Vol. 30 No. 4　　情　報　処　理　　Apr. 1989

解　説

1. プログラミング言語と環境

1.2 Prolog のプログラミング環境†

田村直之††　浅川康夫†††

1. はじめに

Prolog は Lisp に比べて言語の歴史が浅く，プログラミング環境に関する研究も遅れているといわれている．しかし，最近では ICOT の PSI マシン上の SIMPOS[1]をはじめとして，統合的プログラミング環境を備えたシステムも開発されつつあり，デバッガなどの個々のプログラミング・ツールについても，Prolog 言語の特徴に着目した研究が活発に進められている．これらの研究は，Prolog の応用分野が広がるにつれ，今後ますます重要になってくるものと考えられる．

本稿では，まず Prolog プログラミング環境の特徴を明らかにし，次に具体的な Prolog のプログラミング支援ツール群の紹介を行い，最後に今後の Prolog プログラミング環境への期待を述べることとする．

2. Prolog プログラミング環境の特徴

Prolog プログラミング環境の特徴を明らかにするためには，Prolog の使用目的および Prolog 言語の特徴について述べる必要があるだろう．

Prolog は，一般には人工知能システムの研究・開発を行う上でのプログラム記述言語として用いられている．人工知能システムの研究・開発では，新しいアイデアを実験的に計算機の上で実現してみること，すなわちプロトタイピングがしばしば行われ，研究・開発過程での重要な役割を果たす．つまり，研究者や開発者は，新しく思いついたアイデアのエッセンスを実現したプロトタイプのプログラムを作成し，実行し，その結果をみてプログラムを改善したり，あるいは別のアイデアに基づいた新しいプログラムを作成する．このような作業は，システムが完成するまで繰り返し何度も行われる．したがって，Prolog のプログラミング環境では，エディタ，Prolog 実行系，デバッガなどのプログラミング・ツールが相互に結合され，それらを会話的に使用でき，上記の作業を能率的に行える必要がある．

また，Prolog で開発する応用システムの規模は今後しだいに大きくなっていくと考えられるが，その場合，プログラムのモジュール化をサポートするツールや，未定義述語などのソース・プログラム中のエラーを発見して警告するツールなど，大規模システム開発のためのツールが必要になる．一つのアプローチとしては，ICOT の ESP[2] にみられるように，Prolog をオブジェクト・オリエンティッドな言語に拡張し，環境を整える方法がある．他のアプローチとしては，Prolog に Modula-2 や Ada にみられるようなモジュール化機能を付加し，それに対応したプログラミング環境を整える方法がある[3),4)]．しかし，本稿では言語の拡張については触れないこととする．

Prolog の言語上の特徴としては，ユニフィケーションとバックトラッキングがあげられる．このうち，バックトラッキングによる自動後戻り機能は，有効ではあるが，ときとして実行の制御の流れがみえにくくなり，デバッグが困難になることがある．これに対しては，制御の流れをユーザに分かりやすく表示するツールや，バグの発見を助けるツールなどが必要になる．

3. プログラミング支援ツール群

ここでは，Prolog プログラミング環境を構成するツールとして，エディタ，デバッガ，コンパイラなどを中心に，具体的なシステムの紹介を行う．

3.1 エディタ

これまでに述べたように，Prolog のプログラミング環境では，Prolog 実行系と各種ツールとの結合が重要である．特に，エディタとの結合はプロトタイピングを能率的に行う上でもっとも有効である．ここで

† Prolog Programming Environments by Naoyuki TAMURA (Faculty of Engineering, Kobe-University) and Yasuo ASAKAWA (IBM Tokyo Research Laboratory).

†† 神戸大学工学部
††† 日本アイ・ビー・エム東京基礎研究所

いうエディタと実行系の結合とは，次のようなことが可能な環境を指す．

（1） エディタと実行系を並行して（コルーチン的に）実行できる．エディタと実行系との間を行き来しても，編集中のテキストや実行系の述語定義などは保存されている．

（2） 相互にデータをやり取りできる．編集中のテキストの一部をゴールとして実行したり，実行結果をテキストに取り込んだりできる．

このような環境を実現する方法としては，

（1） Prolog 処理系中に専用のエディタを組み込む．

（2） Prolog 実行系と外部のエディタを結合する．

の二つが考えられる．以下では，（1）の例として Prolog-KABA を，（2）の例として PCALL を取りあげ紹介する．

3.1.1 Prolog-KABA[5]

Prolog 処理系内に組み込まれたエディタの例として，Prolog-KABA の ProEdit を取りあげる．Prolog-KABA は，京都大学のグループによって開発された Prolog の処理系で，NEC の PC-9800 などのパーソナルコンピュータで利用できる．ProEdit は Prolog-KABA に組み込まれた EMACS[6] ライクな画面エディタである．

ProEdit は，Prolog-KABA の実行系から組込み述語 edit により呼び出せる．ProEdit からはコマンド ^X^Z（コントロールXにつづけてコントロールZを打鍵する）で Prolog 実行系に戻るが，戻る時点でのテキストの内容やカーソルの位置などは保存されており，再び ProEdit を呼び出したときに，同じテキストの同じカーソル位置から編集を継続できる．

ProEdit 中で編集中のテキストを Prolog プログラムとして読み込む（reconsult する）には，コマンド ^X^S を使用する．reconsult 中にシンタックス・エラーが発見されれば，カーソルはその場所に移動するので，ユーザはただちにエラーを修正できる．

このように ProEdit では，プログラムの作成・修正→reconsult→実行という，Prolog の典型的なプログラミング過程をきわめてスムーズに行える．また，テキストはエディタ中にローカルに保存されているから，いちいち修正したプログラムをファイルにセーブする必要がない点もプロトタイピングに適している．

ProEdit のテキスト領域は，Prolog-KABA で入出力デバイス edit として指定できるので，それを利用してエディタと実行系がデータをやりとりできる．たとえば，ゴール

```
?-tell(edit), listing, told.
```

を実行すると，現在定義されている述語のリスティングが ProEdit のテキスト・バッファに入り，編集の対象となる．

ProEdit は，約200行の Prolog プログラムでコンパクトに記述されているため，ユーザがそのプログラムを書き換えて，ユーザ好みの環境にカスタマイズできる．たとえば，WordStar に慣れているユーザが，カーソル移動などのキーを変更するのは容易である．

しかし，完全にユーザの好みを反映した環境を整えるには，かなりのプログラミング量を必要とする．たとえば，ユーザが日頃から各種言語のプログラミングにあるエディタ（仮にエディタAと呼ぶ）を愛用しているものとする．その場合，ユーザの立場からは，ProEdit とエディタAとの環境が同一であることが望ましいが，ProEdit をカスタマイズしてエディタAに完全に一致させるのは，エディタAの機能が豊富であればあるほど困難になる．

一つの解決方法としては，次の PCALL で述べるように，汎用エディタを Prolog 処理系と結合する方法がある．

3.1.2 PCALL[7]

日本アイ・ビー・エムの東京基礎研究所では，Prolog 処理系の VM/Programming in Logic（以下 VM/Prolog と略す）と XEDIT と呼ばれる汎用エディタとを結合した環境を実現するために，PCALL と呼ばれるツールを使用している．PCALL は VM オペレーティング・システム上の CMS と呼ばれる会話型環境で使用できる．

PCALL では，XEDIT をフロントエンドとして使用する．すなわち，XEDIT 中から，

（1） Prolog ゴールの実行

（2） 編集中のプログラムの consult

（3） 視覚的デバッガの呼び出し

などが可能である．

PCALL は，Prolog 実行系とエディタの並行実行を実現するのに，CMS の中核拡張領域を用いている．中核拡張領域は，エディタやユーザ・プログラムとは別に，Prolog などのシステムをロードしておくことのできる領域で，ロードされているシステムには，エディタからマクロ言語をとおして命令を与え，実行させることが可能である．また逆に Prolog から，エデ

ィタにコマンドを与え実行することもできる．すなわち，エディタと Prolog 処理系は，それぞれ内部状態を保存したままお互いを呼び出せる．

実行系とエディタとの間でのデータのやり取りについては，CMS のプログラム・スタックと呼ばれる機能を用いる．プログラム・スタックは CMS が管理している通信用のバッファ領域である．エディタのマクロ・プログラムは，プッシュ命令を用いてプログラム・スタックに任意の文字列をプッシュすることや，プル命令でプッシュした文字列を読み込んだりすることができる．VM/Prolog のプログラムからは，プログラム・スタックを入出力デバイスとして使用する．

以下では，VM/Prolog と XEDIT を結合する具体的な方法について，PCALL を簡略化したプログラムを使って解説する．

まず，XEDIT 中から VM/Prolog のゴールを実行するために，XEDIT に PRO という名前のマクロ命令を追加する．PRO マクロでゴールを実行するには，XEDIT のコマンド行に

PRO ゴール

と入力する．PRO マクロは，マクロ・プログラム中で，Prolog 実行系を呼び出し，ゴールを実行し，実行結果を XEDIT のメッセージ行に表示する．たとえば，

PRO sum(1, 2, X)

を入力すると，メッセージ行に

Success : sum(1, 2, 3)

が表示される．

XEDIT にマクロ命令 PRO を追加するには，マクロ・プログラムを記述した PRO XEDIT という名のファイルを作成する．そのファイルを図-1 に示す．このプログラムは，引数の文字列を変数 Goal に受け取り，第1引数を文字列 XEDIT，第2引数を Goal として，CMS コマンド PRO を呼び出している．

CMS コマンド PRO を定義しているファイル PRO EXEC を図-2 に示す．5行目で Prolog 処理系が中核拡張領域にすでにロードされているかどうかを調べ，ロードされていなければ新たにロードし(7〜8行)さらに Prolog プログラム PRO PROLOG を consult する(9行)．11〜16 行で，第1引数が文字列 XEDIT

```
1: /*                                                    */
2: /*  PRO XEDIT                                         */
3: /*                                                    */
4:     parse arg Goal
5:     address CMS 'PRO XEDIT' Goal
6: exit rc
```

図-1 XEDIT マクロ PRO

```
 1: /*                                                    */
 2: /*  PRO EXEC                                          */
 3: /*                                                    */
 4:     parse arg Goal
 5:     'NUCEXT PROLOG'
 6:     if rc ¬= 0 then do    /* load VM/Prolog system */
 7:         'VMPROLOG WS MIXED LANGUAGE ENGLISH',
 8:             'RULE 300K LS 100K GS 200K TRAIL 50K'
 9:         'PROLOG <- consult(pro).'
10:     end
11:     if word(Goal,1) ¬= 'XEDIT' then
12:         XED = 0
13:     else do
14:         parse var Goal . Goal
15:         XED = 1
16:     end
17:     if Goal = '' then     /* call VM/Prolog system */
18:         'PROLOG'
19:     else do               /* execute Goal */
20:         Goal = strip(Goal)
21:         if right(Goal,1) = '.' then
22:             Goal = left(Goal,length(Goal)-1)
23:         'PROLOG <- pro_goal(' Goal ').'
24:     end
25:     RETC = rc
26:     call GetStack         /* output the result */
27: exit RETC
28:
29: GetStack:
30:     do while queued() > 0
31:         parse pull Answer
32:         if XED then       /* output in XEDIT */
33:             address XEDIT 'MSG' Answer
34:         else              /* output in CMS */
35:             say Answer
36:     end
37: return
```

図-2 CMS コマンド PRO

かどうかによって，このコマンドが XEDIT か CMS のどちらから呼び出されたかのフラグ XED をセットする．17〜24 行で，引数としてゴールが与えられているかどうかを調べ，ゴールがあればそれを述語 pro_goal の引数として与え，Prolog 処理系に実行させる(例では〈- pro_goal (sum (1, 2, X)).)．実行結果はプログラム・スタックにプッシュされているので，29〜37 行のサブルーチン中でプルし，フラグ XED に応じて XEDIT のメッセージとしてか，あるいはコンソール出力として表示する．

図-3 が，ゴールを実行し結果をプログラム・スタックにプッシュする Prolog プログラム PRO PROLOG である．述語 pro_goal_init は，プログラム・スタックを出力デバイス名 prostack としてオープンする．述語 pro-goal (G) は，ゴール G を実行し，その結果を prostack に出力，すなわちプログラム・スタックにプッシュする．

次に図-4 に，XEDIT で編集中のテキストを reconsult するためのマクロ命令 CONSULT XEDIT を示す．4〜8 行目で，編集中のファイル・タイプ(フ

```
 1: /*                                                    */
 2: /*  PRO PROLOG                                        */
 3: /*                                                    */
 4: pro_goal_init <-
 5:     dcio(prostack, output, stack, 255).
 6:
 7: pro_goal(<- G) <- / &
 8:     pro_goal(G).
 9: pro_goal(G) <-
10:     call(G) & / &
11:     prst('Success : ', prostack) &
12:     write(G, prostack).
13: pro_goal(G) <-
14:     prst('Fail : ', prostack) &
15:     nl(prostack) &
16:     fail.
17:
18: <- pro_goal_init.
```

図-3 Prologプログラム PRO

```
 1: /*                                                    */
 2: /*  CONSULT XEDIT                                     */
 3: /*                                                    */
 4:     'EXTRACT /FNAME/FTYPE/FMODE/'
 5:     if FTYPE.1 ¬= 'PROLOG' then do
 6:         'EMSG File type should be PROLOG'
 7:         exit 1
 8:     end
 9:     'SAVE'
10:     File = '"'strip(FNAME.1)'"'
11:     address CMS 'PRO XEDIT <- reconsult('File').'
12: exit rc
```

図-4 XEDIT マクロ CONSULT

ァイル拡張子に相当する）が PROLOG かどうかを調べ，PROLOG ならファイルをセーブし（9行），セーブしたファイルに対して述語 reconsult を実行する（11 行）．

ここに示したプログラムは，ごく簡単なものであり，シンタックス・エラーの場所へカーソルが移動しないなど，多くの点で不十分ではあるが（PCALL ではもちろん移動する），エディタをフロント・エンドとしたシステムの構成法の一例を示している．

他システムの例としては，ICOT の Pmacs がある[8]．Pmacs は，EMACS の特徴を多く取り入れたテキスト・エディタで，PSI マシン上で利用可能である．キーボード・マクロ，プログラム・マクロ，ウィンドウ機能などをもっており，文字列出力を行うすべてのシステムが Pmacs のウィンドウをフロント・エンドとして使用し，統一的なユーザ・インタフェースを提供することができる．

3.2 デバッガ

デバッグは，バグを特定し修正する作業と定義できる．その意味では，デバッガといえばバグの修正までするものを指すことになる．実際，そのようなシステムの研究も行われている[9]が，ここでは一般的な意味で，バグの特定を助けるものとして扱う．その意味で，人間の労力をいかに削減できるか，すなわち，システムとの対話を分かりやすく，頻度を少なくできるかということはデバッガの良し悪しを判断する重要な基準といえる．

Prologにおけるデバッガのアプローチは大きく分けると，（1）手続き的な実行手順に従いながらできるだけ計算状態を分かりやすい形でユーザに示し，ユーザの判断を助けるものと，（2）あるアルゴリズムに従ってデバッガが（半）自動的にバグの特定を行うものとに分類できる．前者ではあくまでデバッグの主導権は人間にあるのに対し，後者では，デバッガがプログラムの意図したプログラムの意味を知るのを人間が助けるという立場をとる．以下，それぞれのアプローチによる成果を概説する．

3.2.1 手続き的デバッガ

現在多くの処理系で利用可能なデバッグのための機能としてはボックス・モデル[10]に基づいたトレーサがある．ボックス・モデルでは個々のゴールの実行を，CALL（実行の始まり），EXIT（成功終了），REDO（バックトラックによる再実行），FAIL（失敗による深いバックトラックの起動）の4つのポートをもった箱で捉える（図-5）．基本的に，トレーサは各ゴールの各ポートにおけるインスタンス（instance）を順次表示していくが，多くの処理系では，特定のゴール（spy point）や特定のポート（leashed port）についてのみ表示したり停止して，「次のポートに進む（creep）」「ゴールの実行の詳細を省く（skip）」といったデバッグのためのコマンドを受付けたりすることができる．この種の処理系に組み込まれたデバッガは，デバッグのために少ないリソース（時間的オーバヘッドやメモリの消費量）しか消費せずデバッグの対象となるプログラムに制限がないが，実際の実行の流れに沿ってしか情報を得ることしかできず，必ずしも分かりやすいものとはいえない．

ボックス・モデルによるトレースが分かりにくい原

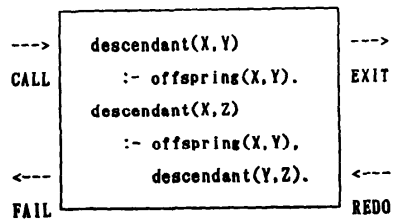

図-5 ボックス・モデル

```
(0) 0 p([1,2,a,b], 3, A)

    p([1,2,a,b], 3, ⇒ [B|C]) :-
          D is 3-1,
          !,
          q(1, B),
          p([2,a,b], D, C).

          (a)
```

```
(0) 0 p([1,2,a,b], 3, [A|B])

    p([1,2,a,b], 3, [A|B]) :-
          2 is 3-1,
          !,
    ⇒     q(1,A),
          p([2,a,b], 2, B).

          (b)
```

図-6 Coda における表示例[11]

因の一つは，ゴールを節，あるいは述語定義全体と無関係に扱ってしまっていることにある．Coda[11] では，実行のトレースをゴールとそのゴールの実行に使われている節のインスタンスの組で表し，また，現在の実行ポイントを示すマーカ，=〉(成功) や 〈= (失敗) を節のインスタンスの表示の中に置くことで分かりやすくしてる．図-6 は，

```
p([1, 2, a, b], 3, NewList)
```

というゴールに対して

```
p([H|T], N, [NH|NT]) :-
        M is N−1,
        !,
        q(H, NH),
        p(T, M, NT).
```

という節が適用された例で，(a)では第2引数までのユニフィケーションが成功したことを，(b)ではカットまで実行が成功したことを表している．実行が進むに応じてデータ構造が作られていくことが分かる．Coda ではそのほかに，spy point の指定の仕方を

```
watch p([a|_], _, _) in q/2
```

のようにある述語の中で呼ばれた特定のパターンまで指定できるようにするといったデバッグ・コマンドの拡張も行われている．

PROEDIT[12] や PROEDIT 2[13] では，ゴールと述語（節の集まり）を対応づけている．PROEDIT 2 では，この関係を BPM (Box and Plane Model) によって明確にしている．BPM では，一つのゴールに対する述語内での計算の推移を一つのプレーンで表す．プレーンは，それを呼び出したゴールに対するインタフェースとして，in-call, out-exit, in-redo, out-fail の4種類のポートをもち，その内部は，ユニフィケーションを表すヘッド・ボックスとその述語内で生成された個々のサブ・ゴールに対応するゴール・ボックスからなる．各ゴール・ボックスはボックス・モデルと同様に4種類のポートをもち，新たなプレーンに対応するというように，計算全体はプレーンの階層構造として捉えられる．

PROEDIT 2 では，述語内のすべてのトレースを一度にプレーンとして表示する．そして，ユーザの指示に従ってゴール・ボックスに対応する新たなプレーンを生成する．表示は，図-7 のように視覚的に分かりやすいものとなっている．PROEDIT 2 はトップ・ダウン的なデバッグを支援する視覚的なトレーサとして位置づけられるが，プレーンの計算にメタ・インタプリテーションを必要とするなど，見方によっては，次々と質問を発する代わりに黙ってプレーンを表示するアルゴリズミック・デバッガともいえる．

TPM (Transparent Prolog Machine)[14,15] では，計算の履歴を AORTA ダイアグラム (図-8) と呼ばれる AND-OR 木を使ってグラフィカルに表示する方法を使っている．個々の AORTA ダイアグラムは一つのゴールに対応し，そのゴールに対して適用された節が OR ノードとしてぶら下がる．節は，その節を

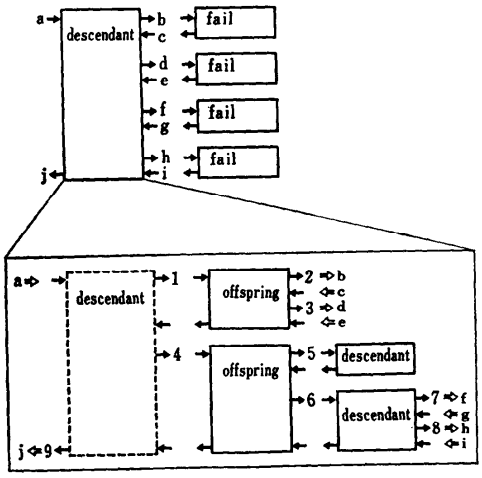

図-7 PROEDIT におけるプレーンの表示例[13]

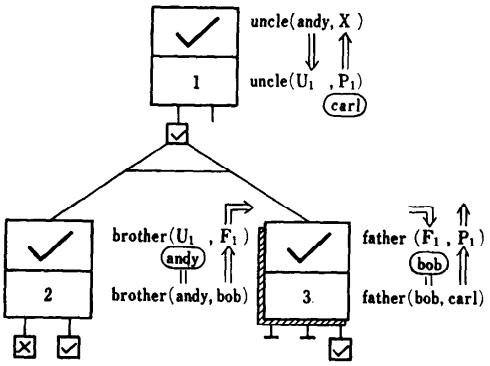

図-8 AORTA ダイアグラム[15]

構成するゴールに対応するAORTAダイアグラムのANDとして表されるが，ヘッドでのユニフィケーションで失敗した節（バー），ボディで失敗した節（ばつ印），成功した節（チェック）などが一目で区別できたり，ユニフィケーションにおけるデータの流れなどが分かるようになっている．AORTAダイアグラムは個々のゴールの局所的な計算の流れを表すが，TPMではこのほかに，LDV (Long Distance View) と呼ばれる計算の履歴全体を表すAND-OR木を表示する機能をもち，このLDVから特定のノードを選択して，AORTAダイアグラムに移行することができる．このほか，再実行，逆戻り，シングル・ステップ実行，指定された条件（成功/失敗，引数のパターンなど）に適合するノードの強調表示など多くの有用な機能が実現されている．

3.2.2 アルゴリズミック・デバッガ

トレーサを使ってのデバッグでは，プログラマがトレースの結果を順次みて，そこにバグがあるかどうかを判断し，次に行う作業を決定していた．Shapiroの研究[9]などによって注目を浴びるようになったアルゴリズミック・デバッギングでは，この過程をプログラムによって半自動化しようとしている．ただし，デバッグの対象となるプログラムだけから途中の計算結果が正しいかどうかを判定するアルゴリズムは存在しないから，完全な自動化はできず，神様（たいていはプログラマ自身が代行する）にお伺いをたて，oracle（正しい答え）をもらうことになる．そこで，できるだけお伺いの回数が少なくなるように，できるだけプログラマが答えやすいお伺いになるようにアルゴリズムを改善することが必要となる．

Prologにおけるバグの症状は，（1）間違った答を返す，（2）予期せぬ失敗，（3）無限ループ，（4）システム・エラーといったものに分類できるが，それぞれに対して診断アルゴリズムが研究されている．

たとえば(1)のケースでは，成功した計算の中に，ボディの中のすべてのゴールはプログラマの意図したとおりに解かれたが，その節を呼んだゴールは意図したとおりに解かれていないという節のインスタンスを発見できる．そのとき，その節そのものにバグがあると考えられる．そのようなケースをみつけるもっとも単純な方法は，計算中に含まれるすべてのゴールについて，それが成功したときに，その結果が正しいかどうかをチェックする方法である．しかし，この方法ではゴールの数 N に比例した手間がかかる．Shapiroは，Divide-and-Queryという $\log N$ の手間ですむアルゴリズムを考案している．計算木の中間点をみつけ，そこで計算が間違っていれば前半にバグがあるとし，正しければ後半にバグがあるとしてバグのある範囲を半分につめていく方法である．しかしこの方法は，お伺いが計算木のどこに対するものかが分からず，しかも必ず「yes/no」で答えなければならないために人間には答えづらいという欠点をもつ．これを緩和する方法として，計算木についてトップ・ダウンに調べていく方法[16)~18)]や，同じ仕様で異なるアルゴリズムの正しいプログラムが利用できるときにはそれを利用することでoracleを得たり，ユーザが入力したパターンをデータベースに蓄えoracleとして利用したりする方法[19)]が研究されている．また，ゴールだけではなく，ゴールの引数に現れる項の依存関係まで扱う方法も考案されている[20)~22)]．間違った解のどの項が間違っているのかをユーザに示してもらうことによって，その項と関連のあるゴールをさらに調べるというものであり，バグの存在範囲を大きく絞ることができる．

アルゴリズミック・デバッガをPrologのデバッガとしてみたときにもっとも問題なのは，多くの場合対象となるプログラムがpure-Prologの範囲で書かれたプログラムに限られるという点である．すなわち，カット・オペレータ，データベースの変更，入出力，メタ述語などが使われてなく，かつ，解の順番などを問題としない場合のみしか利用可能にならない．データベースの更新[13),17)]やメタ述語[13),16)]などについて解決は試みられているが，現実のPrologにはまだ遠いと思われる．

また，実現の方法として，メタ・インタプリテーションを使ったり途中の計算状態を内部データベースに

蓄えたりするために，実行時間やメモリ量といった現実的制限から扱えるプログラムの大きさにも制約ができてしまうという欠点もある．

このように，アルゴリズミック・デバッガが現実に道具として広く使えるようになるには，理論・実現方法両面にわたっての改善が必要とされる．

3.3 コンパイラなど

コンパイラの利用には，最適化処理や機械語生成による高速実行という目的のほかに，プログラムのエラー検出，プログラムの静的情報の収集，ソース・コードの隠蔽，実行モジュールの生成といったことも含まれる．これまで WAM (Warren's Abstract Machine) のアーキテクチャ[23]に基づいた多くの最適化コンパイラの研究が行われているが，高速実行という点以外についてはあまり配慮されていないように思われる．

たとえば，デバッガなどを使って苦労してバグをみつけたところが，「List」という変数を「Lsit」とタイプ・ミスしていたなどということがよくある．この場合，「Lsit」という変数はその節の中で一度しか出現していない可能性が高く，失敗するはずのゴールが成功してしまうといった症状が出る．最近の処理系には，このような一つの節の中に一度しか出現しない変数に対して警告メッセージを出すためのオプションをもつものも多くなってきた．しかしこの種の単純な間違いとしてはほかにも，述語名の間違い，述語呼び出し名の間違い，引数の個数や位置の間違い，定数の間違いなどたくさん考えられる．Prolog の言語仕様上，このような間違いをエラーとして検出することが難しいが，述語の参照関係や変数・シンボルなどの出現などに関するクロス・レファレンスがあればユーザは比較的容易に判断できる．そのようなツールの例として文献 17), 24), 25) などがあるが，このような機能は本来言語処理系自身が備えているべきである．

また，多くの処理系が本当の意味での分割コンパイルの機能，すなわち，個別にコンパイルされた Prolog のオブジェクト・コードをライブラリとして他の Prolog のプログラムから利用可能にする機能を提供していない．そのため，他人の書いたプログラム，ライブラリ・プログラムを利用しようと思えば，ソース・プログラムを自分の環境でコンパイルしなおさなければならない．このことは，多人数で大きなプログラムを開発するときの大きな妨げとなる．

Prolog がプロトタイピングだけでなく，実際のアプリケーションの開発に利用されるためには，このような，C などの開発用の言語では当然な，分割コンパイル，オブジェクト・コードのライブラリ化，デバッグ情報の生成といったことが実現される必要がある．

また，プログラムも大きくなって，実行にもそれなりに時間がかかるようになると，これまでのようにインタプリタでデバッグして最後にコンパイルするという利用形態ではなく，コンパイラをベースにした開発環境が望まれる．インタプリタを使うメリットは，プログラムを変更・実行するサイクルに要する時間が短い，デバッグの機能が利用できるなどの点にあるが，最近の Turbo-C や QuickC といった例にみられる，コンパイラ，エディタ，リンカ，そしてソースコード・デバッガを統合した環境は，コンパイラをベースにしてもそのような対話型の開発環境を提供できることを示しており，Prolog においてもそのような環境が期待される．

4. 今後への期待

本稿では，エディタ，デバッガ，コンパイラなどのツールを中心にプログラミング環境を述べた．今後は，これらを含めた各種プログラミング・ツールが統合化された環境が一般的になるであろう．特に，コンパイラを中心としたプログラミング環境は，Prolog で開発するシステムが大規模になるにつれ，重要になってくると考えられる．

一方，新しいプログラミング・スタイルに対応したツールに関する研究も今後重要になるであろう．たとえば，プログラム変換[26),27)]は，プログラムの「意味」を保存したまま，プログラムを（たとえば，より効率的なプログラムに）変換する．プログラム変換をユーザが簡単に利用できるなら，プログラミングを行う上での有能な「助手」になりうる．このほか，引数の入出力方向の推論，タイプ推論などのシステムも，有効な道具となる．

最終的には，これらのツールがすべて統合され，設計・仕様作成から管理・保守までのサポートを含めた，知的プログラミング環境へと成長していくことが期待される．

本稿以外で，Prolog プログラミング環境について述べている文献としては 28), 29), 30) がある．

参考文献

1) 高木茂行他：SIMPOS のプログラミング環境，情報処理学会オペレーティング・システム研究会資料，27-2 (1985).
2) Chikayama, T.: ESP Reference Manual, ICOT Technical Report 044 (1984).
3) Dömölki, B. and Szeredi, P.: Prolog in Practice, IFIP 83, pp. 627-636 (1983).
4) 江藤博明他：Prolog へのパッケージ・システムの導入，Proc. Logic Programming Conference 87, pp. 175-180 (1987).
5) 柴山悦哉，桜川貴司，萩野達也：Prolog-KABA 入門，岩波書店 (1986).
6) 斉藤康己：拡張可能な画面エディタ EMACS，情報処理，Vol. 25, No. 8 (1984).
7) Maruyama, H. and Hirose, S.: A New Prolog Program Development Environment, IBM Tokyo Research Laboratory Internal Memo.
8) 佐藤裕幸，白須裕之，堀 敦史：SIMPOS のプログラミング環境ーテキスト・エディタ Pmacs ー，情報処理学会第 33 回全国大会論文集，4 D-2 (1986).
9) Shapiro, E. Y.: Algorithmic Program Debugging, MIT Press (1983).
10) Byrd, L.: Understanding the Control Flow of Prolog Programs, Proc. of the Logic Programming Workshop (1980).
11) Plummer, D.: Coda: An Extended Debugging for PROLOG, 5 th ICSLP (1988).
12) 沼尾雅之：PROEDIT, Proc. of the Logic Programming Conference '85 (1985).
13) 森下真一，沼尾雅之：PROLOG の視覚的計算モデル BPM とそれに基づくデバッガー PROEDIT 2, Proc. of the Logic Programming Conference '86 (1986).
14) Eisenstadt, M. and Brayshaw, M.: Graphical Debugging with the Transparent Prolog Machine (TPM), IJCAI '87 (1987).
15) Brayshaw, M. and Eisenstadt, M.: Adding Data and Procedure Abstraction to the Transparent Prolog Machine (TPM), 5 th ICSLP (1988).
16) Lloyd, J. W.: Declarative Error Diagnosis, New Generation Computing (1987).
17) 高橋秀久，柴山悦哉：Prolog のデバッグ支援環境に対する一提案，Proc. of the Logic Programming Conference '85 (1985).
18) Maeji, M. and Kanamori, T.: Top-down Zooming Diagnosis of Logic Programs, ICOT Technical Report TR-290 (1987).
19) Dershowitz, N. and Lee, Y.-J.: Deductive Debugging, ICLP (1987).
20) Pereira, L. M.: Rational Debugging in Logic Programming, 3 rd ICLP (1986).
21) Pereira, L. M. and Calejo, M.: A Framework for Prolog Debugging, 5 th ICSLP (1988).
22) 田村英郎，相磯秀夫：制御仕様を用いた論理型言語のデバッガ，Proc. of the Logic Programming Conference '88 (1988).
23) Warren, D. H. D.: An Abstract Prolog Instruction Set, SRI Technical Note 309 (1983).
24) 溝口文雄他：Prolog とその応用，総研出版 (1985).
25) 加藤暁夫，溝口文雄：論理型言語によるデバッグ支援システムの試作，日本ソフトウェア科学会第 3 回大会論文集 (1986).
26) 吉田紀彦：プログラム変換による自動プログラミング，情報処理，Vol. 28, No. 10 (1987).
27) 古川康一，溝口文雄（編）：プログラム変換，共立出版 (1987).
28) 奥乃 博：知識工学のためのプログラミング環境，情報処理，Vol. 26, No. 12 (1985).
29) 小川 裕：AI プログラミング言語とユーザインターフェース，人工知能学会誌，Vol. 2, No. 2 (1987).
30) 上野晴樹：知的プログラミング環境ープログラム理解を中心にー，情報処理，Vol. 28, No. 10 (1987).

(平成元年 1 月 27 日受付)