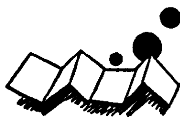
解　説



Prologによるエキスパートシステム†

溝　口　文　雄††

1. は じ め に

現在のエキスパートシステムは，LISP を用いて開発されている．そして，LISP の発達は，エキスパートシステムの開発を促進してきた．また，LISP の機能，例えば，表示機能や，スペルの訂正機能を導入して，ユーザとのインタフェースを考慮したエキスパートシステムが作成されている[1]．その他，FORTRAN 等の言語で作成されたエキスパートシステムもある[2]．

LISP に対して Prolog によるエキスパートシステムの開発は，自然言語解析で Prolog が使われているほどの報告がなされていない．また，Prolog の応用については，Hungary[3] で活発に行われているが，文献そのものが広く普及していないこともあって，Prolog のエキスパートシステムへの適用には不透明な部分が多いようである．本稿は，我々の経験にもとづいて，Prolog のエキスパートシステムへの応用の可能性について述べたものである．そして，Prolog のもつ記述力や応用性に対する前述の不透明な部分を明らかにすることが本稿の動機でもある．

本稿の構成は，エキスパートシステムの構築についての考え方を最初に述べる．次に，この考え方を Prolog でどのように実現していくかに触れる．ここでは，ルール指向型のエキスパートシステムを中心にしている．次に，対象領域の知識をオブジェクトとして表現するオブジェクト指向の考え方を述べる．終りに，エキスパートシステムの今後の動向と Prolog との関係を検討し，それがどのような形で実現されていくかについて考察する．

なお，本稿の Prolog シンタックスは DEC-10 の Prolog[4] を用いていることに注意されたい．

† The Design of Expert Systems Using Prolog by Fumio MIZOGUCHI (Science University of Tokyo).

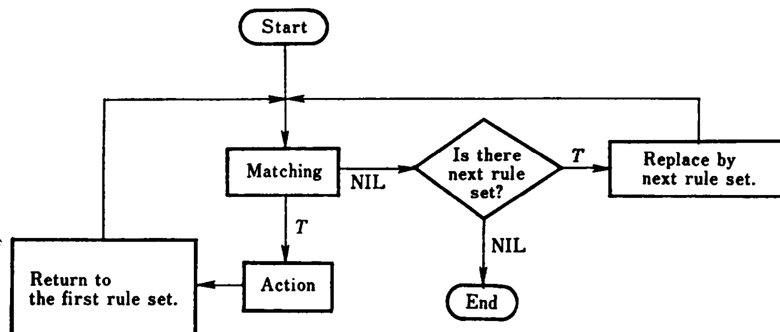
†† 東京理科大学理工学部経営工学科

2. エキスパートシステムの構築について

自然言語処理と異なって，エキスパートシステムを構築するときの理論的枠組は明確な形では存在していない．専門家（エキスパート）の知識を，コンピュータのプログラムとして利用し，専門家なみの性能を実現していくのがエキスパートシステムとすると，そのシステムの構築は経験的な方法に頼る部分が多い．と言うのは，知識の方略は，種々の領域で，それぞれの特徴を有しているからである．

現段階では，エキスパートシステムの構築についての方法論は，知識利用のパラダイム (paradigm) の考え方を採用している[5]．もちろん，このパラダイムが確立するまでは，ゲームやパズルを対象にした時代があった．その中から，人間の問題解決のモデルとして，プロダクションシステムが Newell, A. によって提案された．この方法が，医療診断システムの作成に応用され，ルールによる知識利用のパラダイムとなった．したがって，ルールベースのアプローチは，エキスパートシステムを構築するときの知識利用のパラダイムのひとつと言うことができる．

ルールベースシステムは，“もし～ならば～せよ” (IF～, THEN) の手続きで表わされた知識を処理するためのシステムである．こうしたシステムをプロダクションシステムと言う場合もある．従来，ルールベースシステムは記号処理言語の LISP を用いて開発されてきた[6),7]．LISP でのルールベースシステムの特徴は，“IF～, THEN” で記述されたルールを適用するためのインタプリタを作成するところにある．このルール適用の流れは，図-1 に示したようになる．ルールの条件部，すなわち，IF 部が入力データと一致していれば，ルールの実行部の THEN 部を適用するものである．IF 部には，例えば，医療の場合に，“体温が 40℃ 以上あるかどうか” が書かれている．また，THEN 部には，“その時は，カゼを疑え” という IF 部に対応した結論を実行する内容がある．こうした




```
(PS-TOP
(LAMBDA NIL
        (* PS-TOP CONTROLS SYSTEM FLOW OF PURE PRODUCTION SYSTEM)
        (PROG (RULES MODEL RULES* DB RULE RULE-NAME)
          LOOP 1
                (START)
          LOOP 2
                (SETQ RULES* RULES)
          LOOP 3
                (COND ((NULL RULES*)
                        (COND ((END) (GO LOOP 1)) (T (RETURN NIL)
                (SETQ RULE  (CAR RULES*))
                (SETQ RULE-NAME (CAR RULE))
                (* IF PATTERN-MATCH-DB RETURNS TRUE ACTIONS OF
                   THE MATCHED RULE IS ACTIVATED)
                (COND ((PATTERN-MATCH-DB (CADR RULE))
                        (ACTIVATE-ACTIONS (CAR (CDDDR RULE)))
                        (PRINT-RESULT)
                        (GO LOOP 2)))
                (SETQ RULES* (CDR RULES*))
                (GO LOOP 3))))
```

図-1 LISP によるルールベースシステムのルール適用部基本関数

ルールが集合になっており，そのルールを上から下に検索する形でルールが適用されていく．これがもっとも単純なルールベースシステムである．LISP でルールの適用部を書くためには，IF 部のルール名を第1要素とし，条件項を第2要素とすれば，それぞれのリストの要素を，CAR や CADR により取り出すというリスト処理と，また，CADR (第2要素の条件項) とデータが一致しているかのマッチング処理が主体のプログラムとなる．IF 部の条件要素のリスト処理やマッチングを LISP できめ細かくプログラムを書くことができる．逆に，システム全体からみると，この部分の処理が大きな割合となってくる．

LISP に対して Prolog で，ルールベースシステムを作成するとどのような特徴を持つのであろうか．直観的には，Prolog は次に示すようなホーン節の形式でプログラムが書かれている．

A:−$B1$, $B2$

この形は，“A は $B1$ と $B2$ の問題に分解できる” あるいは，“A を解くためには，$B1$ と $B2$ を満足していなければならない” と読むことができる．つまり，ホーン節は基本的には，次のプロダクションルールと同じ形式を持つ．

A if $B1$ and $B2$

したがって，Prolog とルールベースシステムとは，ほとんど同一の処理内容であると言える．LISP のように，IF 部や THEN 部の適用プロセスが，Prolog ではホーン節の実行プロセスに対応する．その意味では，Prolog はルールベースシステムとみなすことができる．ただし，実際には，入力データの読み込み等を考えると，種々の工夫を要する．

結果的には，Prolog はルールベースシステムと同一と考えられるが，Prolog でルールベースシステム

を作成するというアプローチも考えられる．つまり，Prolog をプログラム言語とみなすか，あるいは，推論システムとみなすかで，エキスパートシステムの構築方法が異なってくる．

前者の立場では，Prolog の持つ記述力や柔軟性が問題になり，特に，ルールベース以外の知識利用のパラダイム，例えば，フレームや意味ネットの表現に対しては，どのようなアプローチとなるかが議論の中心となる．後者の立場では，ユーザとのインタフェースやルールの記述方法が問題になってくる．

ここでは，最初に，前者の立場でエキスパートシステムをどのように構築していくかに触れ，次に，後者の考え方でシステム構築をどのようにすればよいかを検討する．

3. Prolog をプログラム言語と考えたときのエキスパートシステムの構築について[8]

Prolog を用いてルールベースシステムを作成することは，LISP に比べれば効率は良い．Prolog によるルールベースシステムの基本部を示したのが，次のプログラムである．

```
try_rules (Rule, Conditions, Facts) :-
    action (Rule, Actions), print (Rule, Actions),
    set_value (Facts, Conditions, Actions, New
    Facts), search_rule (NewFacts).
```

このプログラムは，ルールを適用する部分を示したものである．

節の頭部のうち，第1引数はルールの番号，第2引数は，ルールの条件部，Facts は入力データを示すものとする．この述語は，あるルールを取り出して，そのルールの条件部の Conditions と入力データとがマッチした結果，ルールの条件部の Action を実行するためのものである．節の本体では，ルールの番号に対応した action のうちから，Actions を取り出してそれを印刷 (print) する．set_value は Actions の結果を Facts (知識ベース集合) に加えて，新しい集合，NewFacts とする．その集合のなかから，未処理部データのために，再帰的に search_rule を実行する．この述語は，次のようなプログラムである．

```
search_rule (Facts) :-
    premise (Rule, Condition), unify (Condition,
    Facts), try_rules (Rule, Conditions, Facts).
```

この述語は，あるルールを取り出して，その条件部 (premise の第2引数) と Facts とが一致するかを，述語 unify でチェックする．その結果，try_rules でマッチしたルールの条件部を駆動するというものである．try_rules と search_rule の説明が逆になってしまったが，ルールのインタプリタの中心部の述語が，この二つである．

また，unify は，次のようなリストのマッチングを調べる述語である．

```
unify ([ ], _) :- !.
unify ([Condition| Rest], Facts) :-
\+(is_list (Condition)), member (Condition,
Facts).
unify ([Condition| Rest], Facts) :-
unify (Condition, Facts), unify (Rest, Facts).
```

述語の内容は，以下の通りである．

イ．条件部が空 [ ] のときはストップ

ロ．Condition| Rest
　CAR 部の条件項と Facts とがマッチするかをみる．このとき Condition がリストでないときは，Facts の中に条件項があるかをみる．

ハ．Condition| Rest
　Condition がリストのときは，リストの CAR 部をとって，Facts とマッチするかどうかをみる．そして，Rest について同様の操作を繰返す．

その他，try_rules で使われた set_value は，次の内容の述語である．

```
set_value ([Fact| Facts], Condition, Actions,
    NewFacts) :- member (Fact, Condition),
    set_value (Facts, Conditions, Actions, New
    Facts).
set_value ([Fact| Facts], Conditions, Actions,
    NewFacts) :-
    set_value (Facts, Conditions, Actions, NewF),
    append ([Fact], NewF, NewFacts).
```

述語の意味は，以下に示す通りである．すなわち，Fact が Conditions 部のメンバであるかを調べ，もしメンバであれば，Facts について調べ，Actions を新しい NewFacts とする．

以上のルールベースシステムは，前向き推論 (Forward Reasoning) のもっとも簡単なものである．処理の流れは，図-2 に示すような Fact (入力データ) によるデータ駆動型のルールベースシステムである．

我々が試作した Prolog によるルールベースシステ

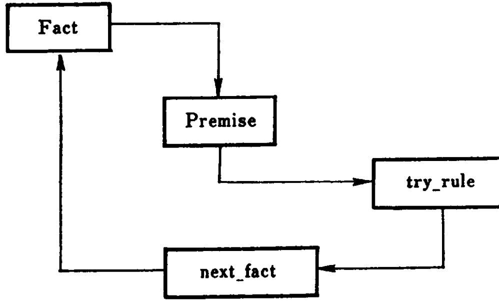


図-2 Fact 駆動型のルールベースシステムの制御の流れ

ムの APLICOT[9]の原形は，以上のような制御構造を持つ．最初の試作システムでは，LISP から Prolog へのプログラミング技術が未完成だったために，プログラムのステップ数にも，むだな部分が多かった．しかし，tail-recursive なプログラムに慣れるにつれて，プログラム構造そのものは整理された形となってきている．LISP では，プログラムコードを積み上げるようにして作成していくために，プログラムのコード数そのものは増加していくのに対し，Prolog では，プログラム技術により，よりコンパクトにまとめることが可能である．Shapiro, E.[10]によると，Prolog のプログラミング技術は，現在でも，未開発な部分が多く，また，それだけに，新しいエキスパートシステムを作り出す可能性を潜めていると考えられる．

4. Prolog を推論システムと考えたときのエキスパートシステムの構築について[11]

Prolog をプログラム言語としてでなく，ルールを実行する推論システムとみなしたときには，ルールの記述形式をどのようにするかが重要である．もちろん，Prolog そのものの推論システムは，前節でみたように逆むき推論 (Backward reasoning) となる．この形式は，次のようなルールで書くことができる．

```
may_have (X, Y) :-
    has (X, Z), suggests (Z, Y).
```

このルールの意味は以下の通りである．すなわち，“X が Y であるためには，X が Z という性質を持ち，かつ，Z が Y を示唆することである”．具体的に，次のようなデータを Y, Z に入れてみればよく分る．

```
may_have (X, [gramstain, 'Grampos']) :-
    has (X, [morphology, coccus]),
    suggests ([morphology, coccus],
              [gramstain, 'Grampos']).
```

MYCIN に対応したルールの記述である．読み方としては，以下の通りである．

IF: the morphology of organism is coccus, then, there is suggestive evidence that the gramstain is Grampos.

Hammond, P. らの apes (Augmented Prolog for Expert System) のルールは上記の形式によっている．apes は Micro-Prolog で作成されている[12]．

Prolog を推論システムとみなして，エキスパートシステムを構築するときの問題点は，説明機能の実現であろう．MYCIN における why や how といった推論のプロセスを表示することがルールベースシステムでは必要である．Prolog そのものには，もちろんこうした機能を持っていないので，why や how のような述語を準備して，逆むき推論のプロセスを取り出せるようにしなければならない．この点に関しては，apes は Micro-Prolog を用いて，why, why not および how といった機能が準備されている．apes のマニュアルにはこうした機能の詳しい説明はない．しかし，Clark, K. が編集した Micro-Prolog の著書における第11章には，細かい説明が Hammond, P. によりなされている．ここでは，Hammond の作成した how の機能を拡張して次のような述語を用いて，推論のプロセスを出力することができることを示す．

```
how (Sentence) :-
    parse_of_ConjC (Predicate, Sentence),
    clause (predicate, Body),
    solve_demo (Body, [ ]),
    is_shown (Sentence, (Predicate :- Body)),
    explained (Body), nil, nl.
```

述語 how の内容は以下に示す通りである．

1) センテンスから Predicate を生成する．
2) その Predicate が知識ベース内にあるかどうか．
3) その Predicate を証明することができるかどうか．
4) その Predicate のリスティングを出力する．
5) その Predicate を英語で表示する．

この述語の how 中で使われる demo は Bowen & Kowalski が提案した demo 述語と同じ意味である[13]．したがって，現在システムが持つ知識ベース内にあるルールから，how に対応するルールが存在するかどうかを demo によりチェックされる．つまり，メタレベルからみて，現在の how が，知識ベース中の節集合

から証明可能かどうかを検討する述語である．how の実際の動きは，以下に示したような流れとなる．

how 内の処理：
sentence=[taro, should_take, aspirin]
"太郎はアスピリンを飲むべきである"

① parse_of_Conj*C*
入力文中の述語を取り出して，次の形に分解する．
should_take (taro, aspirin)

② clause
should_take (taro, aspirin) とユニファイする clause が知識ベース中にあるかどうかをチェックして，あれば，その Body を取り出す．
Body=(has_complained_of (taro, pain) & suppress (aspirin, pain) & not (is_unsuitable_for (aspirin, taro))).

③ solve_demo
第1引数に Body 部が入る．知識ベース中には，has_complained_of (taro, pain) が存在するので，solve_demo は成功する．

④ is_shown による結果の出力
"taro should_take aspirin"
どのようなルールを用いたかが，次のようにして示される．

```
should_take (taro, aspirin) :-
  has_complained_of (taro, pain) &
  suppresses (aspirin, pain) &
  not (is_unsuitable_for (aspirin, taro)).
```

以上の処理のうち③が成功しない場合には，ユーザに，次のような問い合せを行う．

"taro has_complained_of *X*"

これに対して，is_reported が confirmed かまたは，denied かどうかをユーザに問い合わせる．

以上のようにして，ルールがどのようにして適用されるかを実行する述語 how を作成することができる．Prolog をプログラム言語と考えるのではなく，推論システムと見なしてエキスパートシステムを設計するときの問題点は，知識ベースの構築と，説明機能をどのように対応づけていくかが，むずかしい．そして，Prolog そのものの推論システムの構造は，逆むき推論であり，前向き推論にするためには，なんらかの工夫が必要である．例えば，ホーン節の頭部と本体を逆にするような変換が必要であろう．その結果，条件項から，目標項を解くための前向き推論を実現することができる．

次に，逆むき推論を用いたエキスパートシステムの具体例をみてみよう．

Clark & McCabe[14] は Prolog を推論システムと考えた時の，簡単な故障診断システムを，次の述語で作成できることを示した．

```
1) certainty_fault ([A|B], [X|B]) :-
     is_all (L, [S, F of A, P],
     is_syndrome_for (S, F of A, P)),
     gives_certainty_of_symdrome (L, X).
2) certainty_fault ([A|B], X) :-
     is_part_of (Y, A),
     certainty_fault ([Y, A|B] X).
```

1) の述語は，次のような内容である．

ある部品が与えられると，その部品の故障原因を検索し，原因に対して表われる症状の CF 値をユーザに入力してもらい部品に故障があるかどうかを判断する．

[*A*|*B*] は部品のリスト，[*X*|*B*] は故障原因と CF 値のリストを，また，*S*, *F*, *A*, *P* はそれぞれ，症状，故障原因，部品，CF 値を表わしている．

2) の述語は，次のような内容である．

ある部品に対して，その下位の部品を検索する．

以上のようにして，部品のもつ階層構造から，故障部品を診断する述語を定義することができる．ある部品の故障が，それに関連する部品の故障につながるような故障には有効な考え方である．また，Prolog の持つ論理性を十分に生かした故障診断と考えることもできる．

5. Prolog を用いてオブジェクト指向を導入したエキスパートシステムの展開

Prolog をプログラム言語としてみたときに，LISP のようにデータ構造を書いて，例えば，フレームのような知識構造を表現するには相当の努力を要すると言われてきた．このことは，Prolog は言語自身が平板で，構造がないという結果に対応している．ルールベースシステムについては，Prolog が適していることが直観的にもよく分る．しかし，フレームのような構造を扱うことは，Prolog ではむずかしいという見方も自然である．

しかし，一方では Prolog がデータベースの作成に適している．このことは，Prolog のプログラムの例

```
(<FRAME NAME>(<SLOT 1>(<FACET 1>(<VALUE 1>)
                                (<VALUE 2>)
                  ⋮              ⋮
                                          )
             (<SLOT 2>(<FACET 1>(<VALUE 1>)
                  ⋮              ⋮
                                          )
```
(a)

```
assert (<Frame Name>([<Slot 1>, <Facet 1>, <Value 1>],
                     [<Slot 2>, <Facet 1>, <Value 1>],
                          ⋮                         ])).
```
(b)

図-3 フレームの構造
(a)は LISP(b)は Prolog を用いていることを示している.

題の多くが，データベースの問合せに関するものであることからも理解できる. Prolog でデータベースが簡単に作れるということは，すでに，構造を持ったデータを扱えることを意味している. ところで，Prolog は LISP と違って，ホーン節でプログラムが書かれるために，プログラムが線状になってしまう. その結果，LISP のような構造的な読み方ができないということから，Prolog は平板であるということになったと考えられる.

LISP でのフレーム構造は，次のような形式で書かれる. フレームは属性リストである. それに対し，Prolog では Assertion でフレームを記述している. LISP では，フレーム構造の属性リストが基本構造であるのに対し，Prolog では Assertion 群が，基本構造となる. そして，フレーム処理は LISP では，ポインタのつけ変えであるのに対し，Prolog では Assert をし直すことになる. フレームに対する手続き付加は LISP では Demon (デモン) や Default (デフォルト) を用いる. Prolog では，Demon や Default と等値のプログラムを Prolog で作成することによって実現している. 例えば，フレームへの値を付加する put は，次のような Prolog プログラムとなる.

```
pput (Frame, Slot, Facet, Value) :-
    fact1 (Frame, Lst), Pro1 (Frame, Slot,
    Facet, Value, Lst),
    pro_attach (Frame, Slot), !.
Pro_attach (Frame, Slot) :-
    make_frame (Slot, Name),
    pset (Name, when-filled, value, [X]),
    Type= ..[X, Frame], call (Type) ; true.
```

このプログラムの例では，フレーム構造を Prolog で直接的に作成したものである. 述語Pro_attach はフレームへの手続き付加を示したものである. この手続きは，pget の中で示されているように，デモンの when_filled を持っている.

Prolog を用いてフレームシステムを作成することは以上のように，ごく自然に実現することができる. 我々の経験では，Prolog プログラムのコード数にして，100～200 行程度で，Winston, P. の著書のフレームシステムを作成している (文献，前出).

オブジェクト指向概念は，フレームシステムのより一般化と考えられる. フレームが静的な構造の，階層性や性質の遺伝性を実現しているのに対し，オブジェクト指向概念が，動的なプロセスとして階層性 (クラス概念) と遺伝性をプログラミングシステムに導入している. 例えば，上位一下位のクラスの結びつきは，フレームではスロットを用いているが，オブジェクト指向では，メッセージ (手続き名+引数) を使用している.

Prolog を用いてオブジェクト指向概念を作成するには，オブジェクト構造をシンタックスシュガとして実現する方法もある. 我々が開発した知識表現言語の鏡[15] (MIRROR : Multiple Image Representation foR Objects and Rules) では，オブジェクトをローダを通して作成していく方法をとっている.

鏡におけるオブジェクトのシンタックスは，図-4 に示すような形式を採用している. この形式のオブジェクトがローダにより，Prolog のプログラムに変換される. 例えば，次のようなクラス定義がロードされた時に，どのような形式に変換されるかを見てみよう.

```
defclass abc
  begin
  supers [object];
  metas class;
  cvars cx;
  ivars [x, 0], [y, prop, _];
  methods
    plus (X, Y) : ......;
    minus (X, Y) : function minus;
          ⋮
end
```

以上のクラス定義が鏡のローダにより，次のような，(1)のような内部データベースの部分 (record) と，(2)のようなデータベースの部分 (assert) に分割される.

```
defclass クラス名
  begin
    metas   メタクラス名[,プロパティ,…];
    supers  スーパクラスのリスト[,プロパティ,…];
    cvars   クラス変数の定義;
    ivars   インスタンス変数の定義;
    methods
      セレクタ(アーギュメント):プログラム;
      セレクタ(アーギュメント):function 述語名;
      (プロパティ,セレクタ(アーギュメント)):
        プログラム;
    before セレクタ(アーギュメント):プログラム;
    after  セレクタ(アーギュメント):プログラム;
end.
```

図-4 鏡におけるクラスのシンタックス

```
(abc, super ((nil, [object])))              )
(abc, meta ((nil, class)))                  |
(abc, cv ((cx, nil,_)))                     |
(abc, iv ((x, nil, 0)))                     |
(abc, iv ((y, prop,_)))                     }  (1)
(abc, method ((plus, nil, function          |
  abc)))                                    |
(abc, method ((minus, nil, function         |
  minus)))                                  )
            ⋮
abc ([plus|(X, Y)|nil], self, z) :-……       }  (2)
minus ([minus (X, Y)|nil], self, z):-……     }
```

(1), (2)の部分を整理すると, 次のような形式となる.

(オブジェクト名, super((プロパティ, 値)))
(オブジェクト名, meta((プロパティ, 値)))
(オブジェクト名, cv((変数, プロパティ, 値)))
(オブジェクト名, iv((変数, プロパティ, 値)))
(オブジェクト名, method((セレクタ名, プロパティ, 値)))　　(1)′

(1)のうち, 特に重要なのは(1)′である. ここでセレクタ名はクラス定義の中のセレクタのファンクタ名である.

(2)の部分に, すなわち method に対応する部分は, 次のようになる.

クラス名([メッセージ|プロパティ],
　self, Return) :-
　Body 部　　(2)′

ここで, 変数 self には, 実行時にメッセージを受取っているオブジェクト名が結合される. 変数 Return には, クラス定義の method で, 例えば, "X" というものがあれば, この X が変数 Return に結合される. もし, ない時には, 変数 self の値が結合される.

このように Prolog の形式に変換された(1)′, (2)′は, それぞれ, 次のような役割を持つ.

(1)′ 実際に実行される本体である.

(2)′ メッセージパターンがあるかどうかのチェックに使われる.

処理のプロセスをみるために, クラス a-class のインスタンス a を作成し, インスタンス a に msg というメッセージが送られたとしよう.

$ a_class ← new(a). インスタンス a を作成する.

a ← msg. インスタンス a に msg というメッセージを送る.

インスタンス a は当然, 自分の所属しているクラスを知っているので, そのクラス (a_class) の method (内部データベース(2)′) に, 自分が受け取ったメッセージパターンと同じメッセージパターンがあるかどうかチェックする. そして, もし, 存在していたときには, (1)′のような形式を作成し, その method を実行する. もし, メッセージパターンが存在しなかった時は, a_class のスーパクラスに対して同様のことを繰り返していく. このことを示したのが**図-5**である.

以上が Prolog を用いたオブジェクト指向言語の鏡の基本部分である. 実際のエキスパートシステムの設計には, 対象世界の記述をオブジェクトの概念により記述する必要がある. Rutgers 大学で開発された EXPERT 形式のように, 事実→事実のルール (ff ルール), 事実→仮説のルール (fh ルール), 仮説→仮説のルール (hh ルール) のような三つのルール集合で表現できるときは, **図-6** のようなクラス定義となる. このうち fh_rule を鏡で表現した例を, この図-6 に示す. ルールに3つのクラスが存在し, どのデータに対して, どのルールを適用していくかは各々のルール自身が知っているということになる. 従来のルールベースには, こうしたオブジェクト指向の考え方はなかったが, ルールそのものをオブジェクトとみなすと, 以上のような構造のエキスパートシステムを作成することができる.

以上が, Prolog を用いたオブジェクト指向のエキスパートシステムを作成するときの基本部である. このアプローチは, Prolog をオブジェクト指向言語設計のためのプログラム言語とみなしている点で, 本論の第2節と同様の方向である.

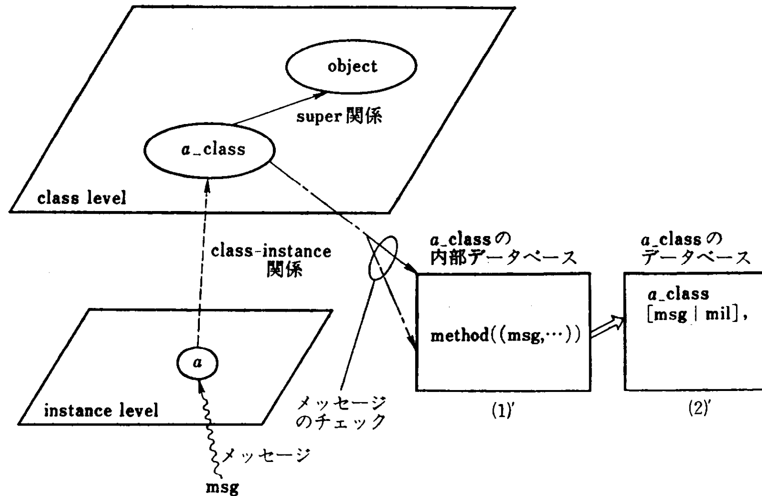
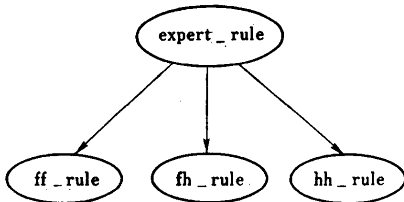


図-5 メッセージ送信の制御



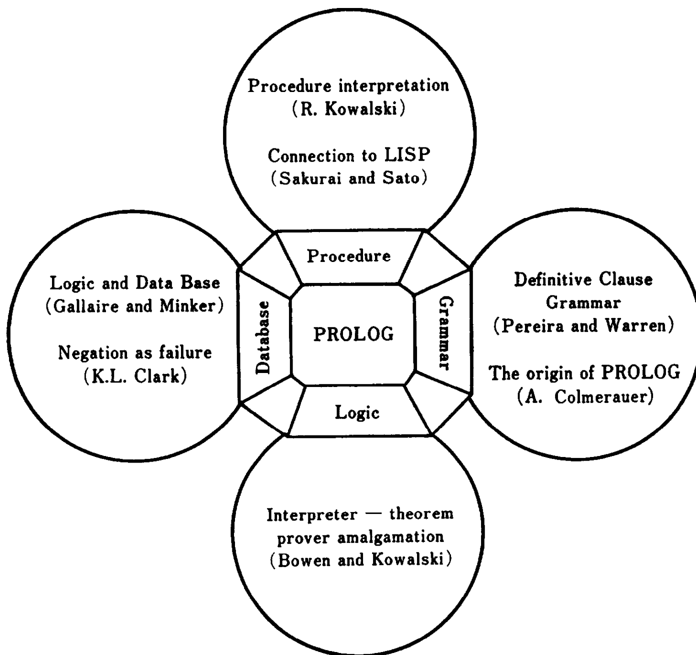

```
defclass fh_rule
  begin
    metas class: supers [expert_rule];
    cvars [current_fhrule, [ ]],
          [fhrule_set, [ ]],
          [success_fhrule, [ ]];
    ivars [fhrule, fh(_), _];
    methods
      after instance:
        addclassvalue ($fh_rule, fhrule_set, self);
      apply_fh:
        (@. (fhrule, fh(Num), Rules),
          pushclassvalue ($fh-rule, current_rule,
            Num), apply_fh (Rules, Num).
          popclassvalue ($fh_rule, current_rule,_),
            fail;
        true);
end.
```

図-6 EXPERT 形式のルール集合のクラスと fh_rule の鏡の定義

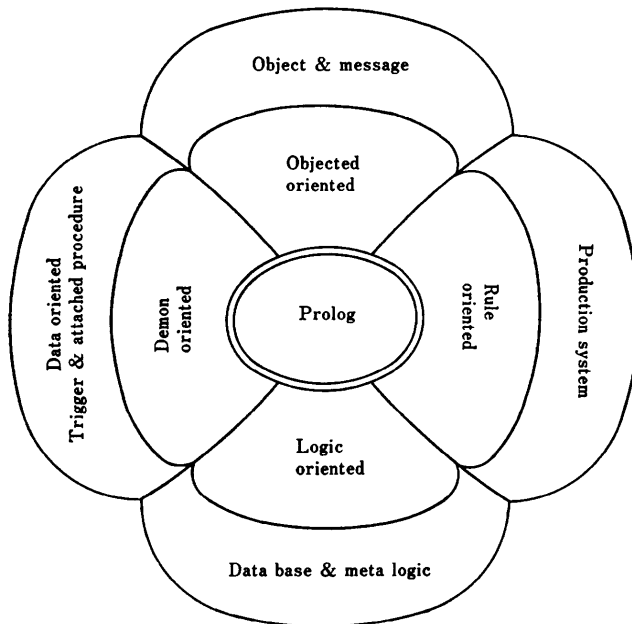
6. おわりに

本稿は，Prolog を用いたエキスパートシステムについて，ルールベースとオブジェクト指向の考え方で実現することを具体的に述べたものである．Prolog は LISP に比べてエキスパートシステムへの応用が少ない．また，どのように設計するかが不透明である．そうした疑問に対応することを目的としたのが本稿である．

Prolog は論理を基礎とする側面と，ユニフィケーションという強力なパターンマッチング機能を有する言語の側面を持っている．論理（推論システム）と考えると，Prolog はルールベースシステムそのものになってしまう．それに対して，プログラム言語と考えると，ルールベースシステムのインタプリタを作成することでエキスパートシステムを開発することができる．Prolog の持つ利点は，こうした記述力の柔軟性にある[16]．このことをさらに具体的にみたのが図-7(a)(b)である．図-7(a)は Prolog を文法記述 (Grammar)，手続き記述 (Procedure)，データベース記述 (Data base) および論理 (Logic) という多面性を実現するときの触媒 (catalyst) とみなすという考え方である[17]．図-7(b)は，ルール指向，オブジェクト指向，デモン指向，論理指向というプログラミング方式を映す鏡 (Mirror) としての働きをするのが Prolog であるという見方である[18]．これを基礎にした知識表現言語が第5節で説明した鏡である．さらに，オブジェクト指向概念を導入し，オペレーティングシステムの記述にも適用しようというのが新世代コンピュータ技術開発機構の ESP[19](Extended Self-contained Prolog) である．ESP では，ミクロで論理型言語，マク



(a) Prolog acts as cata lyst [後藤 1983]



(b) Prolog serves as reflecting Paradigms (溝口 1984)
図-7 Prolog の多様性

ロにオブジェクト指向言語，全体としてマクロ展開機能を持つという考え方である．ESPは PSI マシンのシステム記述言語で使われており，別の見方からすると，オペレーティングシステムというエキスパートシステムを実現していると言えよう．

このように，Prolog は記述力において多様な解釈と柔軟性を持ち，エキスパートシステムの設計に適している．ただし，現在の DEC-10 Prolog の問題点は，ユーザ領域が約 200 kW (1 word=36 bit) であり，大規模のルールベースシステムを作成できないことにある．その意味で，第5世代プロジェクトの前期の成果である PSI マシンは，エキスパートシステムの新しい展開にとって，きわめて重要な役割を持つと考えられる．Prolog の特に記述力に加えて，さらに強力な言語である ESP と PSI マシンは，従来のエキスパートシステムを吸収するだけでなく，実用規模の知識ベースの構築が容易になると考えられる．

その意味で，従来の Prolog でエキスパートシステムを開発し，その有効性と限界を明確にするための試行が現在，もっとも必要である．本稿がそうした方向に沿っていれば幸いである．

謝辞　本稿で述べたエキスパートシステムのインプリメンテーションは東京理科大学工学部溝口研究室の諸君によるところがきわめて大である．特に，ルールベースシステムの基本型は三輪和弘，大和田勇人君らに，また，鏡は本田栄司氏（現インテック(株)）による．また，新世代コンピュータ技術開発機構のワーキンググループ4の諸氏，および，同機構の第二研究室長古川康一氏との討論が本稿を書く上で有益であった．これらの諸氏に感謝したい．

参考文献

1) Bobrow, D. G. and Stefik, M.: The Loops Manual, Xerox Corp. (1983).
2) Weiss, S. M. and Kulikowski, C. A.: EXPERT: A System for Developing Consultation Models, Proc. Sixth IJCAI, pp. 942-947 (1979).
3) Santane-Toth, E. and Szeredi, P.: Prolog Applications in Hungary, in Clark & Tärnland (Eds.), Logic Programming, Academic Press (1982).
4) Warren, D., Pereira, F. and Pereira, L. M.: User's Guide to DEC System-10 Prolog, Dept. of Artificial Intelligence, Univ. of Edinburgh (1979).
5) 溝口文雄：知識工学，コンピュータソフトウェア，Vol. 1, No. 3, pp. 13-22 (1984).
6) 斎藤正男，溝口文雄：知的情報処理の設計，コロナ社 (1984, 第3版).
7) Winston, P. H. and Horn, B. K. P.: LISP, Addison-Wesley, Reading, MA (1981).
8) 溝口文雄：論理型言語による推論システムの展開，医療情報学，Vol. 3, No. 4, pp. 165-173 (1984).
9) Mizoguchi, F.: PROLOG Based Expert System, New Generation Computing, Vol. 1, No. 1, pp. 99-104, Springer-Verlag (1983).
10) Shapiro, E.: Methodology of Logic Programming, Proc. of Logic Programming (1983).
11) Clark, K. L. and McCabe, F. G.: Micro-PROLOG: Programming in Logic, Prentice-Hall (1984).
12) Hammond, P. and Sergot, M.: Apes: Augmented Prolog for Expert System, Reference Manual, Logic Based System Ltd. (1984).
13) 三輪和弘，溝口文雄：メタ述語による推論システム，第1回日本ソフトウェア学会論文集 (Dec. 1984).
14) Clark, K. L. and McCabe, F. G.: PROLOG: A Language for Implementing Expert System, Technical Report: DOC 80/21, Univ. of London (1980).
15) 溝口文雄，本田栄司，片山佳則：オブジェクト指向概念を導入した知識表現言語：鏡と姿の設計と応用，The Logic Programming Conference '84, pp. 1-12 (1984).
16) 渕　一博：述語論理型言語，情報処理，Vol. 22, No. 6, pp. 588-591 (1981).
17) 後藤滋樹：Prolog の図形的な動作表示法，情報処理学会記号処理研究会資料 (1984).
18) 溝口文雄：LISP から Prologへ―人工知能の新しい展開，ICOT ジャーナル，No. 4. pp. 4-8 (1984).
19) Chikayama, T.: ESP Reference Manual, ICOT TR-044, ICOT Research Center (1984).

(昭和59年10月7日受付)