu(X,Y,Z) :- meishi(X,X1,Z1) & joshi(X1,Y,Z2) & Z=[shubu,Z1,Z2].
              [3]  shubu(X,Y,Z) :- keiyoushi(X,X1,Z1) & meishi(X1,X2,Z2) & joshi(X2 * ,Y,Z3)
                                   & Z=[shubu,Z1,Z2 ** ,Z3].
              [4]  jutsubu(X,Y,Z) :- doushi(X,Y,Z1) & Z=[jutsubu,Z1].
              [5]  keiyoushi(X,Y,Z) :- X=[chiisai | Y] & Z=[keiyoushi,chiisai].
              [6]  meishi(X,Y,Z) :- X=[inu | Y] & Z=[meishi,inu].
              [7]  joshi(X,Y,Z) :- X=[wa | Y] & Z=[joshi,wa].
              [8]  doushi(X,Y,Z) :- X=[hashiru | Y] & Z=[doushi,hashiru].
```

<デバッグ手順>

＊印の変数を誤ってX1とした場合

```
[1]  *bun([chiisai,inu,wa,hashiru],[],Z) :-                    誤
[2]    *shubu([chiisai,inu,wa,hashiru],X1,Z1)                  正
[3]    *shubu([chiisai,inu,wa,hashiru],X1,Z1)...               誤
                 |
                 |  [3]shubuの追跡
                 ↓
[3]  *shubu([chiisai,inu,wa,hashiru,Y,Z) :-                    誤
[5]      keiyoushi([chiisai,inu,wa,hashiru],                   正
           [inu,wa,hashiru],[keiyoushi,chiisai]) &
[6]      meishi([inu,wa,hashiru],                              正
           [wa,hashiru],[meishi,inu] &
[7]      *joshi([inu,wa,hashiru],Y,Z3)...                      正
```

プログラムが途中でfailした場合のデバッグ例

＊＊印の変数を誤ってZ1とした場合

```
[1]  bun([chiisai,inu,wa,hashiru],[],[bun,[shubu,              誤
       [keiyoushi,chiisai],[keiyoushi,chiisai],
       [joshi,wa]],[jutsubu,[doushi,hashiru]]]) :-
[2]    *shubu([chiisai,inu,wa,hashiru],X1,Z1)                  正
[3]    shubu([chiisai,inu,wa,hashiru],[],[shubu,               誤
         [keiyoushi,chiisai],[keiyoushi,chiisai],
         [joshi,wa]]) &
       jutsubu([hashiru],[],[jutsubu,[doushi,                  正
         hashiru]])...
                  ↓  [3]shubuの追跡
[3]  shubu([chiisai,inu,wa,hashiru],[],[shubu,                 誤
       [keiyoushi,chiisai],[keiyoushi,chiisai],
       [joshi,wa]]) :-
[5]    keiyoushi([chiisai,inu,wa,hashiru],                     正
         [inu,wa,hashiru],[keiyoushi,chiisai]) &
[6]    meishi([inu,wa,hashiru],[wa,hashiru],                   正
         [meishi,inu]) &
[7]    joshi([wa,hashiru],[hashiru],[joshi,wa])...             正
```

プログラムの実行結果が誤っている場合のデバッグ例

図６　論理トレース機能を用いたデバッグ例

```
<プログラム>  [1]  reverse(L,L1) :- reverse-concatenate(L,[],L1).
              [2]  reverse-concatenate([X | L1],L2,L3) :- reverse-concatenate(L1,[X | L2],L3).
              [3]  reverse-concatenate([],L,L).
```

<述語の意味（利用者登録）>

reverse(A,B)　：　ＢはＡの逆リスト。

reverse-concatenate(A,B,C)　：　Ａの逆リストとＢを結合したものがＣ。

<論理の意味（システム出力）>

[1]　Ｌの逆リストと［］を結合したものがＬ１であれば、Ｌ１はＬの逆リスト。

[2]　Ｌ１の逆リストと［ＸｌＬ２］を結合したものがＬ３であれば、
［ＸｌＬ１］の逆リストとＬ２を結合したものがＬ３。

[3]　［］の逆リストとＬを結合したものはＬ。

（注）上記は宣言的な読み方による出力であるが、手続き的な読み方による出力も考えられる。

図７　日本語ドキュメンタの出力イメージ