ICOT Technical Memorandum: TM-0142

TM-0142

エキスパートシステムにおける知識情報処理技術

北上　始（富士通）

October, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

エキスパートシステムにおける知識情報処理技術

㈱富士通研究所　北上　始

1. はじめに

人間の知的精神活動の中でも，医師，電気技術者，機械技術者などの専門家が行っている問題解決活動を自動化するために，種々のエキスパートシステムが研究・開発されている。エキスパートシステムは，問題解決部と知識ベース部に大別されるといわれている。著者が所属していた（財）新世代コンピュータ技術開発機構（ICOT）では，図1に示すように，1990年代の第五世代コンピュータの実現を目ざし知識情報処理技術の研究を進めてきた。エキスパートシステムを構築するためには，この知識情報処理技術を明確にする必要があるが，知識工学が発展するにつれ，この知識情報処理技術をしだいに利用できるようになりつつある。知識工学は，知識表現，知識利用，知識獲得といった視点から研究されてきたが，著者は，これらの視点に対する研究アプローチとして，論理プログラミング言語 Prolog をインプリメンテーションの基本としてきた。

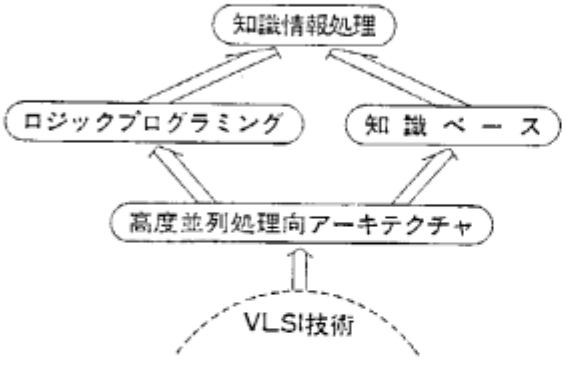


図1．第五世代コンピュータのアプローチ

このような事情から，本稿では，エキスパートシステムを構築するための基本技術として，主に，知識表現，知識利用，知識獲得といった視点から Prolog で整理した結果について述べる。また，エキスパートシステムがもつ知識ベースの知識は，物事の客観的な事実や規則を表現したオブジェクト知識と，その知識に関する知識（使い方，意図，制約などが含まれる）を表現したメタ知識とに大別されるとする。一般に，実世界の知識は，この両知識を組み合わせた表現になっている。すなわち，実世界の知識は，ホーン節の枠を越えているが，著者は，この実世界の知識のかなりの部分がホーン節形式のオブジェクト知識とホーン節形式のメタ知識に分解した知識構造で表現できると考えている[1]。したがって，ここで取り扱うオブジェクト知識とメタ知識は，どちらも論理型プログラミング言語 Prolog で記述可能なホーン節だけで表現されると制限し，プログラミング言語のシンタックスは，DEC－10 Prologを前提としている[2]。

2. エキスパートシステムのアーキテクチャ

エキスパートシステムの究極の目標は，専門家の問題解決活動を自動化することであるが，一般に，人間の知的問題解決活動は，野外科学，実験科学，書斎科学の3つ仕事の連係として見ることができる[3]。

野外科学は，ありのままの自然の姿をなんらかの対策を立てて探究する科学であり，仮説を発想的に発見する方法と関係している。実験科学は，ありのままの自然の中から操作を加えて人工的な自然をつくり，それを対象にした探究科学であり，仮説を帰納的に検証または精密化するところに重要な性格がある。この2つの科学とは異って，書斎科学は，自然そのものにたずさわる科学ではなく，過去の情報，即ち，一度だれか先人の頭脳フィルターをとおして，体系づけられた形の情報を対象にした科学である。これらの3つの科学の

連絡を図２に示す[10,11]。これは，人間の知的問題解決活動と見ることができる。即ち，問題解決の仕事は，A→B→C→D→E→F→G→Hのプロセスとして示すことができる。

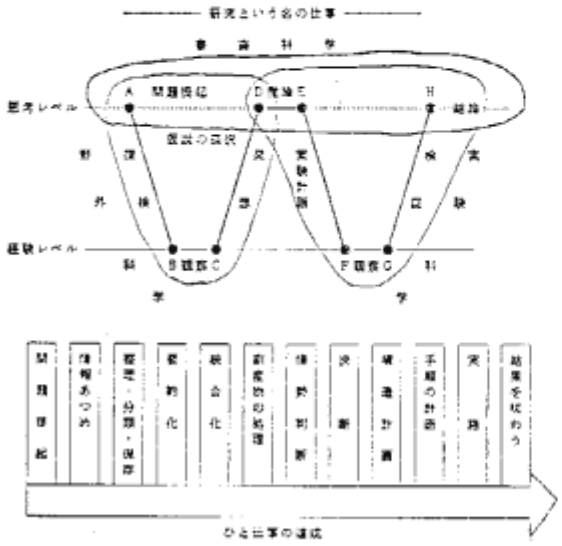


図２．人間の知的問題活動

このＷ型の図２は，問題が難かしくなればなるほど，鮮明になるが，問題が簡単になるにつれ，ＡとＤ，ＥとＨなどが近接して行き，Ｗ型がだんだん崩壊して（Ｉ型に近くなって）行く。ＤとＥだけで，問題解決を行うエキスパートシステムは演繹推論機能だけで問題解決を行うシステムであるといえる。

このように，図２で示した問題解決活動を完全に自動化することは，むずかしいが，その自動化の一例を示すことにしよう。

エキスパートの問題解決活動を自動化するためには，(1)野外科学，実験科学，書斎科学という問題解決活動の各段階で，エキスパートに問題解決のために使用する専門家知識を知識ベースに登録させ，(2)高度な推論エンジンや説明機構などによる問題解決能力をシステムにもたせなければならない。高度な推論エンジンとは，図２のＣ→Ｄ，Ｄ→Ｅ，Ｇ→Ｈの各プロセスで，おのおの必要な発想推論エンジン，演繹推論エンジン，帰納推論エンジンのことである。

これらを，エキスパートシステムのアーキテクチャとしてまとめると，図３に示すようになる。エキスパートにより登録された知識ベースの知識は，ユーザが行う種々の問題解決（診断，設計，計画など）を支援するために利用される。問題解決は，推論エンジンを中核として行なわれ，その結果は，一時的な知識ベースに蓄積される。推論エンジンの動作中，ユーザが，入力したデータを推論のために利用させたいときは，この一時的な知識ベースにそのデータを蓄積しておくことによって，それが可能になる。これにより，エキスパートシステムのユーザは，エキスパートの問題解決活動に匹敵する能力をもつことができる。

以上は，エキスパートシステムのアーキテクチャに対する大枠であるが，議論をさらに進めて，エキスパートシステムの理想的な設計目標を列挙すると，(1)拡張性に富んでいる，(2)効率が良い，(3)使いやすく習得しやすい，(4)大規模性に富んでいる，(5)多様問題解決が可能などを挙げることができる。これらの設計目標をすべて満足するようなエキスパートシステムを Prolog で作成するためには，多くの問題を解決しなければならないが後述するメタ推論方式を上手に使用したメタプログラミングを行うことにより，多くの問題が解決されることも確かである[11,13,19,39]。

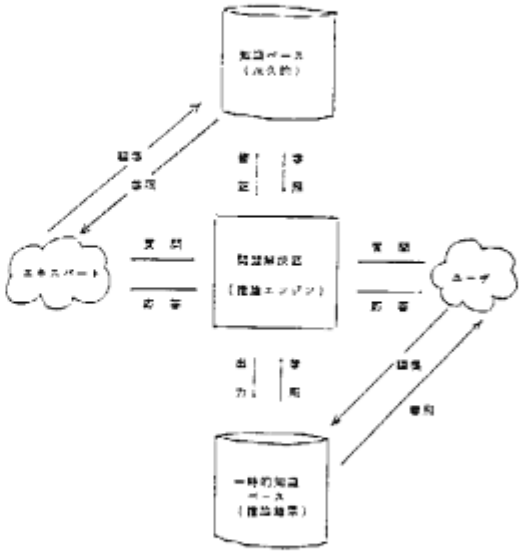


図３．エキスパートシステムのアーキテクチャ

3．　知識表現

ここでは，知識ベースに格納される知識の表現形式について述べる。既に述べたように，知識ベースの知識は，オブジェクト知識とメタ知識に分類される。以下，これらの知識の詳細を説明していく事にしよう[5-11]。

3．1　オブジェクト知識

オブジェクト知識は，概念の階層関係が明確に整理された構造化知識と，構造化するのが難しいために手続き的表現をとったプログラム化知識とに大別される。本章では，この二種類の知識について述べることにする。ここで，ホーン節を"p(X₁，X₂，…，Xₙ) :- body"とするとき，"p"のことを便宜上，概念と呼ぶことにする。

（1）　構造化知識

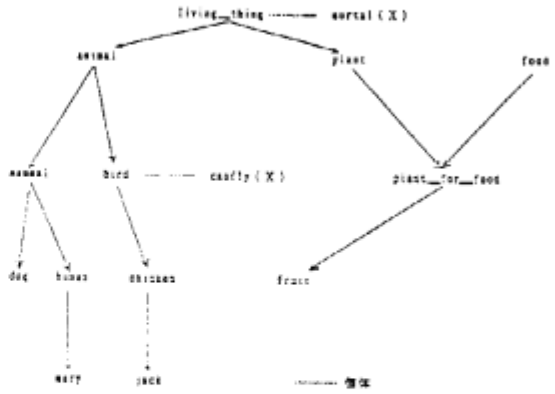
構造化知識は，自然言語処理の分野で知られる概念階層関係をPrologの事実として表現した知識である。概念そのものを定義域として，その関係を表現していることから，この種の事実は，二階述語論理の形式の知識を表現していることになる。これには，図4，5の例で示されるように，三種類の表現が知られている。一番目には，ある概念の特性（性質）を表現するproperty関係，二番目には，概念間の上位・下位関係を表現するis__a関係，三番目には，概念間の全体・部分関係を表現するpart__of関係があり，次のように表現できる[7-11]。

property(概念，特性を評価する述語)．　　(3－1)

is__a(下位概念，上位概念)．　　(3－2)

part__of(全体概念，部分概念)．(3－3)

property関係は，ある概念がもつ特性がどのようなルールによって評価されるかを示す述語である。is__a関係は，概念間の上下関係を示す最小単位であり，上位概念の特性が，下位概念に継承されるという性質をもっている。part__of関係は



図－4　is__a，property関係の構造
矢印　→　は，is__a関係を示し，
矢印　⋯>　は，property関係を示す。

```
/* Structuralized Knowledge */
is_a(animal,living_thing).
is_a(plant,living_thing).
property(living_thing,mortal(X)).

is_a(mammal,animal).
is_a(bird,animal).
property(bird,canfly(X)).
is_a(fruit,plant_for_food).
is_a(plant_for_food,plant).
is_a(plant_for_food,food).

is_a(dog,mammal).
is_a(human,mammal).
is_a(chicken,bird).

part_of(mammal,face).
part_of(mammal,body).
part_of(mammal,arms_2).
part_of(bird,face).
part_of(bird,body).
part_of(bird,wings_2).

is_a(mary,human).
is_a(john,human).
is_a(jack,chicken).
```

図5．構造化知識の知識表現例

概念間の全体・部分関係を示す最小単位である。ここでは，部分概念の諸特性が全体概念に継承されるという性質をもってると仮定する。

継承関係の例外処理は，他の機会で述べるとして，これらの継承関係を調べるためには，is__a関係をもとにして，連続的に何段もの上位・下位の概念をたどったり，part__of関係をもとにして連続的に何段もの全体・部分概念をたどったりする機能が必要である[7]。

(2)　プログラム化知識

前節の構造化知識は，物事を三種の関係（property, is__a, part_of）で表現しようとしていたが，このような関係だけで表現できない知識は，数多く存在する。プログラム化知識は，それを Prolog のプログラムとして表現した知識である。構造化知識とプログラム化知識の関係は相補的であるので，上手に概念の構造化が達成されるほど，構造化知識をプログラム化知識に利用するのが容易になるので，プログラム化知識を簡潔に表現できる。図6に，プログラム化知識の表現例をしめす。ここでは，"生物は，いつかは死ぬ（mortal(X)）"，"大部分の鳥は，空を飛べる（canfly(X)）"，"jackは，飢えている（hungry(jack)）"，"jackは，鳥の一例である（super__concept(jack, bird)"などの知識が示されている[7]。

```
/* Programmed Knowledge */
mortal(X):-super_concept(X,living_thing).
not_canfly(jack).
canfly(X):-super_concept(X,bird),
     \+not_canfly(X).

food(chickweed).
hungry(jack).

super_concept(jack,bird).
super_concept(mary,human).
super_concept(jenn,human).
```

図6．プログラム化知識の知識表現例

3.2　メタ知識[10,17,18,24]

メタ知識とは，"知識に関する知識"のことであり，メタ知識には，(1) オブジェクト知識を成長させるときに，誤った方向に成長することを防止するための制約型知識，(2)問題解決のオブジェクト知識を効率良く使用するための戦略型知識，(3)オブジェクト知識及びメタ知識の形式的な枠組みを示すための辞書型知識，(4)知識の集合体に対する問合わせ，更新操作などを行うためのメソッド型知識などがある。知識の集合体の論理構造については，図7に示す通りである。図の矢印（→）は，メタとオブジェクトの関係を表す記号であり

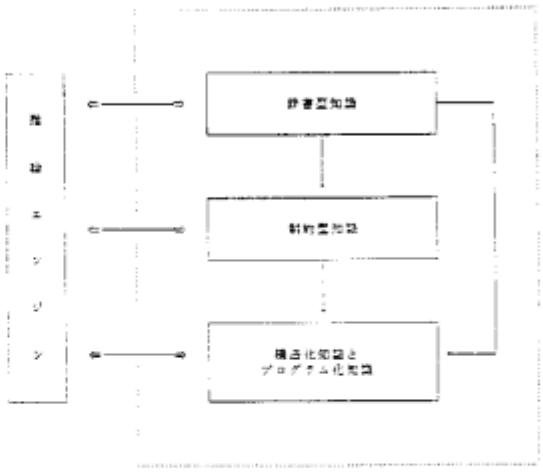


図7．知識の集合体（フレーム）の論理構造

"A → B"は，オブジェクト知識（あるいはメタ知識）Bのメタ知識がAであることを示している。したがって，プログラム化知識と構造化知識をオブジェクト知識とすると，その他の知識は，メタ知識になる。

(1)　制約型知識[13,17,24]

制約型知識には，次の種類がある。一つは，知識ベースの更新アクセスなどのオペレーションに対し，動的に整合性をとるために，種々の知識の追加又は削除を行う機能をもつ trigger型のメタ知識である。他の一つは，知識ベースの更新アクセスとしてのオペレーションに対する無矛盾性を監視する inconsistency型のメタ知識である。このメタ知識は，オブジェクト知識が矛盾する条件を記述しているので，否定型ルートと見なすことができる。両メタ知識ともに，後述するトランザクション処理の最後に起動する遅延型（delayed-type）のメタ知識と，トランザクション処理とは独立であって更新などの知識アクセスがあるごとに起動される即時型（immediate-type）のメタ知識とに分類される。以上から，これらのメタ知識は，次のように表現される。

trigger（起動タイプ，オペレーション）:-
　起動条件，！，複数の更新オペレーション，

（3－4）

inconsistent(起動タイプ, オペレーション):-
起動条件, !, 矛盾判定条件. （3－5）

同知識とも，起動タイプとして，immediate, delayed のタイプをもつ。同知識中に現れている記号 " ! " は，カットシンボルと呼ばれるメタ述語であり，ここでは，" A , ! , B " を " A ならばBが成立する " と解釈している。図8に，制約型知識の知識表現例を示しておく。図中には，五つの制約型知識があるが，その中から二つを取り出して，説明してみよう。第1番目のメタ知識は trigger 型のメタ知識であり，知識の集合体としてのフレーム frameに，" Agent が Information を Receiver に与える " という知識 (give(Agent, Receiver, Information)) を追加する (insert) ときに，Information が食べ物 (food) であり，Receiverが空腹 (hungry) であれば，その frameに " Receiverは，Information を食べる " という知識 (eat(Receiver, Information)) を連続的に追加する (insert) 処理を行う。次に，三番目のメタ知識を説明してみよう。この知識は，inconsistency 型のメタ知識であり，フレーム frameに " X の上位階層が Y である " という知識 (is__a(X , Y)) を追加した (insert) 後は，必ずその " Y " が " living __thing " につながっていなければ矛盾すると主張している。メタ知識の評価方式およびこれらの例を使用した実行例については，文献17) を参照されたい。

```
/* Constraints */
trigger(immediate,insert(frame,give(Agent,Receiver,Information))):-
   food(Information),hungry(Receiver),!,
   insert(frame,eat(Receiver,Information)).
trigger(immediate,insert(frame,eat(Agent,Information))):-
   \+ill(Agent),food(Information),!,insert(frame,cheerful(Agent)).
inconsistent(delayed,insert(frame,is_a(X,Y))):-
   \+super_concept(Y,living_thing).
inconsistent(delayed,X):-
   member(X,[insert(frame,_),delete(frame,_),update(frame,_)]),!,
   canfly(Y),not_canfly(Y).
inconsistent(immediate,X):-
   member(X,[insert(frame,_),delete(frame,_),update(frame,_)]),!,
   hungry(Y),full(Y).
```

図8．制約型知識の知識表現例

4. 知識利用

知識ベースの知識を使って，問題解決を行うためには，種々のタイプの推論エンジン及びそれらのエンジンを組み込んだ応用プログラムを作成しなければならない。本章では，知識の利用に際し最も基本となるメタ推論機構及び種々の推論エンジンについて述べる。

4.1 メタ推論[11,17,18]

エキスパートシステムの設計目標は，既に述べたように，いくつか挙げることができるが，それらの中でも，高い効率，拡張性，多種問題解決性などの点に着目すると，Prologの処理系自身がもつ単純な推論機能だけでは，対処しきれない。なぜなら，高い効率を実現するためには，推論の効率的な制御が必要であり，拡張性をもたせるためには，知識の集合体を利用目的に応じた単位で分割し，その分割単位ごとの推論を実現する必要があるからである。さらに，多種問題解決性を実現するためには，推論過程において種々の証明木が必要とされるような場合がある。以上から，本章では，このような種々の状況に対処するために必要不可欠なメタ推論について述べる。メタ推論は，" 推論に関する推論 " であり，demo述語と呼ばれるメタ述語により達成できる。demo述語の一般形式は，" demo(Frame, Goals, Control, Result) " である（付録を参照）[13,20]。この demo 述語は，知識の集合体としてのフレーム Frame の中で，与えられた制御 Controlに従って，与えられたゴール列 Goalsを証明し，その証明プロセスから必要な情報 Result （証明木などがある）を抽出する。この4引き数の demo 述語は，エキスパートシステムをインプリメントするために利用できる以外に，メタ知識を表現するためのツールとしても利用できる。以下，本稿の説明では，問題ごとに重要な機能が浮き出るようにするために，この一般形式の demo 述語を持ち出すことを避け，

問題ごとに特殊化した demo 述語で解説を試みることにする。図９に demo 述語の構造を知る上で最も簡単なプログラム例を示しておく。図９の demo の述語は " demo (Frame,Goals) " 形式の demo 述語であり，フレーム Frame単位で分割された知識の集合はごとの推論ができるようになっている。ただし，フレーム内のホーン節形式の知識は，すべてカットシンボルと呼ばれるメタ述語を含まず，ホーン節表現として，論理和表現を含まないと仮定する。図３の第一番目のルールは，ゴールを証明していったときの停止条件である。第二番目のルールは，論理積の展開証明を行っている。第三番目のルールの後半分は，
" P " と単一化可能な帰結部をもつホーン節をフレーム Frameの中からさがし，" Q " の証明を行っている。このホーン節をさがす部分は，clause__of述語で実現されている。図中の " ; " は，論理和記号であり，" A → B ; C " は，
" if A then B else C " と同じ意味である。またこの単一化可能なホーン節が条件部をもたないとき，" Q " には，真 " true " が返され，第三番目の最後の述語 demo(Frame, true) の実行結果が第一番目のルールにより真になる。図９のeval述語は，" P " がシステム組み込み述語
(system(P) で示されている)，または，フレームへの明示アクセス述語 (method__type(P) で示されている) のときにPを実行評価する述語である。図中の第四番目のルールは，clause__of述語を表現するルールであるが，この中の " X " には，メタ述語 " =.. " により述語 " Frame(Clause) " が割り当てられる。

```
demo(Frame,true):-!.
demo(Frame,(P,Q)):-
     demo(Frame,P),demo(Frame,Q).
demo(Frame,P):-
     (system(P);operation type(P))->
     eval(Frame,P);clause_of(Frame,(P:-Q)),
     demo(Frame,Q).

clause_of(Frame,(P:-Q)):-
     X=..[Frame,Clause],X,
     (Clause=P->Q=true;Clause=(P:-Q)).
```

図９．２引数の demo 述語

4.2　演繹推論エンジン[11,12,16,17]

demo述語は，一般に，後向き推論や不確実性を伴う推論，時間を伴った推論などの推論エンジンを実現するための中核になっているが，これらはすべてゴール形式（G_1，G_2，……，G_n）の証明問題を解決するためのエンジンと見ることができる。ゴール中に存在する変数は，すべて，存在限定されている。しかし，知識同化の問題や冗長性の判定問題などは，ホーン節形式（Head:-G_1，G_2，……G_n）の証明問題を解決しなければならない[12,13]。ここでは，この問題を解く推論エンジンを演繹推論エンジンと呼ぶ。この推論エンジンを実現する方式には，前提条件の違いにより２つの方式が考えられる。それらは，ホーン節に存在する変数Xの値が，存在知識だけに左右されると仮定した場合と，そうでない場合に区別されてる。ここでは，後者の一般的な場合を deduce 述語と呼び，その実現の方法について述べてみたい。

図10に，deduce述語の実現プログラムを示しておく。

```
deduce( Frame, Clause ) :-
     select __variable( Clause, Variable__List),
     skolemize( Variable__List),
     ( Clause= ( Head :-Body ) -> demo( Frame, Head, Body ) ;
     demo( Frame, Clause, [] ) ),
```

図１０．deduce述語の実現例

スコーレム化述語と，demo述語を用いて，" Head :-Body " が証明できるかどうかを判定する。

図10のプログラムを実現するために，論理式の変形操作により，「Gから " Head :-Body " を証明する」問題を，「Gおよび " Body" " から " Head" " を証明する」問題に帰着させた。ここで，＊印は，証明すべきホーン節 " Head :-Body " が持つ変数に，知識ベース内に二度と現れることがない変数（アトム）を割り付けた（スコーレム化した）状態を指す。証明すべきホーン節 " Head :- Body " が持つ変数は，いかなる値をとって

もよいことを示している（全称限定付きである）のであるから，このスコーレム化の方法は妥当な方法である。"Head" "の証明可能性は demo 述語で判定できる。

図10の deduce 述語を実現するのに利用されている各種の述語についての意味を証明してみよう。"select__variables "述語は，証明すべきホーン節 "Clause" が持つ変数（すべて全称限定されている）をすべて，抽出する述語である。この述語は，その抽出結果を変数リスト "Variable__list" として返す。"skolemize "述語は，与えられた変数リスト "Variable__list" に基づいて，"Clause" をスコーレム化する述語である。証明すべきホーン節 "Clause" が条件部 "Body" を持つ場合は，その "Body" を知識ベースに追加したと仮定した状態の下で，帰結部 "Head" の証明可能性を判定する。この判定は，demo述語で実施でき，この demo 述語は第三引数に，知識ベースに "Body" "を追加するという仮想状態を作る機能を持っている。この機能は，推論を制御する機能と解釈できる。第三引数の機能を使用しないときは，〔〕を指定することにする。ここで，図10に "（A→B；C）という形の処理があるが，これは，DEC－10 Prolog でも使える機能であり，"if A then B else C" と同じ機能を持つ。各組み込み述語の詳細は，付録を参照されたい。

4.3 その他の推論エンジン

演繹推論エンジンの他に帰納推論エンジン，発想推論エンジンなどが考えられるが，ここでは，帰納推論エンジンについて詳述することにする。発想推論エンジンについては，類推などで代表される推論エンジンがあるが，これについては，他の文献を参照されたい。

帰納推論エンジンの代表的なものには，モデル推論アルゴリズムがある[87,88]。これは，哲学者 K. Popper の「推測と反証」の考え方を使って，Shapiro が考案したアルゴリズムである。以下，それについて述べて行くことにする。

図11に，モデル推論アルゴリズムの抽象的なイメージを示す。その詳細は，順次，示していく。

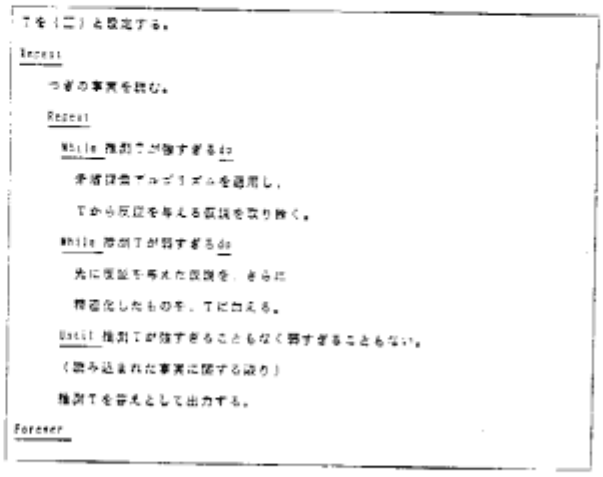


図11．モデル推論アルゴリズムの概略

図11の "□" は，矛盾としての空節を意味する記号であり，本アルゴリズムは，どのようなルールも，この空節 "□" を出発点として仮説を精密化していくことによりモデルを作成することができるとしている。この精密化は，精密化オペレータ（Refinement Operator）を用いて達成される。選択された仮説は，与えられた事実の少なくとも一つを満足していなければならない。

推測Tは，仮説の集合であり，推測Tからユーザが与えた負の事実（あり得ない例題）を証明できたとき，「推測Tが強すぎる」という。これは，思い込みの強い性格の人が持つ知識と似ている。そのような人には，正確な知識を持たせてやるために，現在その人が持っている知識をさらに精密化してやる必要がある。

これとは反対に，測定Tからユーザが与えた正の事実（満足しなければならない例題）を証明できないとき，「推測が弱すぎる」という。これは，疑い深い性格の人が持つ知識と似ており，そのような人には，（その人がまだ疑わしいと思っている）正しい知識を積極的に教えてやる必要がある。

そして，本アルゴリズムは，この両者の証明関係のどちらも満足しなくなったとき（推測が強すぎることもなく弱すぎることもない），その時点までに読み込まれたすべての事実（正および負の事実）に対して，適切な推測T（モデル）を推論したことになる。図12は，このアルゴリズムが備えているモデル推論過程のイメージ図である。図12は，正（真）および負（偽）の事実からモデルを推論する課程で，仮説がどのように変化していくかを示している。　図中の平面は，その仮説が変化していく段階で各々の状態を示したものであり，正および負の事実は，"●"および"×"で示されている。H₁，H₂，…　および，I₁，I₂，…　は，ともに仮説であり，H₀ and I₀（＝T₁）は，最終的な目標として得られる推測のモデルである。各平面の中に記された円（細線）の大きさは，仮説がもつ概念の大きさを示す。正の事実が与えられると，それを満繕するもっと大きな円が選択されるが，負の事実を与えていくと，徐々にその円の周囲が削られ（負の事実を満繕しない仮説を見つける），目標のモデルに近づいていく。ここでは，HとIの二つの仮説を示しているが，一般には，任意個でよく，T₁＝　H₀ and J₀ and …　のようになる。

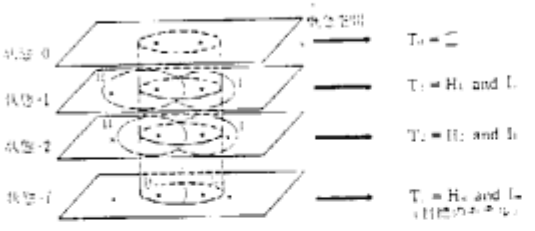


図１２．モデル推論過程における仮説の反駁と精密化。

以下では，モデル推論アルゴリズムの詳細について説明する。図13に，Shapiro が定式化したモデル推論アルゴリズムの詳細を示す。この図中でShapiro が定式化したモデル推論アルゴリズムの詳細を示す。この図中で，S_{true}，S_{true}はそれぞれ負，正の事実を蓄積する集合であり，

```
S_true = {□}, S_true* { }.
L = {□} とし，□に "false" 印を付ける。
Repeat
  次の観測データF₀ = <OBS₀, TF>を読み，OBS₀をS_trueに追加する。
  Repeat
    While (S_trueに属するOBS₀がL₀から演繹可能)，
      矛盾追跡アルゴリズムにより，反駁仮説をみつけ，その仮説に "false" 印を
      付ける。
    While (S_trueに属するOBS₀がL₀から演繹不可能)，
      "false" 印に付けてある仮説に，精密化オペレータを適用し，OBS₀がL_q ("L₀ and H_i") から演繹可能となるように新仮説H_iを見つける。そして，その新仮説をL₀に付け加える。
    Until (上記，Whileループのどちらも実現しない)，
      L₀の中で "false" 印を付けていない仮説を出力する。
  Forever
```

図13　モデル推論アルゴリズムの詳細

S_{true}のなかに，事実として，矛盾としての空節"□"が蓄積されている。L_0，L_1，…，L_q，L_{q+1}，……は，推測の状態変化を示すために容易された記号である。初期状態としては，推測L_0の中に矛盾としての空節"□"が蓄積されているが，本アルゴリズムの先頭に，この空節は本アルゴリズムが求めるべきモデルではないと宣言されている。そのために，"□"に false（偽）という印を付けてある。

次に，F_0＝（OBS₀，TF）は，観測事実（ユーザが与えた正または負の事実）であり，観測文OBS_0は，ファクトである。TFには，真偽値として true または falseを与える。このアルゴリズムによる処理は，外部から観測事実を与えているかぎり継続される。

本アルゴリズムのなかで，L_0，やL_qは推測であり，H_iは，仮説である。推測のなかで，false印が付けられていない仮説の集合が解である。図13で示されている演繹（証明）問題については，demo述語で実現することができる。この問題ではそのゴールはOBSであり，ファクトとも呼ばれているものである。また，図中の矛盾追跡アルゴリズム（ contradiction backtracing ）も，demo述語を少し変形することで実現できる。すなわち，負のファクトを推測L_qから演繹証明できるときその原因となる知識ホーン節（知識）集合が，反駁仮説になるので，demo述語の証明プロセスでホーン節を抽出する機構を付ければよい。　次に，図13のアルゴリズム中で用いている精密化オペレ

ータについて簡単に説明する。このオペレータは反駁仮説としてpをとり，その仮説に対する新仮説の候補としてqを作成するのに利用される。そこでは，図14に示すオペレータ（書き換え規則）のうちいずれかがpに適用される。

図14の精密化オペレータを順に適用していくと図15に示すような仮説の精密化グラフを作成することができる。図のH_2～H_7は，精密化オペレータにより作成された仮説であり，太い矢印は，精密化の前後関係を示す関係である。たとえば，仮説H_2を精密化すると，H_5～H_7の仮説を作成できる。図中では，H_4がゴールとして発見される仮説であることを示している。

本アルゴリズムを効率よく実行させる種々の戦略（strategy）を挙げることができるが，現在，よく知られているものをまとめると，次のようになる[2,15]。

(1) 仮説に与えられているデータ・タイプを利用し，精密化オペレータ（図14）の二番目のルールにおける単一化の組み合わせ数を低減する。これにより，同じデータ・タイプどうしの変数を単一化するだけで，新仮説を作成することができる。異なるデータ・タイプ間の単一化により生ずる新仮説は，明らかに無意味である。

(2) 仮説を作成するときに組み込み述語をできるだけ多くユーザに定義させ，システムにその述語を積極的に組み込んだ新仮説を作らせる。適切な組み込み述語が事前に定義されていないとすると，帰納的にあらゆる可能性を尽くした新仮説を作成することになるので，ゴールにたどりつくまでに多くの新仮説を作成しなければならない。

(3) 仮説が持つ引数の入出力仕様を利用し，出力引数が，入力引数に対して制御可能（可制御）な構造だけを持つ仮説を選択するようにする。

(4) 仮説（ホーン節）条件部のなかに同じふるまいをするゴールを存在させないようにする。これは，むやみに長い条件部を持つ仮説を作成することを禁止し，新仮説を見つけるための探索空間を狭めることに役立つ。

(1) $p::=\square$ならば，$q::=a(X_1,X_2,X_3,\cdots,X_n)$とする。ここで"a"は，帰納的に一般化される述語名（概念名）であり，n子の引数を持つ。

(2) $p(X_1,X_2,X_3,\cdots,X_n)$に対して，$X_i$と$X_j$を単一化し，その結果をqとする。ここで，$X_i$と$X_j$は，異なる変数である。

(3) $p(X_1,X_2,X_3,\cdots,X_n)$に対して，$X_i$に$t(Y_1,Y_2,Y_3,\cdots,Y_m)$を割り付け，その結果をqとする。ここで，tは，m引数の関数名である。

(4) $p::=(Head:-Goals)$に対して，$q::=(Head:-Goals,G_n)$とする。ここで，G_nは，辞書や知識などに登録されているユーザ定義述語または算術関数として定義される述語である。

図14．精密化オペレータ

これは，図13のモデル推論アルゴリズム中で使用されている。

(5) ループを許さない仮説だけに着目する。

(6) 反駁仮説を有効利用し，同じ誤りは二度と繰り返さないという原則にのっとり新仮説作成のための再計算を防止する。

よく知られているPrologプログラムmember(X Y)を例に挙げ，高速化戦略と精密化グラフの間の関係をみてみよう。member(X，Y)は次のようなプログラムである。ここでは，memberの第1引数を出力引数（メンバ・リスト中のあるメンバが返される）とし，第2引数を入力引数（メンバ・リストを記述する）とする。

このプログラムで，1行目の知識を精密化グラフ上で捜す例を図16に示す。図16のカッコ内に記されている数字は，図14で示した精密化オペレータのルール番号を指す。図16の＊印は，高速化戦略の(1)を適用することにより，その印が付けられた仮説の生成を省略できることを示している。さらに，高速化戦略の(3)，(5)と精密化オペレータの(4)をmember(X，[Y|Z])に適用すると，memberプログラムの第2行めの知識をただちに作成することができる。

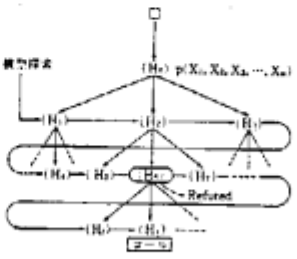


図15．精密化グラフ

精密化オペレータを順に適用していったときに，作成されるグラフである。ここでは，"H_4"が反駁された(Refuted)仮説とする。

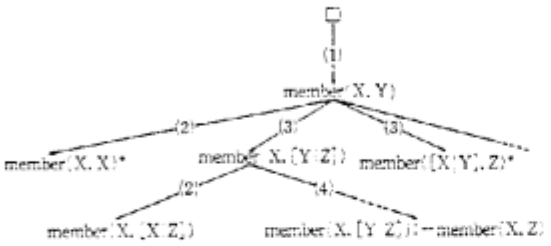


図16．member（X，Y）の精密化グラフ。

＊印は最適化戦略（1）（3）（4）により省略できる仮説を示している．ただし，精密化グラフのノード間に変数名としての依存関係はない．

5. 説明機能

ユーザが推論結果を理解し，納得するために，システムは，推論過程に対する説明を必要に応じて行なう必要がある。エキスパートシステムで重要であるといわれている説明機能としては，Why，How，Whynot という型の質問に答えられる機能が挙げれらる。ユーザが推論結果にWhy 型の質問をしたときは，その推論結果を証明するのに使われた最も新しいルールを表示し，How 型の質問に対しては，その推論結果の導出過程を表示すれば良いことがわかる。また，Whynot型の質問に対しては，あるゴールの証明が失敗したとき，なぜ失敗したかを説明すれば良い。

以上の３つの質問に答えるためには，メタ推論機を上手に利用すれば良いことがわかる。

その１例として，Why 型，How 型の質問に答えるためのプログラム例を図-17 に示す[35]。

```
demo(FR,true,[],[]):-!.
demo(FR,(P,Q),Why,How):-
     demo(FR,P,Why1,How1),
     demo(FR,Q,Why2,How2),
     append(Why2,Why1,Why),
     append(How1,How2,How).
demo(FR,P,Why,How):-
     clause_of(FR,P,Body),
     demo(FR,Body,Why1,How1),
     append(Why1,[(P:-Body)],Why),
     append([(P:-Body)],How1,How).

clause_of(FR,P,Body):-
     X=..[FR,Clause],X,
     (Clause=P->Body=true;Clause=(P:-Body)).

append([],X,X).
append([X|Y],Z,[X|W]):-append(Y,Z,W).
```

図17．Why 型，How 型 質問に答えるための demo 述語。

6. 知識獲得[12,15,16,17,20,22,24]

知識獲得時に，エキスパートから獲得される知識には，構造化知識やプログラム化知識といったオブジェクト知識の他に，制約型知識のようなメタ知識がある。このときの知識獲得能力は，認知心理学で知られるピアジェの知性発達モデル を参考に実現できる。このモデルは，図18の最上位階層であり，その階層では，知識の同化（Assimilation），調節（Accommodation）をはじめとする種々の機能をエキスパートに提供することができる。知識同化は，知識ベースのフレームに暖昧な新知識（assumption- typeの新知識）を無矛盾に追加することを意味する。知識調節は，知識ベースのフレームに正しい新知識（preaise-typeの新知識）を無矛盾に追加するために，そのフレーム内の知識を修正することを意味する。知識の同化・調節の本質的な問題は，知識ベースの無矛盾性を維持し，選択的に知識ベースの非冗長性を維持することである。さらに，知識ベースが持っている知識を，エキスパート（またはユーザ）が容易に調べられるようにするために，知的な質問応答（Intelligent Question-Answering）に基づく知識の検証（Verification）能力をもつ。 さて，知識獲得機能を支える基礎としては，図18に示すように，最初にメタ推論メカニズムを挙げることができる。このメカニズムには，推論エンジンによる推論過程を自由自在にあやつるメカニズムで

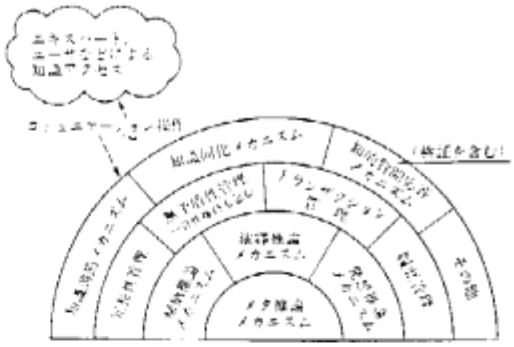


図18．知識獲得機能のダイヤグラム

もある。図中のメタ推論をベースとして，その次に基本的なメカニズムは，演繹，帰納，発想という推論メカニズムがある。メタ推論メカニズムは，ある制御条件下でゴール列を証明する役割を演じるのに対し，演繹推論メカニズムは，与えられたホーン節が証明可能か否かを解決する役割を演ずる。

また，帰納推論メカニズムは，ファクトからルールを合成する役割を果たすメカニズムであり，このメカニズムでは，Shapiro が研究したモデル推論アルゴリズムを利用することができる。さらに，発想推論メカニズムでは，類推などの高度な推論を行うメカニズムが必要であると考えている[9]。さらにその上には，冗長性管理，無矛盾性管理，トランザクション管理，履歴管理などがある。冗長性管理は，知識ベース内の知識の冗長性を除去する問題を考えることであり，無矛盾性管理は，制約型知識で，知識ベースを無矛盾にする管理である。トランザクション管理については，7章を参照されたい。履歴管理は，時系列の知識に対する時間的な推論を中心に扱う機能と考えているが，これについては，機会を改めて，述べることにしたい。

次に，エキスパートの手によって，いかにしてこれらの機能を利用するのかについて説明してみよう。

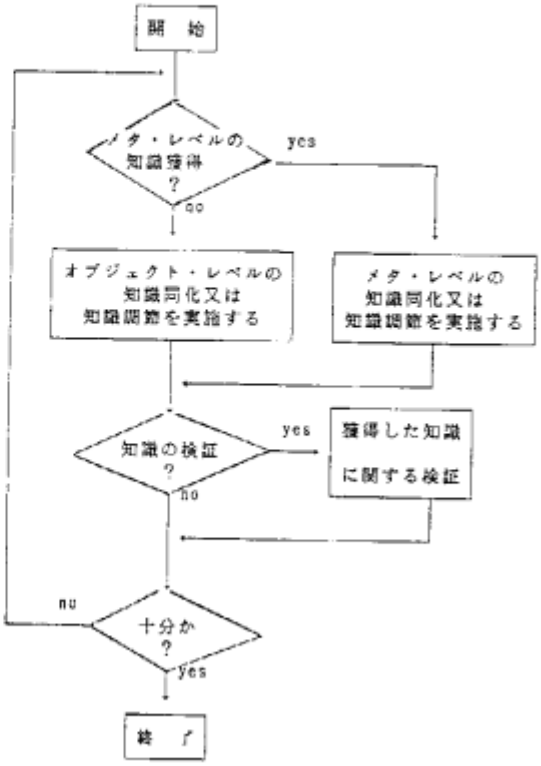


図19．知識獲得プロセス（均衡化プロセス）

獲得される知識は，オブジェクト知識とメタ知識の二種類に大別されているので，図19に示される二種類の知識獲得プロセスが存在すると考えることができる。ピアジェの知性発達モデルで知られる均衡化は，このプロセスを通して達成される。このプロセスに関する例としては，積み木の問題，文法推論の問題としてすでに発表済みであるが，ここでは，ＤＣＧ（Definite Clause Gram- mar）の文法推論を例として，簡単に説明することにする[13,14,15]。

この文法推論においては，オブジェクトレベルの知識獲得が利用されるが，その獲得プロセスにおいて文法推論に対する制約条件としては，次の形式のメタ知識が定義されているものとする。

```
inconsistent(immediate,
  insert( dcg __frame, _ )):-
  s( X,  [ ] ), !, exist_variable(X).
```

（６－１）

これは，文法ルールにより生成される文章には，どの文章も変数（未定義の単語）が含まれてはならないという制約条件である。このような状況で，ユーザが文法ルールに関する知識の断片の一つを知っているものとし，ユーザは，次のような知識同化コマンドを入力することにしよう。

```
| ?- assimilate( dcg _frame,
      [ s(X,Y):- np(X,Z), vp(Z,W)]).
```

（６－２）

ところが，このコマンドの中で，"Ｗ"を"Ｙ"と書かなければならないにもかかわらず，"Ｗ"と書いてしまったとすると，（６－１）式に違反するので，エラーメッセージが出力されることになる。そこで，（６－２）式の"Ｗ"を"Ｙ"に

おきかえて，このコマンドを再実行すると，知識の同化が矛盾なく実行されることになる。次に，文法ルールの断片として同化された知識に基づいて，具体的な文章をいくつか推論してみると，ユーザが具体例としてもっている文章が推論されていないことがある。これは，文法ルールが不完全であることによる。ここでは，出現する文章として前置詞句の表現能力がないとして話を進めることにする。この文法ルールを半自動的に修正するためには，知識調節コマンドを次のように使用すれば良い。

```
? - accomodate( dcg_frame,
        s( [ heroin,walks,in,forest] , [] )).
                                        (6 - 3)
```

ここでは，〔heroin, walks〕の文章までは，文法ルールから推論できるが，〔in, forest〕の前置詞句までは推論できないと仮定すると，〔heroin, walks, in, forest〕の文章が推論できる文法ルールを作りあげるための知識調節コマンドを実行することになる。その結果，文献[4]では，文法ルール"vp(X, Y) :- vi(X, Y)"が"vp(X, Y) :- vi(X, Z), pp(Z, Y)"に修正されることがわかっている。ここでvp(X, Y), vi(X, Y), pp(Z, Y)はおのおの動詞句，自動詞，前置詞句である。

7. 制約型知識のトランザクション処理[15,17]

ここでは，制約型知識を実行するタイミングを明確にするために，トランザクション処理の概念を導入して，制約型知識とトランザクション処理の関係について，述べる。知識ベースに対する更新は，6章で示されたように，知識獲得プロセスの一環として，実施されるが，この更新により，知識が誤った方向に成長することを防止するための判定が制約型知識により常に行われている。この判定のタイミングには，更新があるたびに行われる場合と，ある一連の更新群が終了した後に行われる場合の二つがある。後者は，一連の更新群を定義するためにトランザクションという処理単位が必要であり，このトランザクション処理の中では，遅延型の制約型知識の扱いが重要になってくる。図20に，トランザクション内の一連の更新アクセス群を無矛盾に実行するプログラム例を示す。一般に，トランザクション処理は，排他制御の区切りとしても利用されるが，ここでは，重要でないので，図中のlockとunlock述語については，無視してもらいたい。図20のtransaction述語

```
transaction(X):-
      lock(X),
      execute(X),
      save_transaction(X),
      unlock(X).
transaction(X):-
      restore_transaction(X),
      unlock(X).

execute(X):-
      execute1(X),!,
      maintain_constraints(delayed,X).

execute1(true):-!.
execute1((P,Q)):-
      execute1(P),execute1(Q).
execute1(P):-
      operation_type(P,call(P),!,
      maintain_constraints(immediate,P).
execute1(P):-
      system(P)->call(P);
      clause(P,Q),execute1(Q).
```

図20．トランザクション処理のプログラム例

を実行する第一番目のルールは，トランザクションをexecute述語で実行し，これが成功した場合に，後始末をする処理がsave_transaction述語で行われている。第二番目のルールは，トランザクション処理が失敗したときの処理を示しておりこの後始末は，restore_transaction述語で行われている。execute述語を実現するルールは，図中に示してある通り，トランザクション内のゴール列を実行するexcute1述語と，これにより，遅延型
(delayed-type)の制約型知識を実行するmaintain_constraints述語から成る。maintain述語は，遅延型でも即時型でも評価実行できる述語であり，この述語を実現する三番目のルールに制約型知識の具体的評価が行われている。

以下に，トランザクション処理の例を，図21をもとに説明してみる。図中の最初の例は，is_a（chickweed, vegetable）というis_a関係の知識挿入例であるが，図8の三番目のメタ知識に違反したので，この処理が不成功に終っている。ところが，次の例では，is_a(chickweed, vegetable)および，is_a(vegetable, plant_for_food)を同じトランザクション処理内で知識を挿入しているので，この処理が成功している。

```
| ?- super_concept(frame,X,plant).

X = plant_for_food ;

X = fruit ;

no
| ?- transaction(insert(frame,is_a(chickweed,vegetable))).

* Transaction Error.

yes
| ?- transaction((insert(frame,is_a(chickweed,vegetable)),
|                 insert(frame,is_a(vegetable,plant_for_food)))).

* End Transaction.

yes
| ?- super_concept(frame,X,plant).

X = plant_for_food ;

X = fruit ;

X = chickweed ;

X = vegetable ;

no
```

図21．トランザクション処理の例

8　おわりに

本稿では，Prologによるエキスパートシステムの基本技術として，次のような技術を紹介した。

(1) 構造化知識としての概念階層関係を簡単な形で表現し，この関係に関する継性質を，super-concept, sub-concept, inherit のルールとして整理できることをのべた。

(2) 著者は，demo述語によるメタ推論を利用して，制約型知識の実行方式を示してきた。とりわけ，trigger 型のメタ知識は，自然言語処理で良く使われる因果関係の表現を包含している。

(3) メタ・レベルの知識調節を実現するための基礎技術として，demo述語を利用した矛盾知識の抽出法について簡単に述べた。

(4) 制約型知識の実行に種々の可能性を設けるために，その実行のタイミングを表す概念（immediate-typeとdelayed-type）を取り入れ，トランザクションという処理単位で統合できることをのべた。

demo述語を利用したプログラミング技法は，メタプログラミング技法と呼ばれているが，この技法は，知識ベースのエディタやデバッガの開発，論理プログラムの自動合成，Prologと関係データベースとのインタフェースの実現などにおいてもその有用性が確認されつつある。また，この技法によるコンパイル化技術も開発されつつある。しかしながら，demo述語そのものに対する理論的裏付けについては，現在のところ若干の結果が得られているにすぎない。今後の課題としたい。さらに，曖昧な知識を確信度（Certainty Factor）として表現した場合や，時相論理及び並列処理を導入した場合の知識ベースの管理方法などは，今後の課題である。また，発想推論の問題も今後の大きな課題であることを付け加えておきたい。

［　謝辞　］

本論文の資料は，著者が，（財）新世代コンピュータ技術開発機構（ＩＣＯＴ）へ出向中のときに行った研究の成果をまとめたものである。その研究の機会を与えて頂いたＩＣＯＴの渕・所長，有益な討論をしていただいたＩＣＯＴの古川・第一研究室長および國藤・主任研究員，三菱電気の宮地　氏，の皆様に深謝致します。

〔 参 考 文 献 〕

1) Allen, F. : Recognizing Intention from Natural Language Utterances, in M. Brady et al. (ed.) Computational Model of Discourse, MIT Press (1983).

2) Bowen, D. L. : DECsystem-10 PROLOG [SSR's Manual, DAI, University of Edinburgh (1981).

3) Bowen, K. A. and Kowalski, R. A. : Analgamating Language and Meta-language in Logic Programming, TR 4/81, Syracuse University (1981).

4) Bunge, M. : Causality-the place of the Causal principle in Modern Science, The President & Fellows of Harvard College (1959). (日本語訳では岩波書店, 黒崎宏訳, 「因果性」(1972) をみよ).

5) Chamberlin, D. D. et al. : SEQUEL 2 : A Unified Approach to Definition, Manipulation, and Control, IBM J. RES. DEVELOP. (1976).

6) Furukawa, K., Takeuchi, A. and Kunifuji, S. : Mandala : A Knowledge Representation System in Cuncurrent Prulog, Information Society of Japan, Preprints of WG on Know-ledge Engineering and Artificsal Intelligence (1983).

7) 波多野誼余夫 : 認知心理学講座, 第4巻, 東京大学出版会 (1982).

8) Haraguchi, M. : An Analogy as a Partial Identity, " Proc. of the Logic Programming Conference '84, 11-2, ICOT, Mar. (1984).

9) 市川亀久彌 : 「創造性の科学」, 日本放送出版協会 (1970).

10) 川喜田二郎 : 発想法, 中公新書 (1967).

11) 北上 始, 麻生盛敏, 図藤 進, 宮地泰造, 古川康一 : 知識同化機構の一実現法, 情報処理学会知識工学と人工知能研究会資料30-2 (1983).

12) Kitakami, H., Kunifuji, S., Miyachi, T. and Furukawa, K. : A Methodology for Implementation of a Knowledge Acquisition System, Proc. of the 1984 International Symposium on Logic Programming, Atlantic City, U. S. A., Feb. 6-9 (1984), also available from ICOT as ICOT Technical Report TR 037 (1982).

13) 北上 始, 図藤 進, 宮地泰造, 古川康一 : 大規模な知識ベース管理システムをめざして, 情報処理学会知識工学と人工知能研究会 (1984).

14) 北上 始, 宮地泰造, 古川康一, 図藤 進 : 知識獲得システムの分散処理にむけて(3), 情報処理学会第29回全国大会 (1984).

15) 北上 始, 図藤 進, 宮地泰造, 古川康一 : 知識の同化・調節・均衡化などのユーザ・インターフェースを備えた知識獲得システム, 特集ロジックプログラミング, 日経エレクトロニクス, 11, 5号 (1984).

16) Kitakami, H., Kunifuji, S., Miyachi, T., Furukawa, K., Takeuchi, A., Miyazaki, T., Ishii, S., Takewaki, T. and Ohki, M. : Demonstration of the KAISER System at the ICOT Open House in FGCS '84, ICOT TR-0081 (1984).

17) 北上 始, 図藤 進, 宮地泰造, 古川康一 : 論理型プログラミング言語 Prolog による知識ベース管理システム, 情報処理学会誌, 11月 (1985).

18) Kunifuji, S. and Yokota, H. : PROLOG and Relational Data Bases for Fifth GenerationComputer Systems, Proc. of CERT Workshop on " Logical Bases for Databases " (1982).

19) 図藤 進, 麻生盛敏, 竹内彰一, 坂井 公, 宮地泰造, 北上 始, 横田始夫, 安川秀樹, 古川康一 : Prologによる対象知識とメタ知識の融合とその応用, 情報処理学会知識工学と人工知能研究会30-1 (1983).

20) 図藤　進，北上　始，宮地泰造，古川康一：知識工学の基礎と応用－第4回Prologにおける知識ベースの管理，計測と制御，Vol. 24, No. 6 (1985)

21) 図藤　進：演繹・帰納・発想の推論機構化をめざして，日本創造学会編，創造性研究3，創造と企業，共立出版，(1985).

22) Minsky, M.: Framework for Representing Knowledge, in Winston (ed.): The Psychology of Computer Vision, McGraw-Hill (1975).

23) Miyachi, T., Kunifuji, S., Kitakami, H. and Furukawa, K.: A Knowledge Assimilation Method for Logic Databases, New Generation Computing, 2-4, 385/404 (1984).

24) 宮地泰造，図藤　進，古川康一，北上　始：Constrait に基づく論理データベース管理について，情報処理学会知識工学と人工知能研究会9月 (1984).

25) 三輪和弘，溝口文雄：メタ推論を内蔵したエキスパート・システムの構築，情報処理学会第31回全国大会 (1985).

26) Ong, J., Fogg, D. and Stonebraker, M.: Implementation of Data Abstraction in the Reltaional Database System INGRES, ACM SIGMOD RECORD (1984).

27) Shapiro, E.Y.: Inductive Inference of Theories From Facts, Yale University Research Report 192 (1981).

28) Shapiro, E.Y.: Algorithmic Program Debugging, An ACM Distinguished Dessertation 1982, The MIT Press (1982).

29) 竹内彰一，近藤浩顕，大木　優，古川康一：部分計算のメタプログラミングへの応用，情報処理学会ソフトウェア基礎論研究会 (1985).

30) 田中　譲他：推論とデータベースシステムとの部分評価機構による結合，第1回計測自動制御学会知識工学シンポジウム資料 (1983).

31) Yokota, H., Kunifuji, S., Kakuta, T., Miyazaki, N., Shibayama, S. and Murakami, K.: An Enhanced Inference Mechanism for Generating Relational Algebra Queries, Proc. of Third ACM SIGACT-SIGMOD Symposium on Principles of Database System, Waterloo, Canada, April 24 (1982).

32) Weyhrauch, W.: Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, 13, 13/170 (1980)

〔 付　録 〕

一般的なdemo述語（4引数）のプログラム図22に，4引数のdemo述語のプログラム例を示す。この述語の第1引数には，現在関係している知識の集合体のいくつかがリストとして記述される。第2引数には，証明すべきゴール列が記述され，第3引数には，〔Res, Cond, Cut, Count〕が記述される。Resは，現在，関係している知識の集合体に，一時的に追加される知識であり，demo述語の証明過程で，知識の集合体の一部として利用される。Condは，知識の集合体に存在する知識の中で，証明に利用したくない知識の識別子であり，一般に，その識別子のリストとなる。Cutは，カットシンボルによる制御を行うための作業用変数である。Countは，深さ方向に推論を進めていったときに許される最大推論ステップ数である。これにより，ループによるメモリのパンクを防止できる。第4引数には，推論の結果として，二つの情報が返される。一つ目は，推論が成功したか否かが"true/overflow"で返される。二つ目は，推論過程で利用された仮定型知識だけから成る証明木（前定型知識が除かれている）が返される。

図22のdemo述語を利用した1つのプログラムを図23に示しておく。図23のプログラムは，演繹推論メカニズムを提供するdeduce述語である。この

述語は，知識の同化などに利用されている。dedu
e 述語は，demo述語とは違って，ホーン節（P：
－Q）の証明問題を解く能力がある。図中のsele
ct__variable__list述語は，証明すべきホーン節
Clauseがもつ変数（これは全称限定されている）
を選択する述語である。skolemize 述語は，そこ
で選択された変数に，二度と現れることのない定
数を割り付ける述語である。

```
demo(FRL,true,[Res,Cond,Cut,Count],[true,d(X,X)]):-!.
demo(FRL,Goals,[Res,Cond,Cut,C],[overflow,d(X,X)]):-!.
demo(FRL,!,[Res,Cond,Cut,Count],[Result,d(X,Y)]).
demo(FRL,!,[Res,Cond,cut,Count],[Result,d(X,Y)]).
demo(FRL,(P;Q),[Res,Cond,Cut,Count],[Result,d(X,Y)]):-!,
    (demo(FRL,P,[Res,Cond,Cut,Count],[Result,d(X,Y)]) ;
     demo(FRL,Q,[Res,Cond,Cut,Count],[Result,d(X,Y)])).
demo(FRL,(P,Q),[Res,Cond,Cut,Count],[Result,d(X,Z)]):-!,
    demo(FRL,P,[Res,Cond,Value,Count],[Result1,d(X,Y)]),
    (Value==cut,Cut=cut,Result=Result1,Z=Y,! ;
     Result1=true->demo(FRL,Q,[Res,Cond,Cut,Count],[Result,d(Y,Z)]) ;
     Result=Result1).
demo(FRL,P,[Res,Cond,Cut,Count],[Result,d(X,Y)]):-
    system(P)->P,Result=true,X=Y ;
    Count1 is Count-1,
    clause_of(FRL,ID,(P:-Q),Certainty,[Res,Cond]),
    (Certainty==premise->X=Z ; X=[(P:-Q)|Z]),
    demo(FRL,Q,[Res,Cond,Cut,Count1],[Result1,d(Z,Y)]),
    (Cut==cut,!,fail ; true).

clause_of((FR,FRL),ID,(Head:-Goals),Certainty,[Res,Cond]):-!,
     (clause_of(FR,ID,(Head:-Goals),Certainty,[Res,Cond]) ;
      clause_of(FRL,ID,(Head:-Goals),Certainty,[Res,Cond])) .
clause_of(FR,ID,(Head:-Goals),Certainty,[Res,Cond]):-
     next_clause(FR,ID,Clause,Certainty,Cond),
     ( Clause=(Head:-Goals) ;
       Clause=Head,Goals=true  ) .
clause_of(FR,[],(Head:-true),Certainty,[Res,Cond]):-
     Res\==[],unify(Res,Head).

next_clause(FR,ID,Clause,Certainty,Cond):-
     next_clause1(FR,ID,Clause,Certainty,ID1),
     Clause\==[],check_condition(ID,Cond).

next_clause1(FR,[P1,P2,P3],Clause,Certainty,[P1,P2,P3]):-
     not(var(P1)),X=..[FR,P1,P2,P3,Clause,Certainty],X.
next_clause1(FR,ID,Clause,Certainty,[P1,P2,P3]):-
     var(P1),!,dictionary(FR,[P1,_,_],_),
     X=..[FR,P1,P2,P3,_,Certainty],X,!,
     next_clause1(FR,ID,Clause,Certainty,[P1,P2,P3]).
next_clause1(FR,ID,Clause,Certainty,[P1,P2,P3]):-
     X=..[FR,P2,Q2,Q3,_,_],X,!,
     next_clause1(FR,ID,Clause,Certainty,[P2,Q2,Q3]).

check_condition(ID,[]).
check_condition(ID,[ID|Y]):-!,fail.
check_condition(ID,[X|Y]):-check_condition(ID,Y).

unify((C,CL),P):-!,
     (unify(C,P);unify(CL,P)) .
unify(P,P).
```

図22　4引数 demo 述語のプログラム例

```
/* Demonstrate existential/universal Clause. */

demonstrate(FPL,Clause,Conc):-
  verify( (select_variable_list(Clause,Variable_list),
   skolemize(Variable_list),
   (Clause=(P:-Q)->demo(FPL,P,[Q,Conc,Cut,50],[true,[]]);
    demo(FPL,Clause,[[],Conc,Cut,50],[true,[]])) ).

verify(P):- \+(\+(P)).

select_variable_list(Clause,Variable_list):-!,
     select_variable(Clause,[],Variable_list).
select_variable((P:-Q),Vs,Vf):-!,
     select_variable(P,Vs,Vi),select_variable(Q,Vi,Vf).
select_variable((P,Q),Vs,Vf):-!,
     select_variable(P,Vs,Vi),select_variable(Q,Vi,Vf).
select_variable((P;Q),Vs,Vf):-!,
     select_variable(P,Vs,Vi),select_variable(Q,Vi,Vf).
select_variable(P,Vs,Vf):-!,
     select_variable1(P,Vs,Vf).

select_variable1(P,Vs,Vs):-
     atomic(P),!.
select_variable1(P,Vs,Vf):-
     var(P),!,unique_variable(P,Vs,Vf).
select_variable1(P,Vs,Vf):-
     P=..[Predicate_name|Arguments],
     select_variable2(Arguments,Vs,Vf).

select_variable2([],Vs,Vs):-!.
select_variable2([A1|A2],Vs,Vf):-
     select_variable1(A1,Vs,Vi),
     select_variable2(A2,Vi,Vf).

unique_variable(X,[],[X]).
unique_variable(X,[Y|Z],[Y|Z]):-
     X==Y,!.
unique_variable(X,[Y|Z],[Y|W]):-!,
     unique_variable(X,Z,W).

skolemize(X):-!,skolemize1(X,0).
skolemize1([X|Y],Count):-
     name('$',[L1]),Count1 is Count+1,name(Count1,L2),
     name(X,[L1|L2]),!,skolemize1(Y,Count1).
skolemize1([],Count).
```

図23　deduce述語のプログラム例