Vol. 25 No. 2 情報処理学会論文誌 Mar. 1984

# 知識表現用言語としての Prolog/KR†

中島 秀之††

Prolog/KR は知識表現用言語として，フレーム理論と述語論理を統合し，統一的な視点のもとに見なおすために設計・開発した言語である．また，総合的なプログラミング・システムとしての機能も備えている．フレーム理論に基づいた言語は，知識表現に必要なさまざまの概念を実現する機能をもっている反面，手続き的記述の面が弱い．一方，述語論理に基づいた Prolog は，後者は申し分ない（セマンティクスが述語論理で与えられている）が，前者の機能に欠ける．Prolog/KR は，Prolog を基本とし，その弱点を補強するとともに，フレームを実現する機能として多重世界を導入した．これにより，概念の階層構造やそれらの間の述語の引き継ぎ等が記述できるようになった．

## 1. はじめに

知識表現用の言語としては大別して，

（i） フレーム理論[7]に基づいたもの

（ii） 意味ネットワーク

（iii） 述語論理

がある．Prolog/KR[10),15)] はこの（i）と（iii）を統合し，統一的な視点（論理学）のもとに見なおすために設計，開発したプログラミング言語である*．Prolog/KR は述語論理に基づいた言語 Prolog[12),13)]を基本としているため，（iii）に（i）をとり込んだという表現が適切かもしれない．

フレーム理論に基づいた言語には FRL[11)]や KRL[1)] がある．これらの言語は知識表現に必要なかずかずの機構をもっている．default や inheritance の機能はその代表的なものであるが，1階述語論理にはこれを（素直に）表現する能力がない．本論文では，これらを Prolog/KR でいかにして表現するかを中心に述べる．

上記の言語には procedural attachment と呼ばれる機能がある．これは，知識が手続き的に表現するのに適した形をしている場合には他のプログラミング言語（主として Lisp が用いられる）による表現を許すものである．しかし，これは他のプログラミング言語に逃げることによって，その部分の意味記述を放棄したことになる（少なくとも Lisp は知識表現用に開発された言語ではない）．知識表現用言語のセマンティクスは自分自身のなかで閉じており，宣言的知識，手続き的知識ともに同一の枠組のなかで表現されるべきである．この意味で，宣言的知識と手続き的知識を区別しない Prolog が優っている．

意味ネットワークにおける情況はもっとひどく，そのセマンティクスは（ユーザが書く）ネットワークのインタプリタの中に隠されてしまっている．

たとえば

$$\text{bird} \xrightarrow{\text{can}} \text{fly}$$

と

$$\text{penguin} \xrightarrow{\text{cannot}} \text{fly}$$

という2組の表現がどういう関係にあるのかは，意味ネットワークが与えるのではなく，その上にユーザが作るプログラム中にコードとして埋め込まれることになる．これでは知識表現用言語とは言いがたい．Levesque らの試み[7),8)]はこの意味ネットワークに統一的なセマンティクス（procedural semantics）を与えようとするものである．

Prolog/KR では，述語論理の長所を生かしつつ，フレームのように知識を一カ所に集約する機構（多重世界）を導入することにより，default や inheritance の機能をとり込んでいる．

## 2. 知識表現

ここでは人間の知っていることのうち，プログラム言語や述語，数式といったフォーマルな形で記述できるものだけを知識として取り扱うことにする．もともと記述可能な知識に対象を限定したわけであるが，そのうえで，表現の容易さ，自然さ，明白さを問題にすることにする．

表現の容易さとは，知識（もともと頭のなかにある）を外部表現に変換する過程が単純であることを意味する．自然さとは表現されたものが，もとの知識と

† Prolog/KR as a Knowledge Representation Language by HIDEYUKI NAKASHIMA (Information Engineering Course, University of Tokyo, Graduate School).

†† 東京大学大学院情報工学専門課程

* 同様の試みは文献2) にもある．

できるだけ1対1の対応を保っていることをいい，明白さとは，表現されたものからもとの知識への逆変換が単純なことをいうことにする．もちろんこれらの概念は互いに独立ではない．

たんに知識を表現するだけなら，既存のプログラム言語 (FORTRAN, Pascal, Lisp 等) でも十分なのであるが，表現の容易さ，自然さ，明白さにおいては問題が多い*．

フレーム理論に基づく言語では，宣言的知識の記述に関しては，これらがすべて満足されている．ところが手続き的知識に関しては，前述のように，他の言語にゆだねられているため，まったくといっていいほどこれらの基準が満たされていない．

意味ネットワークは，容易さ，自然さには問題がないと思われるが，明白さに欠ける[7),8)]．本当のセマンティクスはインタプリタの裏に隠れていて，ネットワーク自身はたんなるノードとリンクの集りでしかない．もっと悪いことには，多く場合このインタプリタは Lisp で書かれており，この Lisp 自身は知識表現には適していない (だからこそ意味ネットワークが必要になった)．

述語論理については，そのセマンティクスには確固たる裏づけがあるので明白さに問題はない．また，本来知識の表現，操作用に生まれた言語であるから他の面での問題も少ない．ただ，1階の述語論理に話を限れば，知識の集約 (モジュール化) や階層化の機能に欠けている．逆に高階述語論理や様相論理などでは，表現された知識の操作 (推論) のメカニズムに問題がある．つまり，現状では計算機で効率よく取り扱える枠を超えている．

そこで，ここでは，1階述語論理を基礎とした Prolog に，述語の階層化の機能をとり入れた Prolog/KR による知識表現を提案する．

## 3. ロジック・プログラミング

ロジック・プログラミングという言葉の定義は定かではないが，一般には Prolog あるいはその類似の言語によるプログラミングを指しているようである．ロジック・プログラミングの特徴は，

(i) プログラミングの意味が述語論理によって裏づけられている．

(ii) 宣言的 (declarative) な記述が可能である．また手続き的な記述との文法的な差がない．

(iii) (ii)の帰結として，各手続きの入出力引数の役割が固定していないので，ひとつのプログラムが多目的に使える (場合がある)．

(iv) プログラミングはそれを実現しているハードウェアと完全に独立である．

などが挙げられる．

Prolog は1階述語論理のサブセットであるホーン節を基本とし，その上での証明手続き (resolution) を計算の実行メカニズムとしている．Prolog のプログラムは一般に次の(a)～(c)の型をしている．

(a) $P\leftarrow$.

(b) $P\leftarrow Q_1\wedge Q_2\wedge\cdots\wedge Q_n$.

(c) $\leftarrow Q_1\wedge Q_2\wedge\cdots\wedge Q_n$.

(a)は P という事実の主張である．*で始まるシンボルは変数を表しており，∀で束縛されているものと考える．

(b)は(a)に条件が付いた型で，Pが成立するためには $Q_1$ から $Q_n$ までが成立していればよいという意味である．逆に $Q_1$ から $Q_n$ が成立していればPも成立すると見てもよい．プログラムとしては，Pを実行するには，$Q_1$ から $Q_n$ までを実行すればよいと読める．

(c)はいささか事情が複雑である．論理的には，

$\sim(Q_1\wedge Q_2\wedge\cdots\wedge Q_n)$

と同じで，$Q_1$ から $Q_n$ までを満たすものはないという主張である．しかし，システムはこれをゴールの否定されたものと解釈し，これから矛盾を導こうとする．この過程がプログラムの実行過程と見なせる[6)]．そして矛盾 (すなわち反例) が見つかると，それが解になるのである．

## 4. Prolog/KR

Prolog/KR[10),15)] は知識表現のための総合プログラミング・システムとして，東京大学大型計算機センターの Utilisp[3)]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 (主として制御構造の貧弱さ) をカバーするために大幅に拡張してある．

ここでは，5章以降の議論に必要な部分のみを紹介する．

### 4.1 文　法

Prolog/KR の文法は Lisp と同じS式を採用している．そして各々のプログラムは何らかの述語で始まる．このため，∧や∨の論理記号も，それぞれ AND,

* もちろん，これらの言語で表現するのが自然であるような知識も存在するであろうが，ここではできるだけ広範囲の知識を対象としたい．

OR という述語となっている.

3章で述べた Prolog プログラムの三つの型はそれぞれ以下のようになる.

(a) (assert P)

(b) (assert P $Q_1$ $Q_2$…$Q_n$)

(c) (and $Q_1$ $Q_2$…$Q_n$)

ここで, P, $Q_1$〜$Q_n$ はおのおのS式で表現された述語呼出しである.

この文法の採用により,

(1) 引数の受け渡しに融通がきく,

(2) プログラムとデータが完全に同一の形で表現できる,

(3) 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[16]の使用が楽になる,

等の利点が生まれる.

### 4.2 述語の定義

述語を定義するのにも, 定義用の述語*を用いる. 一般に用いられるのは assert, asserta, assertz である. これらは,

(assert [a/z] 〈頭部〉. 〈本体〉)

の型で用いられる. たとえば

```
(assert (factorial *n *f)
  (minus *n1 *n1)
  (factorial *n1 *f1)
  (times *n *f1 *f))
```

においては, 1行目の (factorial *n *f) が頭部, 残りが本体である. 呼出し時に頭部のパターンにマッチすると本体が起動される.

assert[a/z] は, これまであった述語の定義に新しいものを追加する働きをする. assert はシステムが定めた適当な位置へ新しい主張**を追加するし, asserta は先頭へ, assertz は最後にそれぞれ追加する.

述語を定義するもうひとつの方法に define を用いるものがある. これは,

```
(define 〈述語名〉
    〈主張1〉
    〈主張2〉
      …)
```

の型で用いられる. 主張の部分からは, 述語名が省かれている. 図1に assert と define の対応を示しておく.

```
(assert (member  *x (*x. *y)))
(assert (member  *x (*y. *z))
        (member  *x  *z))
(define member
    ((*x (*x. *y)))
    ((*x (*y. *z)) (member *x *z)))
```

**図1** assert と define の対応. ともに同じ述語 member が定義されている.

Fig. 1 Two ways of defining a membership predicate.

define はいままであった定義を取り消して, まったく新しいものと取り替える働きをする.

### 4.3 制御用述語

Prolog/KR では制御も知識の一部と考えている. そのため, 制御用述語としては人間の概念とできるだけ1対1になるようなものを導入するようにしている. したがって, 少数の(万能)プリミティブ(たとえばカット・オペレータのようなもの)を用意し, 後は適当に組み合わせて使うという方針はとらない. 2章で述べた表現の容易さ, 自然さ, 明白さに反するからである. また, これらの基準に反しない限り, ロジックからの離脱も許されると考える*.

#### 4.3.1 条件分岐

Prolog/KR で条件分岐を表すには, 基本的には4通りの方法がある.

(1) 主張の頭部のパターンによる分岐. これは Prolog 一般で使われているものである.

```
例: (ASSERT (PLUS 0 *x *x))
    (ASSERT (PLUS (s *x) *y (s *z))…)
```

(2) if を用いるもの. これは

(if 〈condition〉 〈then part〉 〈else part〉)

のように書く. 2分岐の場合に用いる.

(3) cond を用いるもの. これは

```
(cond (〈condition-1〉 〈action 1〉)
      (〈condition-2〉 〈action 2〉)
            ⋮          )
```

のように書く. 3分岐以上に用いる**.

(4) select を用いるもの. これは

```
(select 〈original pattern〉
  (〈pattern-1〉 〈action-1〉)
  (〈pattern-2〉 〈action-2〉)
            ⋮          )
```

---

* Lisp ではすべてが「関数」と呼ばれているように, ここでもすべて「述語」と呼ぶことにする.

** Prolog/KR のプログラムのうち, assert [a/z] によって追加される単位を主張と呼ぶことにする. これはホーン節に対応している. そして, ある述語に関する主張の全体を定義と呼ぶことにする.

* 副作用を無視するなら, いつでもロジックの式に書き直すことは可能なので, 離脱というほどおおげさなものではない.

** (2)は(3)の特殊な場合であるから, 原理的には不要であるが, (2)を(3)で書くのは不自然である.

のように書き，パターンによる分岐に用いる．

#### 4.3.2 くり返し

くり返し用の述語としては loop と for-all がある．loop はその引数を，exit が実行されるまでくり返す．たとえば，

```
(loop (read *x)
      (if (member *x (yes no))
          (exit)
          (print yes-or-no?)))
```

によって yes か no を読み込むまでループする．ここで *x の値が毎回変わりうる点に注意してほしい．

for-all は

```
(for-all (member *x *list)
         (p *x))
```

のように用い*，第1引数を満たす全部の解について第2引数以降を実行する．ただし，第2引数以降の実行に失敗すると，for-all も失敗する．これは，

```
(for-all (p *x) (q *x))
```

のセマンティクスを

$$\forall x p(x) \rightarrow q(x)$$

としたため，前件が真なのに後件が偽の場合は，全体が偽となるからである．

## 5. Prolog/KR の多重世界メカニズム

Prolog/KR には，述語定義の間に階層性を持ち込むために多重世界メカニズムがある．

各「世界」は基本的には独立しており，他の世界の述語定義は見えない．これによって modularity を保障している．他の世界に入るには，組込み述語 with を用いる．with を用いることによって，いくつかの世界の間に階層構造を作ることができるが，この階層構造は動的に決まる**．

ある世界から外側の世界の述語定義が見えるかどうかを制御することにより，述語の inheritance の制御が可能になる．

世界には名前付きのものとそうでないものがある．名前付きの世界はその名前で呼ばれるが，最初は，

```
(with 〈name〉
  (assert (p…)…)
  (assert …)
      …)
```

* この例は Lisp の mapcar に相当する．
** 動的である理由は後述．

```
(with human
    (assert (has head))
    (assert (has body))
          …
    (assert (age *n)
        (today *year1 *month1 *day1)
        (date-of-birth *year2 *month2 *day2)
        (cond ((> *month1 *month2)
               (- *year1 *year2 *n))
              ((< *month1 *month2)
                    …
```

**図 2** human フレーム
Fig. 2 The predicate "with" used to construct "human" frame.

のようにして作られる．その後はこの名前で参照され，

```
(with 〈name〉 (p…))*
```

のように呼ばれる．名前なしの世界**を作るのにも with を用いるが，名前の代りに述語定義の列を並べる．

```
(with ((p(…)…)   ┐
       (q…)      ├ 定義
        …)       ┘
  (p…))          ┘ 呼出し
```

これのおもな利用法は，Lisp のような動的スコープをもった非ローカル変数の機能を実現することにある．すなわち，Prolog では非ローカル変数の機能を述語定義（述語名が変数名，引数がその値）を用いて実現しているわけだが，with を用いることにより，これに動的スコープを持ち込むことが可能になる．上の例は，Lisp の

```
(let ((p(…))
      (q…)
       …)
     (f p))
```

に相当すると考えられる．

次に，多重世界メカニズムをフレームの表現に用いる方法について述べる．

おのおのの世界をひとつのフレームと考える．たとえば human という世界を定義し，そこに human に関する主張を集約するのである（図 2）．ここで注意したいのは，Prolog においては，手続き的（procedural）*** な知識と宣言的（declarative）な知識の表現に

* この p は，直接参照することはできない．
** 名前なしの世界は，他から参照の手段がないので，1回外に出ると消滅する．
*** ここでいう手続き的な知識とは，知識を取り出すのに手続きを使うという意味である．KRL ではこれを procedural attachment と呼んで特別扱いしている．

差がないことである．たとえば，ある人の年齢を知るには，その人の生年月日から計算すればよいという知識は，図2の age という述語に表現されている．date-of-birth という述語は宣言的な知識を表しているが，これは human の属性ではなくて，各個人 (human の下位概念*) の属性である．

## 6. 述語定義の引継ぎ

Prolog/KR の多重世界メカニズムが実世界の構造を反映するためには，各世界間での述語定義の引継ぎ (あるいはその禁止) の制御が必要になる．各世界はおのおのひとつの概念に対応していると考えるわけだから下位の概念は上位の概念のいくつかの性質を引き継ぐわけである（しかし，全部を引き継ぐとは限らない)．

いくつかの世界が，

```
(with w1
  (with w2…)
    …)
```

のようにネストして用いられた場合，内側の世界（下位概念に対応）は原則として外側（上位概念に対応）の述語定義を引き継ぐ．この例外は，内側の世界で define を使って積極的に述語定義が変更されるか，見えなくされた場合**である．

この機構を用いれば，たとえば bird の世界で，

```
(assert (can fly))
```

としておけば，その下位の swallow や hawk でいちいち同じことを繰り返さなくてもよい．また，penguin や ostrich が飛べないことをいうには，それぞれの世界で，

```
(asserta (can' fly) (fail))
```

とすればよい．ここで，asserta は定義の最初に追加することを示し，fail は (can fly) の呼出しを失敗させる働きをする．fly の前の ' は，たとえば，penguin のできることすべてを数え上げる試み

```
(for-all (can *a) (print *a))
```

がこの主張につかまって fail することがないようにするためである（詳細はマニュアル[10]参照のこと)．

この (can fly) という記述は鳥に対する default であると考えてよく，とくにそれに反する知識がない限り，鳥は飛ぶとしてよい．それを打ち消すのが，前出の記述である．

ところで多重世界の階層構造は動的に決まるのであるから，たとえば penguin は飛べるのかという問いは，

```
(with animal
  (with bird
    (with penguin (can fly))))
```

のように提示されなければならない．これを毎回行うのは大変であるうえ，動物の世界の静的階層構造を反映しているとはいいがたい．そこで，静的階層構造を実現する必要がある．いわゆる is-a 階層構造である．

is-a という述語を次のように定義する．

```
(assert (is-a *low *high)
  (assert (with-world *low *predicate)
    (with-world *high
      (with *low *predicate))))
```

こうすると，たとえば

```
(is-a penguin bird)
```

を実行すれば，

```
(assert (with-world penguin *predicate)
  (with-world bird
    (with penguin *predicate)))
```

が実行され，with-world という述語にこの主張が追加される．

```
(is-a bird animal)
(is-a animal class)
```

も実行してあるとすると，

with-world の定義は，

```
(assert (with-world penguin *predicate)
  (with-world bird
    (with penguin *predicate)))
(assert (with-world bird *predicate)
  (with-world animal
    (with bird *predicate)))
(assert (with-world animal *predicate)
  (with-world class
    (with animal *predicate)))
(assert (with-world class *predicate)
  *predicate)
```

となっている*から，それ以降は

```
(with-world penguin (can fly))
```

とするだけで，これは，

---

* Prolog/KR による表現では，クラスと個体の区別はない．

** (define ⟨name⟩ nil) によって述語定義が取り消される．

* 最後の，class に関する主張（つまり，class が最上位概念であるということ）は最初に入れておくものとする．

```
(with human
    (assert (age *n)
        (today *year 1 *month 1 *day 1)
        (date-of-birth ···)
        (cond ···)))
(with p1 (assert (date-of-birth 1952 11 14)))
(with p2 (assert (date-of-birth 1955  6  8)))
            ···
(is-a  p1 human)
(is-a  p2 human)
(with p1 (age*n))  ——  *n=29
(with p2 (age*n))  ——  *n=26
```

図 3 human で使われる date-of-birth の定義が呼出し方で異なる例
Fig. 3 A simple example to show the polymorphism of a predicate definition "date-of-birth".

```
(with animal
  (with bird
    (with penguin
      (can fly))))
```

と展開されて，実行される．

多重世界間の動的階層構造を利用することによって Smalltalk[14] 風の polymorphism が実現できる．polymorphism というのは，あるメッセージ (Prolog/KR では述語) が送られる対象 (使われる環境) によって多様に変化することをいう．

この最も単純な例は図 3 にある．これは先に述べた年齢を計算する例であるが，これに用いられている date-of-birth という述語の定義 (この場合はたんなるデータであるが) は with のネスティングの状況によって異なってくる．また，today の方は，ネスティングに依存しない普遍的なものである*．

上の例はあまりにも単純であるが，実際にはもう少し複雑なことが起こる．ソートのプログラムでは，ソートの対象によって，>の意味が異なるであろうし，pretty printer では打ち出す object によって print の動作が異なる．従来の言語では，それらはすべて呼び出す側 (ソートのプログラムや pretty printer) が知っていなければならなかったが，Smalltalk や Prolog/KR では，それらのプログラムを引き継いだ側でコントロールできるようになっている．この機能がなければ，上位の概念からのプログラム引継ぎは，かなり限定されてしまう．それを避けるには，上位の概念は，自分の下位に何があるかを全部把握していなければならない．

* ただし，過去の何年何月何日の年齢を知りたいというような場合には，

```
(with ((today ((1952 11 14))))
  (with-world 中島秀之
    (age *n)))
```

のようにすればよい (上の例では *n=0 である)．

## 7. おわりに

知識表現用の言語として見た Prolog/KR の側面について述べた．

現在，本システムを使用し，抗生剤選択支援システム Anticipator[5] を作る作業が進行中である．Prolog/KR の言語仕様は，そのフィードバックを受けて現在も細部が変更されつつある．多重世界間の述語引継ぎに関しても，もう少しきめ細かな制御が必要かもしれない．

## 参考文献


1) Bobrow, D. G. and Winograd, T.: An Overview of KRL-0: A Knowledge Representation Language, *Cognitive Sci.*, Vol. 1, No. 1, pp. 3-46 (1977).
2) Charniak, E.: A Common Representation for Problem-Solving and Language-Comprehension Information, *Artif. Intell.*, Vol. 16, No. 3, pp. 225-255 (1981).
3) Chikayama, T.: Utilisp Manual, METR 81-6, Dept. of Mathematical Engineering, University of Tokyo (1981).
4) Hughes, G. E. and Cresswell, M. J.: *An Introduction to Modal Logic*, Hethuen and Co., Ltd., London (1968).
5) Kimura, M.: Construction of An Antibiotic Medication Counselling System by Means of Knowledge Engineering, Master Thesis, Information Engineering Course, University of Tokyo (1982).
6) Kowalski, R. A.: Predicate Logic as Programming Language, Proc. of IFIP Congr ess, pp. 569-574, North-Holland, Amsterdam (1974).
7) Levesque, H. J.: A Procedural Approach to Semantic Networks, *Tech. Rep.*, No. 105, Dept. of Computer Science, Univ. of Toronto (1977).
8) Levesque, H. J. and Mylopoulos J.: A Procedural Semantics for Semantic Networks, in Findler, N. V. (ed.), *Associative Networks*, pp. 93-120, Academic Press Inc., New York (1979).
9) Minsky, M.: A Framework for Representing Knowledge, in Winston, P. (ed.), *The Psychology of Computer Vision*, pp. 211-277, McGraw-Hill, New York (1975).
10) Nakashima, H.: Prolog/KR User's Manual,

METR 82-4, Dept. of Mathematical Engineering, University of Tokyo (1982).
11) Roberts, R. B. and Goldstein I. P.: The FRL Manual, MIT AI-memo, 409 (1977).
12) Sowa, J. F.: A Prolog to Prolog, IBM Systems Research Institute (1981).
13) Warren, D. H. D., Pereira, M. L. and Pereira, F.: Prolog—The Language and Its Implementation Compared with Lisp, Proc. of the Symposium on Artificial Intelligence and Programming Language, SIGPLAN Vol. 12, No. 8/SIGART No. 64, pp. 109–115 (1977).
14) The Xerox Learning Research Group: The Smalltalk-80 System, *Byte*, Vol. 6, No. 8, pp. 36–48 (1981).
15) 中島秀之：Prolog/KR の概要，情報処理学会記号処理研究会 18-5, pp. 69–74 (1982).
16) 中島秀之：拡張可能な汎用S式エディタ AMUSE，情報処理学会第25回全国大会予稿集，pp. 541–542 (1982).

(昭和57年9月27日受付)
(昭和58年9月13日採録)