記号処理 18－5
（1982. 3. 25）

Ｐｒｏｌｏｇ／ＫＲの概要

中島秀之
（東京大学工学部）

１．はじめに

Ｐｒｏｌｏｇ／ＫＲ〔NAKA 81〕は高度に会話的なプログラミング・システムとして東京大学大型計算機センターのＵｔｉｌｉｓｐ〔CHIK 81〕上に開発された。言語仕様としてはＰｒｏｌｏｇを基本としているが、その欠点をカバーするために大幅に拡張してある。

Ｐｒｏｌｏｇの最大の特徴はそのパターン・マッチング（ｕｎｉｆｉｃａｔｉｏｎ）の機能にある。これは構造を持ったデータを扱う上で非常に便利である。一方、バックトラックの機能は有用ではあるものの両刃の剣の感があるーー制御されない全域的バックトラックは望ましくない。そこでこれを制御するために、多くの処理系ではカット・オペレータを用意しているが、これはあまりにも原始的で、大規模なシステムを構成するのには使いづらい。Ｐｒｏｌｏｇ／ＫＲでは高階の制御用述語を用意してカット・オペレータに代えている。

ＰｒｏｌｏｇやＬｉｓｐといった言語は人工知能的プログラミングに向いていると言われている。そのようなプログラミングでは、アルゴリズムが確定している場合は少なく、むしろプログラムを変更しながらアルゴリズムのテストを行なっている。そういった環境ではプログラムの実行速度もさることながら、そのテストの容易さが問題となる。会話的にプログラムを構成してゆくためには、なによりもまずそのプログラムが動くことが重要である。虱潰しの手段としての自動バックトラックはその意味では非常に有効である。効率の良い制御は、プログラムが動きだしてデータが集まってから、それらを基にして与えてゆけば良い。もちろん、バックトラックが本質であるような部分は最後まで残ることなるであろう。

Ｐｒｏｌｏｇ／ＫＲは上記のようなプログラミングを想定して作ってある。そのようにして出来上がったプログラムではバックトラックが用いられる部分とそうでない部分が混在しており、どちらも重要であると考えられる。

２．Ｓｙｎｔａｘ

Ｐｒｏｌｏｇ／ＫＲはＬｉｓｐと同じｓｙｎｔａｘを採用している。たとえばAPPENDの定義は、Ｐｒｏｌｏｇ／ＫＲでは

```
(ASSERT (APPEND () *X *X))
(ASSERT (APPEND (*A . *X) *Y (*A . *Z)) (APPEND *X *Y *Z))
```

となる。＊で始まるシンボルは変数を表わしている。

このｓｙｎｔａｘには、次の利点がある：

- プログラムの内部表現（リスト構造）と外部表現が１対１に対応しているので、構造エディタの使用が楽である。
- プログラムとデータが完全に同一の形で表現できる。
- 引数の受け渡しに融通性がある。

３．パターン・マッチング

Ｐｒｏｌｏｇ／ＫＲはｕｎｉｆｉｃａｔｉｏｎに基づいたパターン・マッチングの外に、幾つかの機能を持っている。

３．１．実行可能ｐａｔｔｅｒｎ

Ｐｒｏｌｏｇで用いられているｕｎｉｆｉｃａｔｉｏｎは、構造を持ったデータを取り扱う上で非常に便利であるが、その反面、判定を伴った複雑なパターンは記述できない。たとえば、「FOOあるいはBARで始まるリスト」といったパターンは書けないので、個別に

パターンを用意するか、または本体でその判定を行わなければならない。Ｐｒｏｌｏｇ／ＫＲの実行可能パターンを用いれば、これらは解決できる。実行可能パターンとは、！で始まるリストのことである。そのパターン内では、＊だけの変数は特別扱いされる－－＊がまず相手のパターンに束縛された後、！以降の部分が実行される。実行の成否がマッチングの成否と成る。たとえば次のふたつのパターン：

```
((！ MEMBER * (FOO BAR)) POO)
(BAR POO)
```

は以下の手順でマッチする。

①＊がBARに束縛される。
②(MEMBER BAR (FOO BAR)) が実行され、成功する。
③*ZOOがPOOに束縛される。
④全体のマッチングが成功する。

実行可能パターンは関数の代用とすることもできる。Ｐｒｏｌｏｇでは、述語の引数として現われる関数は実行されることはないが、Ｐｒｏｌｏｇ／ＫＲではそこに実行可能 ｐａｔｔｅｒｎを書いておくと、あたかもそれが関数として評価されたような効果を持たせることができる。たとえば、FACTORIALは次のように定義できる（実行可能パターンが入れ子になって用いられているので、！と＊の対応を示すためにそれぞれに数字を付けてある）：

```
(ASSERT (FACTORIAL *N
 (！1 COND ((= *N 0) (= *N1))
           ((TRUE)
            (TIMES *N (！2  FACTORIAL
                            (！3 MINUS *N1 *3) *2) *1)))))
```

その上で(FACTORIAL 3 *F)の実行を追ってみよう。

①*N = 3
②*1 = *F
③(TIMES 3 ()2 FACTORIAL ...) *1) が実行される
④そのためにまず (FACTORIAL ()3 ...) *2) が実行される
⑤そのためにまず (MINUS 3 1 *3) が実行される
⑥*3 = 2
．．．（中略）．．．
⑦*2 = ｆａｃｔｏｒｉａｌ(2) = 2
⑧*F= *1 = ｔｉｍｅｓ(3, 2) = 6

3．2．パターンのマッチングの制限

パターンを'でｑｕｏｔｅすることによって変数とマッチすることを防止できる。ｑｕｏｔｅされたパターンは、変数名を含めて同一のパターンとしかマッチしない。この機能は、たとえば定義の一部を消去する場合に用いられる。以下のような定義を考えてみよう。

```
(ASSERT (P A 1))
(ASSERT (P A 2))
(ASSERT (P B 36))
(ASSERT (P *X 0))
```

Pの第一引数がAのものだけを消去したい時には

```
(RETRACT (P A *ANY))
```

を繰り返せば良いのだが、これではAが*Xとマッチするので最後の

```
(P *X 0)
```

まで消えてしまう。これを防止するためには、Aをｑｕｏｔｅして

```
(RETRACT (P 'A *ANY))
```

としておけばよい。同様に

```
(RETRACT (P '*X *ANY))
```

とすれば最後のものだけが消去される。

4．制御用述語

4．1．　ｉｆ-ｔｈｅｎ-ｅｌｓｅ

Ｐｒｏｌｏｇで条件分岐を表わすには、カット・オペレータ、！を用いて、

```
P：－Q，！，R．
P：－S．
```

のように表わす。しかし、これは

```
P：－ｉｆ　Ｑ　ｔｈｅｎ　Ｒ　ｅｌｓｅ　Ｓ．
```

と書いたほうが分かり易い。Ｐｒｏｌｏｇ／ＫＲでは、これは

```
(ASSERT　P　(IF　Q　R　S))
```

となる。

4．2．否定

Ｐｒｏｌｏｇでは否定はそのままの形では取り扱えない。～Ｐという述語を導入するとＨｏｒｎ節の枠組みからはみ出してしまうからである。たとえば

```
P：－～Q，R．
```

は

$$P \leftarrow Q \wedge R$$

という意味で、これは

$$P \vee \sim Q \vee \sim R$$

と同じであるから、正項がふたつになってＨｏｒｎ節ではない。そこでＰｒｏｌｏｇ／ＫＲでは、手続き的な否定、すなわち「Ｐの実行に失敗する」という意味で(NOT　Ｐ)を導入している（Ｐｒｏｌｏｇでもカット・オペレータを用いて同様のことができる）。

また、Ｐではないという主張もＰｒｏｌｏｇでは取り扱えない。否定の形はゴールとして実行してしまうからである。Ｐｒｏｌｏｇ／ＫＲでは、否定の主張は

```
(ASSERT　P　(FAIL))
```

の様にする。FAILは親の呼び出し（この場合はＰ）を失敗させる働きをもつ。たとえば

```
(ASSERT (FLY 'PENGUIN) (FAIL))
(ASSERTZ (FLY *X) (BIRD *X))
```

としておけば、一般にBIRDは飛ぶがPENGUINは飛ばないという主張になる。PENGUINの前の'は

```
(FLY *X)
```

が最初の主張に捕まって失敗するのを防ぐ働きをする。

4．3．繰り返し

たいていのＰｒｏｌｏｇ処理系では繰り返しはバックトラックを使って実現される。まず、

```
REPEAT.
REPEAT:-!,REPEAT.
```

と常に成功する述語を用意しておき、

```
REPEAT,P,FALSE.
```

とすれば、Ｐを無限に繰り返す。Ｐｒｏｌｏｇ／ＫＲでは、繰り返し用の述語を用意している。

Ⓐ　FOR-ALL

これは　$\forall x$，Ｐ（ｘ）→Ｑ（ｘ）　にちなんで付けられた名前で、

```
(FOR-ALL (P *X) (Q *X))
```

と書けば、Ｐを満たすすべての*XについてＱを実行する。たとえば

```
(FOR-ALL (MEMBER *X (APPLE ORANGE GRAPE))
         (EAT *X))
```

とすれば、(EAT APPLE), (EAT ORANGE), (EAT GRAPE)を実行することになる。

Ⓑ LOOP

LOOPはEXITが実行されるまでその引き数の実行を繰り返す。ｒｅｐｅａｔ－ｕｎｔｉｌやｗｈｉｌｅ－ｄｏはLOOPの特別な場合として定義できる：

```
(ASSERT (REPEAT *S UNTIL *P)
        (LOOP *S (IF *P (EXIT) (TRUE))))
(ASSERT (WHILE *P DO *S)
        (LOOP (IF *P (TRUE)(EXIT)) *S))
```

4．4．すべての解を集める

ある述語を満たすものすべてのリストを作るという作業は、バックトラックでは容易に行なえない。SETOFやBAGOFを用意している処理系もあるが、Ｐｒｏｌｏｇ／ＫＲではACCUMULATEという述語を用意している。

```
(ACCUMULATE *STRUCT *PRED *VAR)
```

は*PREDを満たすすべての*STRUCT（変数でも、構造を持ったデータでもよい）のリストを*VARの値とする。たとえば

```
(ASSERT (INTERSECTION *X *Y *I)
    (ACCUMULATE *Z
       (AND (MEMBER *Z *X) (MEMBER *Z *Y))
       *I))
```

は*Xと*Y（それぞれリスト）の共通要素のリストを求める述語の定義である。

4．5．コ・ルーティン

Ｐｒｏｌｏｇ／ＫＲのコ・ルーティンには、値を生成する部分と、それを消費する部分がある。値を生成する側は、普通のプログラムでよいが、それは幾つか（無限個でもよい）の解を持っている必要がある。消費側は、そのプログラムをINITIATEを用いて初期化し、続いてNEXTで一つずつ値を取り出す。最後に解が無くなれば消費側も失敗する。たとえば

```
(AND (INITIATE (MEMBER *X (子 丑 寅 ...)) *GEN)
     (NEXT *GEN *E1)
     (NEXT *GEN *E2)
        ... )
```

とすれば、*E1＝子、*E2＝丑　...となる。

4．6．並列実行

Ｐｒｏｌｏｇ／ＫＲの基本戦略はバックトラックを伴った深さ優先の探索であるが、場合によっては広さ優先の探索が必要になる。その場合のためにPOR（PARALLEL OR）が用意してある。

Ｐａｌｌａｌｅｌ　Ｐｒｏｌｏｇ〔NAKA 80〕のような完全な並列実行は、現存のハードウェア上では非能率で使い物にならない。

5．多重世界

Ｐｒｏｌｏｇ／ＫＲにはｗｏｒｌｄと呼ばれる、幾つかの独立な述語定義の世界がある。一般にはシステムが用意した述語の世界（これは、どこからでも見える）とユーザーの標準世界（STANDARD-WORLDと呼ばれる）のふたつが存在する。ユーザーはこの他に任意の世界を構築することができる。それらには名前付きのものと名前なしのものとがある。これらの世界は実行時にネストして用いられ、外側の世界から述語の定義を引き継ぐことができる。これによってｉｎｈｅｒｉｔａｎｃｅやｄｅｆａｕｌｔの機構を実現できるものと思われる。

名前付の世界は

(CREATE-WORLD 名前 . 述語定義の並び)
あるいは
(LOAD-WORLD 名前 ファイル名)
によって作られる。そして
(WITH 名前 . 述語呼び出しの列)
によって参照される。システム述語を除いて、自分以外の世界の述語を参照するにはこのWITHを用いなければならない。

名前なしの世界を作るには、WITHの第一引数に名前の代わりに述語定義の列を与える。

Ｐｒｏｌｏｇで非局所変数が必要な場合には述語の形式で値を保存する。たとえばXの値をAにするには

(AND (RETRACT X) (ASSERT (X A)))

を用いる。これとWITHを組み合わせると、Ｌｉｓｐ風の動的スコープを持った非局所変数を実現できる。名前なしWITHで変数の新しいスコープを開く訳である。

6．会話的デバッグ機能

6．1．ｓｔｅｐｐｅｒとｔｒａｃｅｒ

ｓｔｅｐｐｅｒは新しい述語がよばれるごとに停止しながら述語呼び出しを実行する。停止の時には、その述語呼び出しの様子を出力し、ユーザーからのコマンドを受け付ける。ユーザーはそこで以下のようなコマンドを入力できる：

・ステップ実行を続ける。
・正常実行に戻す。
・その時点で呼ばれている述語は止まらずに実行する。
・指定する述語が呼ばれるまでは止まらない。
・実行を終了する。
・エディタを呼ぶ。
・述語呼び出しの履歴を表示する　ｅｔｃ.

このｓｔｅｐｐｅｒは
(STEP 述語呼び出し)
で呼び出される他、エラーやユーザーによる割り込みの際にも呼び出され、デバッグに用いられている。

ｔｒａｃｅｒはいちいち止まらないｓｔｅｐｐｅｒだと思えばよい。もちろん、全部の呼び出しを出力する必要がない場合には、一部のみを出力させることが可能である。

6．2．エディタ

Ｐｒｏｌｏｇ／ＫＲのエディタは構造エディタになっておりプログラムの実行中いつでも呼び出せる。

このエディタはＰｒｏｌｏｇに似たパターン・マッチングの機能を備えている。プログラム中の変数と混同しないように変数が＆で始まる点と、マッチングが一方向である点だけが異なる。パターンは、たとえば全部取り替え（REPLACE ALL）コマンドRAの引数として書け、非常に強力かつ有用である：

RA (FOO &1 &2 . &X) (FOO &2 &1 . &X)

により、FOOの全呼び出しの第一引数と第二引数の入れ替えができる。

また、エディタ・コマンドはEXECMという述語を介してプログラム中から呼べるので、マクロを作る際にはＰｒｏｌｏｇ／ＫＲの全機能が使える（ようになる予定である）。

このエディタは定義体の入ったファイルをエディットするのにも使える。述語名を変更するときなどには、ファイルにたいしてRAコマンドを発行するのが楽であるが、一般にはプログラムを実行しながらのエディットが多いので述語単位に使うことになる。この場合には更新された定義を再びファイルに書き込んでおく必要がある。そのためにＰｒｏｌｏｇ／ＫＲではファイルから定義体を読みこんだ際にそれらを全部記録しておき、
(STORE ファイル名)

で再格納できるようになっている。

6．3．エラーと割り込み

エラーやユーザーからの割り込みの際にはｓｔｅｐｐｅｒに制御が移る。正確には、ｓｔｅｐ　ｍｏｄｅをｏｎにして、ERRORあるいはATTENTIONが呼び出される。ユーザーはこれらの述語を適当に定義することによってエラーや割り込みの際の動作を変更できる。たとえば未定義述語が呼ばれた場合にこれをエラーにしたくなければ

```
(ASSERT (ERROR *MESSAGE *NAME)
        (SELECT *MESSAGE
                ("UNDEFINED PREDICATE" (FALSE))
                (*OTHERWISE (STANDARD-ERROR *MESSAGE *NAME))))
```

とでもしておけばよい。この例からも判るようにエラーの種類はメッセージによって区別されている。STANDARD-ERRORは本来のERRORと同じ動作をする。

割り込みでｓｔｅｐｐｅｒに入った時には、正常の実行に戻すコマンドを入力すれば実行を継続できる。ただし、（ホストのＵｔｉｌｉｓｐの問題により）今のところ割り込みに対する処理能力が充分でないので、多くの場合、実行の継続は割り込みの時点の直後からとはならない。

参考文献（国産品のみ）

〔CHIK 81〕　Ｃｈｉｋａｙａｍａ　Ｔ．：Ｕｔｉｌｉｓｐ　Ｍａｎｕａｌ，METR 81-6，　東京大学計数工学科，　1981

〔NAKA 80〕　Ｎａｋａｓｈｉｍａ　Ｈ．：Ｐａｒａｌｌｅｌ　Ｐｒｏｌｏｇ，　東京大学情報工学・修士論文，　1980

〔NAKA 81〕　Ｎａｋａｓｈｉｍａ　Ｈ．：Ｐｒｏｌｏｇ／ＫＲ　Ｕｓｅｒ'ｓ　Ｍａｎｕａｌ，　東京大学情報工学，　1981