ICOT Technical Report: TR-134

TR-134

知識獲得システム

北上　始

May, 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

知識獲得システム

北　上　　始

1. はじめに

人工知能の研究の歴史をふりかえってみると，1960年代前半はゲームとパズルの時代，1960年代後半は知能ロボットの時代，1970年代は言語と知識の時代といわれている[10]．さらに，1980年代は知識工学と認知科学の時代であるといわれるようになってきた．そこで，知識工学や認知科学の現状を見渡すと，知識の表現，知識の利用，および知識の獲得などのテーマが研究されている．これらのテーマは，エキスパートシステムを構築するための基礎的な研究テーマであると見ることができる．

エキスパートシステムは，種々の分野のエキスパートがもつ問題解決能力を支援するシステムである．この能力をコンピュータシステムにもたせるためには，問題解決のために使用する知識を知識ベースに蓄積・管理する必要がある．そのためには，エキスパートがもつ知識をコンピュータ処理できる形式に表現し，それを系統的に獲得し，獲得された知識を容易に利用できる機能を実現する必要がある．このようなシステムは，知識ベース管理システムとして知られている．

著者は，第五世代コンピュータプロジェクトの一環として，論理型プログラミング言語Prologを知識ベース管理システムのインプリメンテーション言語と仮定し，そのシステムの実現法を研究してきた．知識ベース管理の実現のために，従来の関係データベース管理[9]の枠を知識化されたプログラムの領域にまで拡張して，研究を進めていかなければならない．そのために，著者は，データベース，認知心理学，形式論理学，プログラミング方法論などといった種々の分野の研究を有機的に統合し，新しい諸概念の研究を進めていかなければならないと考えている．そこでは，ロジック・プログラミング用言語Prologが知識ベース管理の研究をするうえで最も強力な言語であると判断している．

本稿では，知識獲得機能を中心に，知識ベース管理の基本技術について述べ，さらに管理システムとしてとらえた場合の各要素技術の相互関係についても述べてみたい．なお，本稿で，取り扱う知識は，論理型プログラミング言語Prologで記述可能なホーン節であると限定し，プログラミング用言語のシンタックスは，DEC-10 Prolog[2]を前提としている．

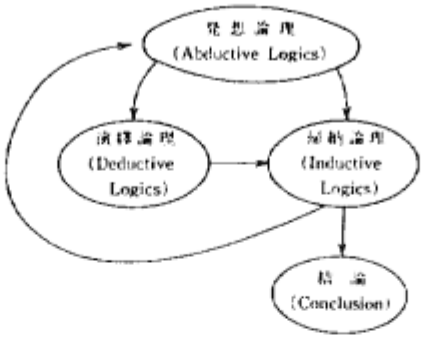


図1　人間の探求過程における三つの論理の関係．
パースが示した体系を整理したもの．本稿の議論は，この体系に基づいている．

2. 記号学から知識獲得

19世紀後半から20世紀初頭にかけて多くの著作を残したプラグマティスト，パース（C.S.Peirce）は，先駆的な記号学者として，人間の探求活動の全過程を演繹論理，帰納論理，発想論理の三つの論理の立場から体系化した[11]．図1は，この三論理の関係を整理した図である[23,24]．彼の記号学によれば，人間の探求過程とは，ある未知の問題に遭遇した人間が，その問題を解決する多くの概念のなかから面白い仮説を発見（発想）し，その仮説から導き出される帰結を推論（演繹）し，そしてその帰結を実験的に検証しながら実験結果に合致する仮説を構築していく（帰納）過程である．実験は，発見されたおもしろい仮説に対する多くの事実を収集し，それらの事実を使って，さらに精密な仮説を作成するために不可欠な探求過程である．後で説明するが，第五世代コンピュータ・プロジェクトでは，パースの哲学に基づいて，演繹推論，帰納推論，発想推論を統合した知識獲得システムの実現方式を研究し，これを知識ベース管理システムのサブシステムとして位置づけている[2,16,20,21]．

3. 論理型プログラミング言語 Prolog の特徴

ここでは，知識獲得システムを実現するときに，最も強力なインプリメンテーション用言語 Prolog の特徴について述べてみよう．Prolog はホーン論理としての特徴を十分に備えたホーン節形式の言語であり，ホーン節は“帰結部：㊀条件部”の形式をしている．“：-”は含意記号であり，「もし“条件部”が成立するならば“帰結部”が成立する」と読む．この条件部には論理和や論理積で表されるゴール列を記述できるが，帰結部には，それらの記述を許していない．

ホーン節は，変数項を含まない基本節（Ground Clause）と，変数項（全称限定されている）を含む一般節（General Clause）に分類される．条件部が存在しない基本節を，ファクト（Fact）と呼び，それ以外をすべてルール（Rule）と呼ぶこともある．以下では，Prolog プログラムの構成要素ともいえるこのホーン節を，特に「知識」と呼ぶことにする．ホーン節が表わす知識の概念名は，その帰結部に示されている述語名になっている．

それでは，以下に Prolog の特徴を順に示していこう[20]．以下の特徴は，Prolog が，知識獲得システムの研究を加速させている理由でもある．

【特徴1】 ホーン節が持つ任意の二変数を単一化（ユニフィケーション）すること，または任意のアトムをそのホーン節の条件部に付加することは，ホーン節そのものが持つ概念をさらに特殊化することを意味する．たとえば，「Xが鳥ならば，Xは飛べる(canfly(X):-bird(X))」というホーン節が持つ概念“canfly（X)”と，「Xが鳥でかつペンギンでないならば，Xは飛べる(canfly(X):-bird(X), not(penguin(X))」というホーン節が持つ概念“canfly(X)”を比較してみると，後者の概念の方が特殊な狭い意味を持つ．ここで“not”は真偽値を反転させる述語である．

【特徴2】 あるプログラムを使って推論を実施したとき，その推論プロセスで利用したホーン節の利用関係が明確なので，プログラマの意図に反した推論結果を出力しても，その誤りの原因を容易に検出できる．また，この推論プロセスは，人間の基本的な思考プロセスによく似ているという意味で自然である．

【特徴3】 Prolog プログラムは，モジュラリティが高く，単調性を持たせやすい．すなわち，ある概念を修正するために，あるホーン節を追加または削除することにより，それぞれ，より多いまたはより少ない推論結果を期待させることができる．これにより，プログラムの修正は，ある規則的な概念変化であるとみなすことができるので，概念変化が予想できる．たとえば，“above(X,Y):-on(X,Y).”というプログラムに“above(X,Y):-on(X,Z), above(Z,Y).”を追加すると，“above”という概念は，より多くのインスタンスを推論する．

【特徴4】 以上は Prolog の言語を論理ベースで設計したことによる特徴であるが，このほかに，Prolog には，プログラムを最適実行させるために，カット・オペレータと呼ばれる推論制御用述語がある．

【特徴5】 さらに Prolog にはプログラミング言語としてデータとプログラムの区別がなく，両者は等価である．ここで，両者は各々，ファクトとルールで表現される．そのため，データの追加はただちに推論結果として反映される．この等価性の意味は，ほかのプログラミング言語より強い意味を持っているといえる．

4. 知識ベースの構造

知識ベースの構造について話を進める前に，人間がお互いに行っているコミュニケーションに注目してみよう．このコミュニケーションは，人間のもつ知識ベースの知識を，増大・拡張させ，種々の問題解決能力を高める役割をもっていることに気がつくだろう．例として，日本人と日本人の間のコミュニケーションと，日本人と外人の間のコミュニケーションについて考えてみると，同じ日本人の間のコミュニケーションは，言語，習慣などに隔たりがない（コミュニケーション・スキーマが同じ）ので，円滑にコミュニケーションを進めることができるが，日本人と外人とのコミュニケーションでは，言語，習慣などの面で隔たりがある（コミュニケーション・スキーマが異なる）ので，それを理解しない限り，円滑なコミュニケーションを期待できない．

このような状況を，類推的に，計算機システムの知識ベースの動作環境に当てはめてみよう．計算機システムでは，以下の三者の間のコミュニケーションを対象にしていることになる．

①ある計算機システム（主サイトとする）を使うユーザ．

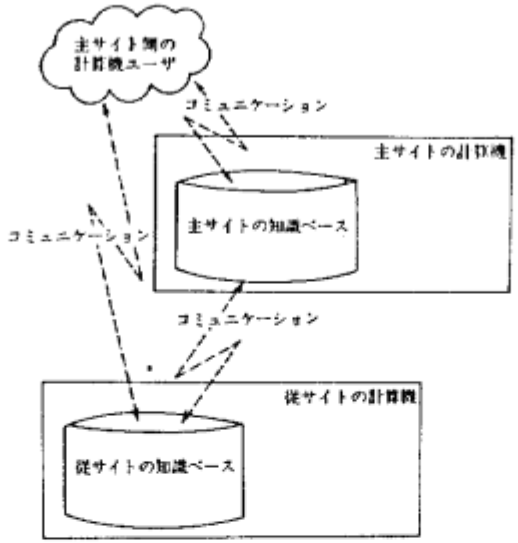


図 2 知識ベース間のコミュニケーション

これは，エキスパートまたはエンドユーザと呼ばれる計算機ユーザである．

②ある計算機システム（主サイト）内の知識ベース

③他の計算機システム（従サイトとする）内の知識ベース

上記三者は，それぞれ，独自のコミュニケーション・スキーマをもっているので，任意の二者間のコミュニケーションを確立するためには，コミュニケーション・プロトコルを整理し，それを適宜，利用していかなければならない．この三者に対する可能な二者の組み合わせのうち，図2に示すように，①と②，①と③，②と②(②内部のコミュニケーション)，②と③のコミュニケーションが重要であるが，本稿では，知識ベース管理という立場から特に重要な①と②，②と②，②と③のコミュニケーションに重点をおいて，知識ベースの構造を決めていくことにする．

知識ベースの構造は，知識の利用および蓄積にとって簡単な構造になっていると同時に，知識の獲得を継続的に実施できるようにするために，拡張性に富んだ柔軟な構造をしていなければならない．ここでは，Minskyのフレーム理論にならい[20]，知識ベースはフレームと呼ばれる知識の集合から構成されるとする．フレームは，ある種の限定された領域に対応する知識のかたまりである．また，フレームの集合としての知識ベースは，既に述べたように，他の知識ベースと有機的な相互関係をコミュニケーションにより，達成できる構造をもっているとする．フレーム中に蓄積されている知識は，変化する可能性がある知識と，変化する可能性がない知識がある．本稿では，前者を仮定型(assumption-type) 知識，後者を前提型 (premise-type) 知識と呼ぶ．

以上の議論をもとに，段階的に知識ベースの構造を明らかにしていこう．ただし，4節から8節までの間で使用する例は，付録1に示されている例を使う．

(1) フレーム間のコミュニケーション

知識ベースには，多数のフレームが存在し，各フレームには，知識のかたまりが蓄積されていることから，これらのフレームをオブジェクト・フレームと呼ぶことにする．また，各オブジェクト・フレームを一元管理するためにメタ・フレームと呼ばれるフレームを使うことにする．このメタ・フレームの構造は，オブジェクト・フレームと同じ構造をもっている．他の知識ベースとのコミュニケーションは，直接，オブジェクト・フレーム間で行われるが，それに必要な情報は，メタ・フレームから得られる．例えば，知識ベース間のコミュニケーションを行うときに，お互いの知識表現形式が異なるときは，お互いに機能的な整合性をとるためのコミュニケーション・プロトコル（メタ・フレームに蓄積されている）がメタ・フレームから得られる．これにより，フレームは他のフレームに必要な情報を的確に送ることができる．そのための基礎技術としては，後述するdemo述語による部分評価方式が有効であることがわかっている[17,19,25,37,40]．

フレーム間のコミュニケーションとしては，ここで述べた，(1)主サイトの知識ベースと従サイトの知識ベースとのコミュニケーションの他に，(2)主サイトにおけるオブジェクト・フレーム間のコミュニケーションがある．両コミュニケーションとも，図3に示すようにほぼ同じ処理方式をとることにより達成でき，並列処理による処理効率の向上が期待される．上記，(1)のコミュニケーションを例にとって，その処理方法を示してみよう．図4は，このコミュ

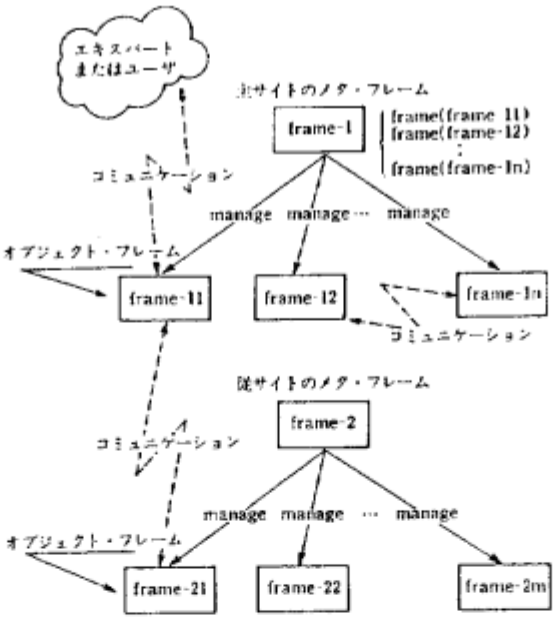


図 3 フレーム間のコミュニケーション

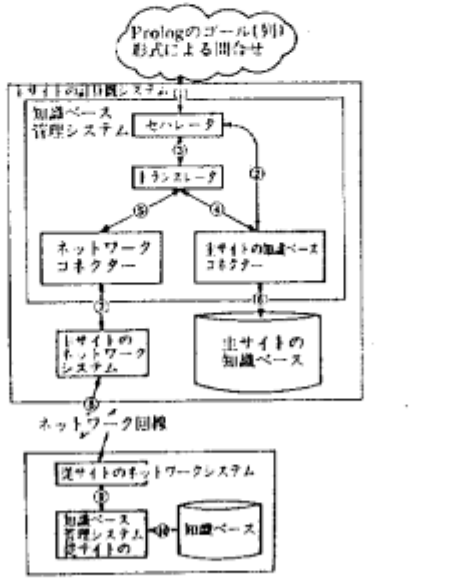


図 4 コミュニケーション処理の論理的な流れ図.

ニケーション処理を行うときの論理的な流れ図であり，この処理を行うための機能は，知識ベース管理システムの一機能であるといえる．

主サイトのあるフレームに問合せをするために，図4の知識ベース管理システムに，Prologのゴール（列）形式による問合せが渡されると，図中のセパレータは，①→②→⑥のパスを使って，そのゴール列をユーザが指定したフレームの上で，部分評価し，他のフレーム（従サイトのフレームも含む）で評価すべきゴール列を抽出する．部分評価については，後述するdemo述語を利用することによって達成される．抽出されたゴール列は，ユーザが指定したフレーム内だけで評価できなかったゴール列であり，このゴールは，従サイトのフレームも含めて，評価すべきフレームごとにクラスタリングされている．このクラスタリングされたゴール列は，③のパスを使ってトランスレータに渡される．トランスレータは，上記セパレータの分類に従って抽出されたゴール列を，各フレームのコミュニケーション・スキーマに合ったコマンド列に変換(プロトコル変換）する．変換されたコマンド列は，その評価を主サイトで実施するか従サイトで実施するかによって，④または⑤のパスに送られる．④のパスを経由してコマンド列を受け取った主サイトの知識ベース・コネクターは，主サイトの知識ベースにコマンドを送る．その結果，問合せの答としてのインスタンスの集合が得られる．⑤のパスを通過して，コマンド列を受け取ったネットワークコネクターは，従サイトのコミュニケーション・スキーマに合わせて変換されたコマンド列を⑦←→⑧←→⑨←→⑩の送／受信パスで利用されるパケットデータに変換する．これは，主サイトの知識ベース管理システムが推論レベルで記号処理可能な形に変換する機能である．

（2） フレームの論理構造

フレームは，図5に示されるように，五つのカテゴリーに分類された知識と，それらの推論手続きに利用されている推論エンジンから構成されると見做すことができる．この構成は，著者らの独自のアイデアで整理されている．五つのカテゴリーに分類される知識には，（1）構造化知識，（2）プログラム化知識，（3）制約型知識，（4）オペレーション型知識，（5）辞書型知識がある[16,23]．

よく議論されているように知識は，オブジェクト・レベルの知識とメタ・レ

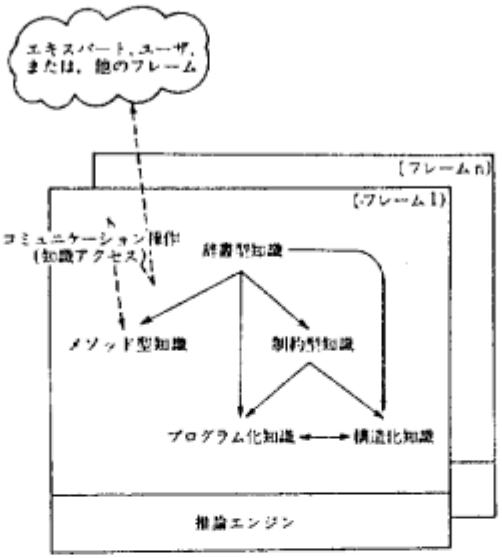


図 5 フレームの論理構成

ベルの知識があり，これらは，普通，われわれがプログラミングの世界で認識している知識であり，構造化プログラムなどといわれるようにプログラムのデータ構造を与える（1）の構造化知識と，それに基づいて作成される（2）のプログラム化知識がある．メタ・レベルの知識の代表的なものには，データ辞書と呼ばれるものを拡張した（5）の辞書型知識がある．オブジェクト・レベルの知識と（5）の辞書型知識の間には，非常に重要な二つのメタ・レベルの知識がある．1つは，知識獲得時に，オブジェクト・レベルの知識が思わぬ方向に成長することを防止するために設定されるメタ・レベルの知識であり，(3)の制約型知識がそれに相当する．他の1つは，前節まで何度も議論の対象になっているコミュニケーション・スキーマを表わす知識である．フレーム内の諸知識は，このコミュニケーション・スキーマを通して外のフレームとのコミュニケーションが行なわれる．ここでは，この知識を（4）でメソッド型知識と呼んでいる．

以下に，この五つのカテゴリに分類される知識について，そのイメージをさらに明らかにしておこう．

構造化知識は，種々の概念をわかりやすくかつ容易に表現するために必要な構造化された知識であり，自然言語処理の分野で知られる概念階層関係がそれに相当する．概念階層関係には，付録1の（1-1）で例示されているようにproperty，is_a，part_of関係があるが，それぞれ次のように表現できる．

property（概念名，特性評価用の述語）． (4-1)

is_a（概念名，上位概念名）． (4-2)

part_of（概念名，下位概念名）． (4-3)

これらは，概念間の関係を表現しているので，一階述語表現の世界の概念を対象化し，高階述語表現へと飛躍するためのある種の側面であるともいえる．

property関係は，ある概念がもつ特性がどのような述語によって評価されるかを示す関係である．is_a関係は，ある概念の一般概念がもつ諸特性を，下位概念にも継承させられるという性質がある．part_of関係は，ある概念に関する部分概念を表現する関係であり，下位概念がもつ諸特性を，上位の全体を表わす概念にも継承させられるという性質がある．

プログラム化知識は，付録1の（1-2）で例示されているように，構造化できなかった概念を手続き的な知識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，プログラム化知識の量は少なく済み，その逆もまた真であるといえる．

制約型知識は，付録1の（1-3）で例示されているように前述の構造化知識およびプログラム化知識に対する概念的な枠組みを示すメタ知識である．このカテゴリーでは，主に，両知識を成長させるときに，ユーザの意図に反した枠を超えて成長していくことがないように，種々の制約条件が記述される．このメタ知識の表現には，次の二種類がある．

一つは，知識ベースの更新などに対する一貫性を維持するために，種々の知識を自動的に更新するメタ知識であり，トリガー（trigger）と呼ばれる．他の一つは，知識ベースの更新に対する矛盾（inconsistency）を判定するメタ知識であり，このメタ知識は，否定型ルールと見做すことができる．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，更新などの知識アクセスがあるごとに起動する即時型（immediate type）のメタ知識に分類される．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． (4-4)

inconsistent（起動タイプ，オペレーション）:-

　　起動条件，!，矛盾判定条件． (4-5)

両知識とも，起動タイプとして，immediate，delayed のタイプをもつ．両知識中に表わされている記号“!”は，カットシンボルと呼ばれるメタ述語である．オペレーションには，メソッドによる知識ベースの更新命令が記述される．

メソッド型知識は，フレーム内の諸知識に対する問合せ，更新操作のために用意された知識である．エキスパート，ユーザ，または他のフレームは，この知識を利用して，フレーム内の知識とコミュニケーションを行うことができる．この詳細については，6節で述べることにする．

辞書型知識は，知識の拡大，種々のインタフェースの向上などのために，さらに高度な処理を実現する必要があるときに利用される．この知識としては，上記，四種類の知識に対する知識のカテゴリー名，個々の知識がもつ構文上の枠組み，使用上の制限（データタイプ，入出力仕様）などが挙げられる．データタイプは，抽象データタイプの考え方を採用し，Prolog の性質を上手に利用すると，ユーザにより，適宜，データタイプを拡張していくことができる[22]．

（3） 知識の格納形式

フレーム内の知識は，すべてホーン型の知識であり，どの知識にも確信の程度を表わすパラメータが付けられている．これらの知識をフレーム名も含めて表現すると，次のように表現される．

フレーム名（KID，ホーン型の知識，確信の程度）． (4-6)

ここで，“KID”は，知識の識別子であり，これは，（1）“KID”に基づく直接検索をし，（2）冗長知識間の識別を行い，（3）知識の格納管理をする上で順序関係を維持するために必要不可欠である．確信の程度の表現方法としては，確信度（0～1）を利用する方法もあるが，本論文では，簡単のため，仮定型知識（assumption-type knowledge），前提型知識（premis-type knowledge）の二つに分類して，議論を進めることにする．

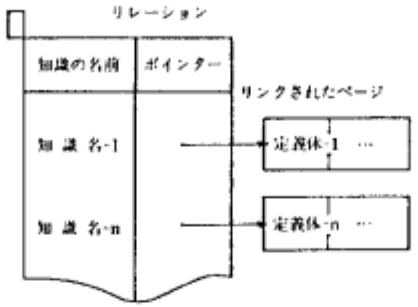


図 6 関係データベースによるテキストデータの管理

フレーム中の五種類の知識は，いずれもホーン節の基本節または，一般節のどちらかで表現されているが，両節ともに，主記憶上で使用しない限り，二次記憶上に蓄積しなければならない．

前提部分（条件部）のない基本節（ファクト）は，関係データベースのタップルとして蓄積できるが，それ以外の節（ルール）は，その構造が可変長の構造になっているので，テキスト・データと同じ扱いをしなければならない．テキスト・データを関係データベースで実現するための一つの方法としては，図6に示す方法がある．

この方法によると，可変構造をもつ知識を，必要な数だけリンクしたページ内に蓄積できる．また，テキスト・データを簡単にアクセスするためには，更に知的なインタフェースを考える必要がある．

このような事情から，同じ知識名（概念名）をもつファクトは，一つのリレーションに蓄積され，その知識名は，リレーション名と一致させることができる．また，ルール（ファクトを混在させてもよい）を，図6に示すリレーションに蓄積できるが，このリレーション名をシステムのユーザに見せないようにしなければならない．

5. メタ推論

これまでにも述べてきたように，知識ベース内の知識は，通常の Prolog プログラムとしての知識よりも，多くの制御構造をもった複雑なデータ構造の中

に格納されている．この複雑な構造は，知識ベースそのものが拡張性に富んだ柔軟な構造をし，多くの問題解決に耐えられなければならないという現実的な制約から生じている．したがって，単純な証明機能だけしかもたない“demo (Goals)”形式の述語だけで知識ベース管理システムをインプリメントするのは，不十分であり，“demo(Frame, Goals, Control, Result)”形式の demo 述語を作成・使用しなければならなくなる[10]．以下，この述語による推論をメタ推論と呼ぶことにする．この demo 述語は，そのフレーム (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，その証明プロセスの中からユーザが必要とするメタ情報(Result)を抽出することができる[10]．この多種の機能をもつ demo 述語は，知識ベース管理システムをインプリメントするために非常に有効である．これ以外に，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，本論文の説明では，説明を簡単にするために，この多種の機能をもつ demo 述語を採用することを避け，問題ごとに特殊化(抽象化)した demo 述語で解説を試みることにする．ここで，メタ知識とは，“既知の知識の扱い方”を表現する知識である．エキスパートは，このメタ知識を demo 述語を使用して容易に表現することができる．

(1) 理論的根拠

ここでは，メタ推論機構の理論的枠組みを簡単に示すことにしよう[27]．

Lを与えられた論理体系，PをLにおける論理式の有限集合，QをLにおけるある論理式とする．QがLにおいてPから証明可能である時，

$$P \vdash_L Q \qquad (5\text{-}1)$$

と表わすことにする．式 (5-1) が成立する時，QをLの定理と呼ぶこともある．

二つの論理体系L，Mにおいて，LからMへの一対一の対応付け関数 (coding function)「′」が定められているものとする．このときメタ推論を行うためにプリミティブ述語 demo を，次のようにして導入する．

$$Pr \vdash_M \mathrm{demo}\ (P', Q') \equiv P \vdash_L Q \qquad (5\text{-}2)$$

ここに Pr は M における論理式の集合，P′ や Q′ はそれぞれ P や Q に関数「′」を適用した結果である．Weyhrauch は式 (5-2) を反射原理 (reflection principle) と呼んでいる[39]．式 (5-2) は，対象言語上の定理をメタ言語上で実証あるいは模倣するための基本原理とみなすこともできる．Bowen らは二つの論理体系 L と M が同一の形式的言語で表現できる場合に着目し，知識ベースに更新，すなわち知識の同化 (Knowledge Assimilation) における demo 述語の重要性を指摘した[9]．彼らはこれを対象言語とメタ言語の融合 (amalgamation) と呼んだ．メタ推論方式が有効なのは，論理式の有限集合 P の無矛盾性を保てる範囲であり，基本原理 (5-2) の性質が常に成立する範囲を明確化することが必要である．

(2) demo 述語の実現法

ここでは，簡単のため，式 (4-5) で“KID”と確信の程度についての項目を除外した場合のフレーム名付きの知識を対象として，demo 述語の実現法について述べる．この条件下での demo 述語は，最も簡単な“demo (Goals)”形式の demo 述語を“demo (Frame, Goals)”形式にまで拡張すればよいことがわかる．これによると，図 7 に示すような二引数 demo 述語を得ることができる．この demo 述語は，Prolog で記述された Prolog のインタプリタになっている．図の第一行目のルールは，ゴールを証明していったときの停止条件である．第二行目のルールは，論理積の展開証明を行っている．第三行目のルールは，“P”と単一化可能な帰結部をもつホーン節“(P:-Q)”をさがし，“Q”の証明を行っている．また，この clause_of 述語は，フレーム“Frame”内で単一化可能な節をみつける述語であり，“Frame”に関する扱いを除いては，DEC-10 Prolog の clause 述語（システム述語）と等価である．以下では，フレーム内に蓄積されるホーン節の知識は Prolog で記述されるような形式の知識であ

```
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).
```

図 7 二引数の demo 述語

るとして，話を進めている．したがって，推論用述語および単一化による検索機能をもつ述語は，それぞれこの demo 述語および clause _of 述語に相当する．

6. メソッド型知識

本節では，知識ベースのフレームに対するコミュニケーション操作（知識アクセス）として，最も重要かつ興味ある問題で知られる知識獲得の処理機能を提供するメソッド型知識の説明をしていくことにする．

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・レベルの知識のほかに，制約型知識，メソッド型知識，辞書型知識といったメタ・レベルの知識があることは既に述べた．この時の知識獲得能力は，認知心理学で知られるピアジェの知性発達モデル[2]を参考に実現されており，他に例をみない構成をしている[16,20,21]．

このモデルは，図8の最上位階層であり，その階層では，メソッド型知識の能力を人間の知識獲得能力としてとらえ，知識の同化(Assimilation)[15,16,20,21,30]，調節 (Accommodation)[16,20,21]をはじめとする種々の機能をエキスパートに提供している．知識同化は，知識ベースのフレームに曖昧な新知識(assumption-type の新知識)を無矛盾に追加する機能である．知識調節は，知識ベースのフレームに正しい新知識 (premise-type の新知識) を

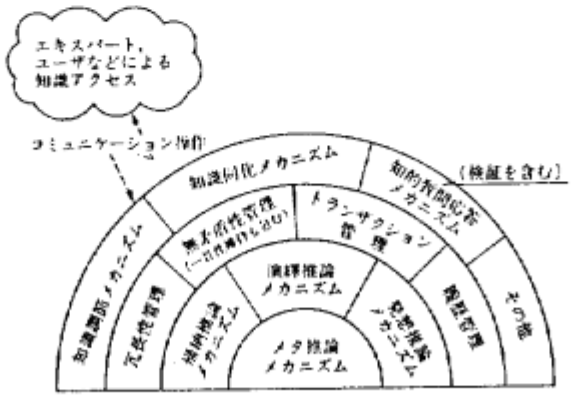


図 8 知識獲得機能のダイヤグラム

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機能である．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) 能力をもつ知識が存在する．

さて，知識獲得機能を支える基礎としては，最初にメタ推論メカニズムを挙げることができる．メタ推論メカニズムは Prolog インタプリタが持つ推論能力だけではなく，ユーザの手によって推論プロセスそのものを制御するときに必要である．このメカニズムは，これを自由自在にあやつれる推論エンジンとも見做せる．図8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には，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）を解く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導出可能か否かを解決する役割を持つ．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，このメカニズムは，Shapiro が研究したモデル推論アルゴリズムを高速化などのために改善し，実現されている[16]．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なうメカニズムが必要であると考えている[13]．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が，現在のところ，履歴管理を除いてすべての処理方式が明らかにされている．履歴管理は，時系列の知識に対する時間的な推論能力を中心に扱う機能と考えている．

次にエキスパートの手によって，いかにしてオペレーション型知識を利用するのかについて説明してみよう．獲得される知識には，オブジェクト・レベル知識とメタ・レベル知識の二種類に大別されているので，図9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実行例は，9節を参照されたい．

以下，知識獲得の基本機能を示すために，本節の（1）項で，冗長性管理，(2)項で帰納推論アルゴリズムについて述べる．無矛盾性管理については，改めて8節で述べることにし，それに関連するトランザクション管理の方式につ

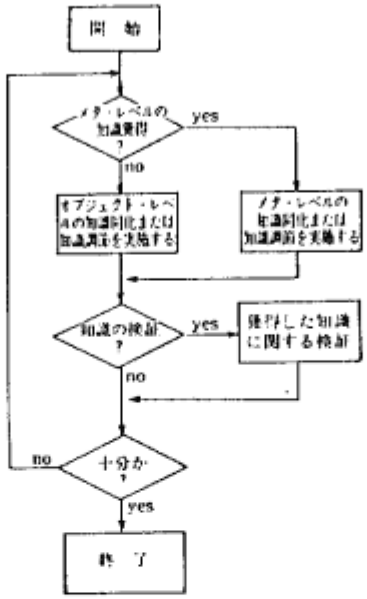


図 9 知識獲得プロセス（均衡化プロセス）

いては，8節の（5）項を参照されたい．

（1） 冗長性管理

実世界に存在する多種の知識は，利用者の目的に応じて非常に利用価値の高い知識とそうでない知識とに分類できる．利用価値の高い知識とは，高い信頼性，高い品質，高い処理効率を持つ知識を指す．

知識に高い信頼性を持たせるためには，知識に矛盾を持たせないようにする必要がある．これを，知識の無矛盾性管理と呼んでいる．

知識に高い品質を持たせるためには，できるだけ冗長でない知識を集める必要がある．いくつかの知識の間に冗長性があるとき，どの知識を除去すべきかという問題が発生するが，これは，知識の真実性（Verisimilituve）という考え方を導入すると，ある程度，解決できる[33]．ここでは，この真実性の考え方を，きわめて制限したかたちで説明していることになる．例えば，あるルールとあるファクトが共に仮定型知識であると同時に冗長であるときは，多くの事

実を説明できるルールの方に真実性をもたせている．

知識の処理効率を高めるためには，できるだけ簡単な推論操作で結果を出せるように整理する必要がある．そのためには，冗長な知識であっても，推論時間を短くするのに役立つ知識もある．たとえば，ルールとして表現されている知識と並べて，よく使われるルールの例（インスタンス）を集めておくと，推論を行なわずとも，単なるデータ検索だけで問題解決を実施できる場合があるので，処理時間を短縮するという効果がある．

以上からも明白なように，知識の品質の高さと，処理効率は，一般に相反することが多く，両者は，問題向きに調和をとっていかなければならない（冗長性管理）．現在のところ，この両者の調和をとる一般的方策があまりないので，著者は，エキスパートが問題ごとに，判断すべきものであると考えている．これにより，以下では，冗長性除去の方式について述べていくことにする．

データベースの分野では，冗長性除去の問題をソーティングやBトリーの技術などにより解決してきた．この分野における冗長性除去の問題は，データ（ファクト）の重複を除去することで解決できるが，ルールも含めて冗長性除去を行なう問題は，そう簡単に解決できそうもない問題のように思える．それは，知識ベースの世界では，ファクトとルールが一体となっており，それらの知識を対象に冗長性の概念をいかに定義すればよいのかという基本的な問題設定から始めなければならないからである．

われわれは，ある知識群Gからある特定の知識“Head:-Body”を取り出し，その知識が証明されるとき，その知識“Head:-Body”は，知識群Gで冗長であると定義している．Gから“Head:-Body”を証明できるか否かの判定は，demo述語で判定できそうだが，demo述語は，ゴール（列）の証明可能性を判定する能力だけしか持っていないので，ホーン節“Head:-Body”の証明可能性を判定する述語を作成する必要がある．著者は，この述語をdeduce述語と呼び，図10に示すインプリメンテーションの方式を採用している[6]．

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:-

```
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．

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説明してみよう．"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[10]．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，[ ] を指定することにしている．ここで，図中に，"(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．

以上の deduce 述語を利用して，冗長性除去のプログラムをいかに書けばよいかを簡単に説明してみよう．図 11 に示すように，冗長性除去の対象となる Prolog プログラムを A とすると，そのプログラムを構成するホーン節（ルールまたはファクト）が格納順に並べられている．この格納順を管理するために，ホーン節を識別するための識別子（Identifier）を KID 1，KID 2，……などで表現する．プログラム A に対する冗長性判定は，各ホーン節に対して実施される．"KID 1" から順にプログラム A のホーン節を取り出し，deduce 述語で証明可能性を調べる．もし，あるホーン節がそのホーン節を除くプログラム A から証明可能であれば冗長であるとみなすことができる．この判定は，図中の最後のホーン節-n に至るまで実施しなければならない．

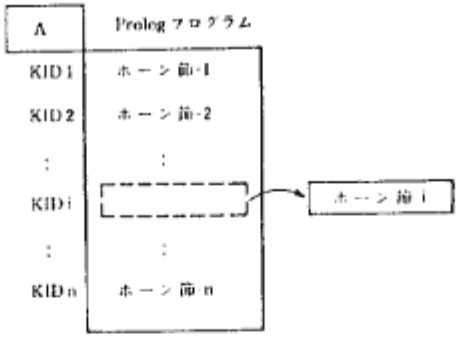


図 11 冗長性除去の方式．
プログラム A の各ホーン節に対して deduce 述語を適用する．
deduce で証明可能と判定されたホーン節は冗長である．

(2) 帰納推論アルゴリズム

従来，人工知能研究の調査文献では，帰納推論と発想推論の区別が不明確である．たとえば，『人工知能ハンドブック第III巻』の第XIV章「学習と帰納的推論」によれば，学習システムの系統的な説明のために，環境，学習部，知識ベース，および実行部から成る学習モデルを設定している[9]．

Deitterich らは，この学習モデルを用いて，次のような四種類の学習状況の分類を行なっている．

①暗記学習（Rote Learning）
学習部は，環境部から実行タスクのレベルで情報（ファクト）を，直接，蓄積している．

②言われるままの学習（Learning by being told）
学習部は，環境部から抽象的または一般的な情報（ルール）を，直接，蓄積している．

③例題からの学習（Learning by examples）

学習部は，環境部から提供される固有かつ具体的な情報（ファクト）から一般的なルールを仮説形成する．

④類推による学習（Learning by analogy）

学習部は，現在の実行タスクに対して，ある類推を発見し，かつ類似の規則を仮説形成する．

著者の見解によれば，①～②は広い意味で演繹推論，③は帰納推論，④は発想推論のカテゴリに属する．さて，Shapiroは，彼の学位論文のなかで，哲学者K.Popperの「推測と反証」を使って帰納推論を実現するために「モデル推論アルゴリズム」を提案した[34,35]．その結果に基づき試作したアルゴリズムの特徴は，ファクト型知識からルール型知識を帰納的に推論することにある．その際に，ユーザは，誤りのないファクトを与えることが要請されるので，彼のシステムは，完全教師付きの学習システムであるといえる．このアルゴリズムは，図8からもわかるように知識調節の一機能として利用することができる．図12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，後で示す．

図12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

```
Tを{□}と設定する．
Repeat
    つぎの事実を読む．
    Repeat
      While 推測Tが強すぎる do
        矛盾探索アルゴリズムを適用し，
        Tから反証を与える仮説を取り除く．
      While 推測Tが弱すぎる do
        先に反証を与えた仮説を，さらに
        精密化したものを，Tに加える．
    Until 推測Tが強すぎることもなく弱すぎることもない．
      (読み込まれた事実に関する限り)
    推測Tを答えとして出力する．
Forever
```

図12 モデル推論アルゴリズムの概略

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，推測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13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．

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14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，S_{false}，S_{true}は，それぞれ負，正の事実を蓄積する集合であり，S_{false}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．$L_0, L_1, \cdots, L_p, L_q, \cdots$は，推測の状態変化を示すために用意された記号である．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_n＝<OBSn，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_n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．

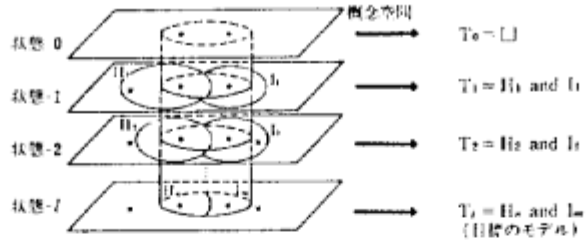


図 13 モデル推論過程における仮説の反駁と精密化.

これは，正（負）および負（偽）の事実からモデルを推論する過程で，仮説がどのように変化していくかを示すイメージ図である．

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"×"で示す．$H_1, H_2, \cdots$，および $I_1, I_2, \cdots$は，ともに仮説であり，H_n and I_m $(=T_i)$ 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，それを演繹するもっとも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(負の事実を演繹しない仮説をみつけている)，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，$T_i = H_n$ and I_m and J_k and…のようになる．

$S_{false} = \{\square\}$, $S_{true} = \{\ \}$.
$L_p = \{\square\}$ とし，□に "false" 印を付ける．
Repeat.
　次の観測データ $F_n = \langle OBS_n, TF \rangle$ を読み，OBS_n を，S_{TF} に追加する．
Repeat.
　While {S_{true} に属する OBS_i が L_p 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 OBS_i が L_p 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，OBS_i が L_n $(=L_p$ and $H_i)$ から演繹可能となるような新仮説 H_i を見つける．そして，その新仮説を，L_p に付け加える．
　Until {上記，While ループのどちらも満足しない}.
　　L_p の中で"false"印を付けていない仮説を出力する．
Forever.

図 14 モデル推論アルゴリズムの詳細

本アルゴリズムのなかで，L_p や L_n 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 14 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 Lp 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 14 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，そのために，図 15 に示すオペレータ（書き換え規則）のうちいずれかを p に適用しなければならない．

図 15 の精密化オペレータを順に適用していくと，図 16 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 $H_0 \sim H_7$ 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，$H_5 \sim H_7$ の仮説を作成できる．図中

(1) p::=□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 個の引数を持つ．

(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，辞書型知識などに登録されているユーザ定義述語または，再帰関数として定義される述語である．

図 15 精密化オペレータ.
これは，図 14 のモデル推論アルゴリズム中で使用されている．

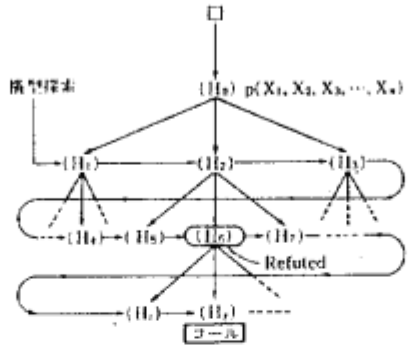


図 16 精密化グラフ.
精密化オペレータを順に適用していったときに，作成されるグラフである．ここでは，"H_6" が反駁された（Refuted）仮説とする．

では，H_7 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[18,20]．

（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15）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．

（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 の第一引数を出力引数（メンバ・リスト中のあるメンバが返される）とし，第二引数を入力引数（メンバ・リストを記述する）とする．

```
member (X, [X|Y]).
member (X, [Y|Z]):-member (X,Z).        (6-1)
```

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17に示す．図17のカッコ内に記されている数字は，図15で示した精密化オペレータのルール番号を指す．図17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 プログラムの第二行めの知識をただちに作成することができる．

7. オブジェクト・レベルの知識

ここでは，構造化知識を中心にオブジェクト・レベルの知識の推論方法につ

図 17 member(X, Y)の精密化グラフ.
＊印は高速化戦略（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．

いて述べることにする[11,18,22]．オブジェクト・レベルの知識の中でもプログラム化知識は，DEC-10 Prolog で記述された普通のプログラムとほぼ同じ意図で表現されているので，"demo (Frame, Goals)" 形式の demo 述語だけで，十分な推論機能が果せる．しかし，構造化知識についての諸概念を評価するためには，この形式の demo 述語以外に，上位—下位概念をたどる super _concept や全体-部分関係をたどる sub _concept といった述語のほかに，特性の継承関係を調べるための inherit 述語などが必要になってくる．ここで，super _concept は is _a 関係をたどる述語であり，sub _concept は part _of 関係をたどる述語である．

次に，これらの三述語の実現プログラムを図 18 に示しておく．図 18 の三述語は，いずれも第一引数に，どのフレームの構造化知識を扱っているかを示すフレーム名が記述されるようになっており，メソッド型知識であることに注意されたい．同じフレーム中の構造化知識を利用してプログラム化知識を表現するときは，しばしば，この三述語を，直接，使用することがある(付録 1 の (1-2) を参照) が，このときは，フレーム名が冗長なのでこのフレーム名の記述を省略した表現を使う．また，各述語とも，第二引数に各々の述語に対する動

```
super_concept(Frame,X,Y):-
        demo(Frame, is_a(X,Y)).
super_concept(Frame,X,Y):-
        demo(Frame, is_a(X,Z)),
        super_concept(Frame,Z,Y).

sub_concept(Frame,X,Y):-
        demo(Frame, part_of(X,Y)).
sub_concept(Frame,X,Y):-
        demo(Frame, part_of(X,Z)),
        sub_concept(Frame,Z,Y).

inherit(Frame,X,Y):-
        super_concept(Frame,X,Z),
        demo(Frame, property(Z,Y)).
inherit(Frame,X,Y):-
        sub_concept(Frame,X,Z),
        demo(Frame, property(Z,Y)).
```

図 18 概念階層関係を対象とする基本ルール

```
|?- super_concept(frame,jack,X).
X=chicken;
X=bird;
X=animal;
X=life

yes
|?- inherit(frame,jack,X).
X=canfly(_291);
X=mortal(_441)

yes
|?- demo(frame,canfly(jack)).

no
|?- demo(frame,mortal(jack)).

yes
```

図 19 super_concept と inherit 述語の実行例．

作主を記述すると，第三引数にその答が返される．以下に，super _concept と inherit 述語の実行例を，付録 1 に基づいて示しておこう．

付録 1 で示される諸知識が frame と名づけられた知識ベースのフレームに蓄積されていたとするとき，"jack は何者であるか" とか，"jack の特徴が何であるか" などといった質問は図 19 に示すようにして実行できる．図 19 の inherit 述語による問合せで jack の特性を調べたところ，「飛べる (canfly (X)) およびいつかは死ぬ (mortal (X)) という知識が特性評価のために利用できる」という結論が出されたので，次の問合せでは，"jack が真に飛べるか？" を調べ，その結果，「それが正しくない」ことを結論づけた．最後に，その次の問い合せでは，"jack がいつかは死ぬか？" を調べ，その結果，「それが正しい」ことを結論づけた．

8. メタ・レベルの知識

ここではメタ・レベルの知識の中でも，知識ベース管理にとって特に重要な，制約型知識の処理方式について述べる．この知識は無矛盾性管理，一貫性維持のために利用される．

(1) 因果関係[22]

制約型知識の中でも，4 節の (2) で述べたトリガーと呼ばれるメタ知識は，自然言語処理の分野で知られる因果関係 (causal relation)[1]の記述に利用される．因果関係は，事象間の原因と結果の関係であり知識ベースにおいては，知識ベースに更新があるたびに，それに伴って因果関係としての更新があると見做すことができる[31]．これを整理すると，図 20 に示すようになる．図 20 の因果関係 "Cause" は，原因 "Operation 2" と結果 "Operation 2" をもつ述語として表現することができる．この関係はトリガーの特殊ケースとして表現されている．

図 20 の一番目のルールは，原因から結果を知りたいとき，二番目のルールは，結果から原因を知りたいときに利用されるルールである．一番目のルール中に現れている "find _causal _operation" は，"delete"，"insert"，"update" といった知識ベースへの更新オペレーションをトリガーのゴール中から見つける述語である．ところで，順序関係 (order relation：≦) は，反射法則

```
cause(Operation 1, Operation 2) :-
        var(Operation 2),!,
        Operation 1 =.. [Access_Operator, Frame, Obeict],
        clause_of(Frame,(trigger(Mode, Operation 1) :-Goals)),
        find_causal_operation(Operation 2, Goals).
cause(Operation 1, Operation 2) :-
        Operation 2 =.. [Access_Operator, Frame, Object],
        clause_of(Frame,(trigger(Mode, Operation 1) :-Goals)),
        setof(X,find_causal_operation(X,Goals), Set),
        length(Set,1),Set=[Operation 2].
```

図 20 cause 述語のプログラム例.

```
/* causal chain*/

causal_chain(Operation 1, Operation 2) :-
        cause(Operation 1, Operation 2).
causal_chain(Operation 1, Operation 3) :-
        cause          (Operation 1, Operation 2),
        causal_chain(Operation 2, Operation 3).
```

図 21 因果チェインを表現するルール

```
|?- causal_chain(insert(frame, give
     (Agent, Receiver, Information)),X).
Receiver=_64,
Information=_91,
X=insert(frame, eat(_64,_91)),
Agent=_40;
Receiver=_64,
Information=_91,
X=insert(frame, cheerful(_64)),
Agent=_40

yes
```

図 22 causal_chain 述語の実行例

(reflexive law：$x \leqq x$)，反対称法則 (asymmetric law：$x \leqq y$ かつ $y \leqq x$ ならば $x = y$)，推移法則 (transitive law：$x \leqq y$ かつ $y \leqq z$ ならば $x \leqq z$) が成立するのに対して，因果関係 (causal relation) は順序関係の反射法則だけが成立しないといわれている[6]．推移法則は，因果関係でも成立していることから，この概念をルール化して，因果チェインをたどるルールを示すと図 21 に示すようになる．このルールは，メソッド型知識としてエキスパートまたはユーザが使うことができる．

この実行例を，付録 1 に基づいて示してみよう．えさを与える"give"という事象が生じたときに，どのような事象が生ずるのかという問合せをすると，図 22 に示すようになる．この問合せに対する答として，えさを食べる"eat"という事象と元気になる"cheerful"という事象が連鎖的に生ずることが示されている．ここでは，トリガーと呼ばれるメタ知識は，因果関係の表現に十分に利用可能であることを示していることになるが，(4-4) の起動条件として，時間に関する表現がなされたときは，タイム・トリガーとよばれるメタ知識になる．

(2) 制約型知識の実行

制約型知識の実行は，demo 述語を利用することにより，容易に実施される．トリガー型のメタ知識は，知識ベースへのオペレーションに伴い，関係する trigger 型式の知識だけをすべて実行すればよい．inconsistent 型式のメタ知識は，知識ベースの更新が妥当か否かを判定し，一つでも矛盾になる知識が存在すれば，その更新を禁止すればよい．

以上から，この制約型知識を実行するプログラム例を図 23 に示しておく．図 23 の maintenance 述語は，トリガー型の知識を選択実行するためのメンテナンス用述語であり，exec_trigger 述語は，与えられたトリガー型知識を実行する述語である．図 23 の check_inconsistency 述語は，矛盾を判定する述語である (真ならば矛盾，偽ならば無矛盾)．ここで，付録 1 の例で "is_a (chickweed, vegetable)" と "is_a (vegetable, plant_for_food)" が事前に獲得されているということを前提にして，"mary gives chickweed to jack" という事象が生じたという知識が知識ベースに挿入されているとしよう．これに伴い"eat(jack, chickweed)"という知識が挿入され，それに伴い，"cheerful (jack)" という知識が連鎖的に知識ベースに挿入されることがわかる．

```
check_inconsistency(Mode, Operation) : -
        Operation =.. [Access_Operator, Frame, Object],
        clause_of(Frame, (inconsistent(Mode, Operation) : -Goals)),
        demo(Frame, Goals).

maintenance(Mode, Operation) : -
        Operation =.. [Access_Operator, Frame, Object],
        clause_of(Frame, (trigger(Mode, Operation) : -Goals)),
        exec_trigger(Frame, Goals), fail.
maintenance(Mode, P).

exec_trigger(Frame, true) : -!.
exec_trigger(Frame, (P,Q)) : -
        exec_trigger(Frame, P), exec_trigger(Frame,Q).
exec_trigger(Frame,P) : -
        operation_type(P) -> execute(P).
exec_trigger(Frame,P) : -
        system(P) -> call(P) ;
        clause_of(Frame, (P : -Q)), exec_trigger(Frame,Q).
```

図 23 制約型知識を評価実行するプログラム例

(3) 矛盾知識の抽出

ここまでは，主に知識ベースに曖昧な知識（前定型知識）の挿入を前提とした説明になっていた（知識同化）が，その逆の立場として，エキスパートが挿入する知識に誤りがなく(これは，前提型知識である)，矛盾の原因が既存知識にある場合がある．このとき，知識ベースから矛盾知識を抽出する問題を解決しなければならない．この場合の知識獲得は，知識調節として知られている．ここでは，メタ・レベルの知識調節の基本原理に焦点を当てて，説明していくことにする．オブジェクト・レベルの知識調節の詳細については，既に著者が発表した論文を参照されたい[20,21]．

ここでは，inconsistent 形式の制約型知識とオブジェクト・レベルの知識との間に発生する矛盾解消問題を考えているので，この形式のメタ知識に矛盾する知識は，図 7 の demo 述語に仮定型 (assumption-type) 知識を収集する機構


```
demo(Frame, true, d(X,X)) : -!.
demo(Frame(P,Q), d(X,Z)) : -
        demo(Frame,P,d(X,Y)), demo(Frame,Q,d(Y,Z)).
demo(Frame, P,d(X,Y)) : -
        (system(P) ; operation_type(P)) -> eval(Frame,P), X = Y ;
        clause_of(Frame, (P : -Q), Certainty),
        (Certainty = = premise -> X = Z ;
        X = [(P : -Q)|Z]), demo(Frame,Q,d(Z,Y)).

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

図 24 三引数の demo 述語

を付加すればよいことがわかる．

図 24 は，その結果，得られた demo 述語である．制約型知識に矛盾するオブジェクト・レベルの知識を修正するとき，修正すべき仮定型の知識を取得するために，demo 述語の第三引数を，d-list 表現し，

"demo (frame, inconsistent (Operation), d (X, Y))"

の証明プロセスで，仮定型知識を取得する機能を与えている．これによると，この証明が真になれば矛盾が生じたことにより，d(X,Y) にその原因となるいくつかのホーン節が抽出される．

(4) 仮定型知識（信念）の修正

仮定型知識の修正は，知識獲得で知識体系の矛盾を解消するために実施されることになるが，8 節の（3）の方式で，ある幾つかの仮定型知識が矛盾の原因になっていることがわかると，次に示す方法で，その知識を修正できる．

(1) エキスパートが事前に，その知識の別解をあたえていたとき，その別解を新しい仮定型知識としてみる．

(2) その知識の否定をとる．

(3) その知識が例外知識であるとわかっているとき，ルールに対しては，"not" を使用した述語（付録 1 の (1-2) を参照）をゴール説に追加し，

ファクトに対しては，否定をとったファクトを登録する．

(4) エキスパートがその知識の別解を直接与える．

しかしながら，8節の（3）で抽出された仮定型知識が二つ以上存在するときは，上記方法で試行錯誤的に，修正される知識の組み合わせを捜し，無矛盾な知識ベースにしなければならない．この試行錯誤的な処理を効率的に実施するためには，少し工夫しなければならない．抽出された複数の知識の一つを順にとりだして，それを修正しその結果の無矛盾性判定を逐次実施することにより，すべての可能な組み合わせの知識修正を実施できる．これの基本部分を，アルゴリズム化すると，次のように表現できる．

抽出されたn個の知識に1から順に番号をつけ，無矛盾性判定の回数をjとすると，k番目の知識を修正すべきか否かは，次の関数f(j,k)の数値計算を実施することにより達成される．

$$f(j,k)=(j+2^{k-1}) \bmod 2^k \qquad (6\text{-}1)$$

ここでf(j,k)=0のとき，k番目の知識を修正し，f(j,k)≠0のとき，k番目の知識を修正しない．また，$1 \leqq j \leqq 2^{n-1}$が成立する．

(5) 制約型知識のトランザクション処理

ここでは，制約型知識を評価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[22]．知識ベースに対する更新は，6節で示されたように，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，知識があやまった方向に成長する事を防止するための処理が常に行なわれている．しかしながら，知識のある成長段階では，成長の過渡状態としての特殊な状況が存在することがあるので，制約型知識を過渡状態の時点で評価実行することが無意味な場合がある．これは，4節で述べたように，制約型知識の中で評価実行を遅らせるメタ知識(delayed型)が必要なことを意味している．このように，過渡状態を作り出すような1組のオペレーション群を無矛盾に評価実行するために，付録2で示されるトランザクション処理を実現しなければならない．

以下に，付録1の例にもとづき，トランザクション処理の例を図25に示してみよう．これは，構造化知識に対する知識獲得の例である．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


```
|?- 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
|?- causal_chain(insert(frame, give
     (Agent, Receiver, Information)),X).
Receiver=_64,
Information=_91,
X=insert(frame, eat(_64,_91)),
Agent=_40;
Receiver=_64,
Information=_91,
X=insert(frame, cheerful(_64)),
Agent=_40;
no
|?- transaction(insert(frame, give
     (mary, jack, chickweed))).
* End Transaction.
yes
|?- demo(frame, eat(X,Y)).
X=jack,
Y=chickweed;
no
|?- demo(frame, cheerful(X)).
X=jack;
no
```

図25 トランザクション述語の実行例

20の上から三行めのメタ知識に違反したので，この処理が失敗している．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．この例題から，トランザクション処理を知識同化や調節といったオペレーション型知識の操作にまで容易に拡張できることがわかる．

9. 実行例

ここでは，シェークスピアの「真夏の夜の夢」の登場人物を利用して，DCG (Definite Clause Grammar) で英文法の簡単なルールを獲得する問題と，メタ・レベルの知識調節問題について考えてみよう[21]．なお，DCGは，Prologで自然言語処理を行うために考案された文法である[38]．ここで示される実行例は，著者らが研究・試作した知識獲得システムを利用して実際に動かした結果であり，わかりやすさに重点を置くため，多少抽象的な説明で終っている部分もあることをお断わりしておく．本節では，三つの実行例を解説することにする．

第一番目は，オブジェクト・レベルの知識同化の例である．ここでは，正しくない英語の文法ルールと，正しい英語の文法ルールを知識同化することを考える．

第二番目は，オブジェクト・レベルの知識調節の例である．ここでは，正しい知識を受け付けるためにその文法ルールを修正することを考える．

第三番目は，メタ・レベルの知識調節の例である．新知識としての制約型知識に基づいて，オブジェクト・レベルの知識を修正することを考える．ここでは，その制約型知識に違反したすべての曖昧な事実に対するオブジェクトの知識の修正を繰り返し実施することを示す．

第一番目と第二番目の実行例については，9節の（1）で解説することにし，第三番目の実行例については，9節の（2）で解説することにする．

（1） オブジェクト・レベルの知識獲得

ここでは，問題を単純化するために，次の制限を置く．

（制限1） DCGの構文ルールs（X，[ ]）の形式は，CFG (Context Free Grammar：文脈自由文法）を対象としており，知識ベースのフレーム"dcg"には図26に示す知識がすでに存在するとする．

図26の知識は，DCGで記述されている．DCGでは，文章をリスト（単語の並び）で表現し，リストの先頭から順に単語の品詞を決定していくルール表現の形式を取っている．そのため，図中の名詞や動詞などは，構文リストの途中で決定されるので，各単語に対する品詞名の関係以外に未解析の部分の文章リストを次の解析ルールへ渡す構成をとらなければならない．たとえば，図中のhermiaという人名は，名詞nであるのでn(hermia)とすればよいように思われるが，これでは，この人名の次に"walks…"などの文章リストがきても，それを解析することができない．このように，次の文章リストを自然な形で，順次，解析できるようにするためには，n([hermia|X]，X）と表現すれば無理なく実現できる．すなわち，この表現では，hermiaは名詞nであり，その名詞の次にくる文章リストをXとして，次の文章解析に委ねることができる．

```
（1） 制約型知識

inconsistent(immediate_) :- s(X, [ ]), exist_variable
(X).
文法ルールによって生成される任意の文は，
    例示されない変数を含まない．
（2） プログラム化知識1 （単語辞書）
n([hermia|X],X).                          名詞
n([lysander|X],X).                        名詞
n([forest|X],X).                          名詞
vi([walks|X],X).                          自動詞
vt([loves|X ],X).                         他動詞
p([in|X], X).                             前置詞
（3） プログラム化知識2 （文法ルール）
np(X,Y) :- n(X,Y).                        名詞句
vp(X,Y) :- vi(X,Y).                       動詞句
vp(X,Y) :- vt(X,Z),n(Z,Y).                動詞句
pp(X,Y) :- P(X,Z),n(Z,Y).                 前置詞句
```

図26 オブジェクト・レベルの知識獲得に対する実行環境．フレーム名はdcgである．

文法ルールは，動詞句や前置詞といったまとまった形の概念で表現できることから，木の構造になっていることが知られている．このような句表現としてのルールも，単語の解析と同じ考え方でルール表現できる．

（制限2） 構文ルールs（X，[ ]）から演繹される文章は，すべて既知知識群（単語およびルール）から構成されるとする．そのために，"dcg" に対する更新の禁止側（制約型知識）として，図26の（1）の示される制約型知識を与えるものとする．

(1·a) 知識同化

ここでは，一番目の実行トレースの解説を行うことにする．図27は文法ルールの同化を表わしている．この図は，実行時の画面状態を示している．同化は，assimilationとも呼ばれ，曖昧な知識（assumption-type knowledge）を知識ベースに無矛盾に追加することを意味する．この文法ルールは，次のことがらを示している．それは，もしXからUまでの単語列が名詞であり，UからWまでの単語列が動詞句であれば，XからYまでの単語列は文章として認識できるということである．正しい文法ルールでは，Wは，Yに等しくなければならないことに注意すべきである．これは，明らかに正しくない文法ルールなので制約型知識により拒否されている．

```
% ?- assimilate(dcg,(s(X,Y):-np(X,U),vp(U,W))).

* The Input_Knowledge "s(X,Y):-np(X,U),vp(U,W)"
 is inconsistent with Integrity Constraint(s)
 "inconsistent(immediate,X):-s(Y,[ ]), exist_variable(Y)".
```

Dictionary：

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules：

```
np(X,Y):-n(X,Y)
vp(X,Y):-vi(X,Y)
vp(X,Y):-vt(X,U),n(U,Y)
pp(X,Y):-p(X,U),n(U,Y)
```

Constraints：inconsistent(immediate,X):- s(Y,[ ]),exist_variable(Y)

図27 実行トレース1


```
% ?- assimilate(dcg,(s(X,Y):-np(X,U),vp(U,Y))).

* Eliminate Redundancy ? (y/n) n.

* The Knowledge "s(X,Y):-np(X,U),vp(U,Y)" was acquired.

yes
% ?-
```

Dictionary：

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules：

```
np(X,Y):-n(X,Y)
vp(X,Y):-vi(X,Y)
vp(X,Y):-vt(X,U),n(U,Y)
pp(X,Y):-p(X,U),n(U,Y)
s(X,Y):-np(X,U),vp(U,Y)
```

Constraints：inconsistent(immediate,X):- s(Y,[ ]),exist_variable(Y)

図28 実行トレース2

Z2829

```
% ?- retrieve(dcg,s(X,[ ]),X).

[forest,loves,forest]
[forest,loves,hermia]
[forest,loves,lysander]
[forest,walks]
[hermia,loves,forest]
[hermia,loves,hermia]
```

Dictionary：

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules：

```
np(X,Y):-n(X,Y)
vp(X,Y):-vi(X,Y)
vp(X,Y):-vt(X,U),n(U,Y)
pp(X,Y):-p(X,U),n(U,Y)
s(X,Y):-np(X,U),vp(U,Y)
```

Constraints：inconsistent(immediate,X):- s(Y,[ ]),exist_variable(Y)

図29 実行トレース3

```
[lysander,loves,forest]
[lysander,loves,hermia]
[lysander,loves,lysander]
[lysander,walks]

yes
```

Dictionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U),n(U,Y)
pp(X,Y) : -p(X,U),n(U,Y)
s(X,Y) : -np(X,U),vp(U,Y)
```

Constraints : inconsistent(immediate,X) : - s(Y,[ ]),exist_variable(Y)

図 30 実行トレース 4

```
yes
% ?- accommodate(dcg,s([hermia,walks,in,forest],[ ])).
Checking fact(s)...
Error : missing solution s([hermia,walks,in,forest],[ ]).diagno

Query : vp([walks,in,forest],[ ])? y.

Query : vi([walks,in,forest],[ ])? n.

Query : vt([walks,in,forest],X)? n.
Error diagnosed : vp([walks,in,forest],[ ]) is uncoverd.

— Trying to accommodate : vp([walks,in,forest],[ ])
```

Dictionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U),n(U,Y)
```

Constraints : inconsistent(immediate,X) : - s(Y,[ ]),exist_variable(Y)

図 31 実行トレース 5


```
Refining : vp(X,Y) : -true
  Checking : vp(X,[ ]) : -true
  Found clause vp(X,[ ]) : -true
     after searching 4 clauses.
Listing of vp(X,Y).
Checking fact(s)...
Error : missing solution s([hermia,walks,in,forest],[ ]). diagno
Error diagn_
```

Dictionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U),n(U,Y)
pp(X,Y) : -p(X,U),n(U,Y)
s(X,Y) : -np(X,U),vp(U,Y)
```

Constraints : inconsistent(immediate,X) : - s(Y,[ ]),exist_variable(Y)

図 32 実行トレース 6

```
vp(X,Y) : -vt(X,U),n(U,Y).
vp(X,[ ]).

The Knowledge "vp(X,[ ]) : -true"
is inconsistent with the Integrity Constraint(s)
"inconsistent(immediate,X) : -s(Y,[ ]),exist_variable(Y)".
```

Dictionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U),n(U,Y)
pp(X,Y) : -p(X,U),n(U,Y)
s(X,Y) : -np(X,U),vp(U,Y)
```

Constraints : inconsistent(immediate,X) : - s(Y,[ ]),exist_variable(Y)

図 33 実行トレース 7

```
Refining : vp([X|Y],[ ]) : -true
  Checking : vp([X,Y|Z],[ ]) : -true
  Refuted : vp([X,Y|Z],[ ]) : -true

Refining : vp([X|Y],Z) : -pp(Y,V)
  Checking : vp([X|Y],Z) : -pp(Y,Z)
  Found clause vp([X|Y],Z) : -pp(Y,Z)
      after searching 16 clauses.

Listing of vp(X,Y) :
    vp(X,Y) : -vi(X,Y).
    vp(X,Y) : -vt(X,U),n(U,Y).
    vp([X|Y],Z) : -pp(Y,Z).
```

Dictionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U),n(U,Y)
pp(X,Y) : -p(X,U),n(U,Y)
```

Constraints : inconsistent(immediate,X) : - s(Y,[ ]),exist_variable(Y)

図 34 実行トレース 8

```
    Found clause vp(X,Y) : -vi(X,U), p(U,Y).
        after searching 10 clauses.
Listing of vp(X,Y) :
    vp(X,Y) : -vi(X,Y).
    vp(X,Y) : -vt(X,U), n(U,Y).
    vp(X,Y) : -vi(X,U), pp(U,Y).

Checking fact(s)...no error found.


* The Knowledge "vp" was accommodated from the Input Fact(s).
yes
% ?- _
```

Dictionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U), n(U,Y)
pp(X,Y) : -p(X,U), n(U,Y)
s(X,Y) : -np(X,U), vp(U,Y)
vp(X,Y) : -vi(X,U), pp(U,Y)
```

図 35 実行トレース 9


```
% ?- retrieve(dcg,s(X,[ ]),X).

[forest,loves,forest]
[forest,loves,hermia]
[forest,loves,lysander]
[forest,walks]
[forest,walks,in,forest]
```

Dectionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U),n(U,Y)
pp(X,Y) : -p(X,U),n(U,Y)
s(X,Y) : -np(X,U),vp(U,Y)
vp(X,Y) : -vi(X,U),pp(U,Y)
```

Constraints : inconsistent(immediate,X) : - s(Y,[ ]),exist_variable(Y)

図 36 実行トレース 10

```
[lysander,loves,lysander]
[lysander,walks]
[lysander,walks,in,forest]
[lysander,walks,in,hermia]
[lysander,walks,in,lysander]

yes
% ?- _
```

Dictionary :

```
n([hermia|X],X)
n([lysander|X],X)
n([forest|X],X)
vi([walks|X],X)
vt([loves|X],X)
p([in|X],X)
```

Rules :

```
np(X,Y) : -n(X,Y)
vp(X,Y) : -vi(X,Y)
vp(X,Y) : -vt(X,U),n(U,Y)
pp(X,Y) : -p(X,U),n(U,Y)
s(X,Y) : -np(X,U),vp(U,Y)
vp(X,Y) : -vi(X,U),pp(U,Y)
```

Constraints : inconsistent(immediate,X) : - s(Y,[ ]),exist_variable(Y)

図 37 実行トレース 11

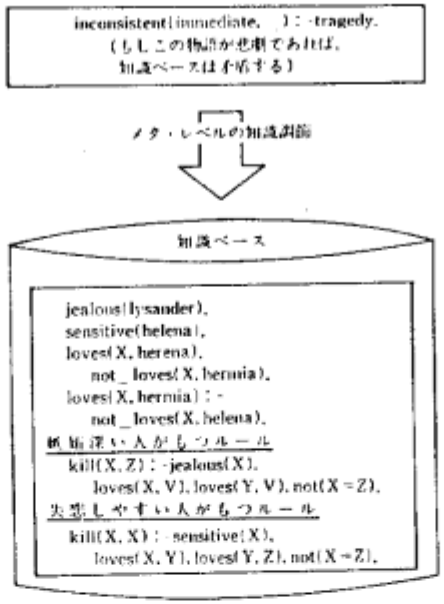


図 38 メタ・レベルの知識獲得に関する実行環境
フレーム名は, story である.

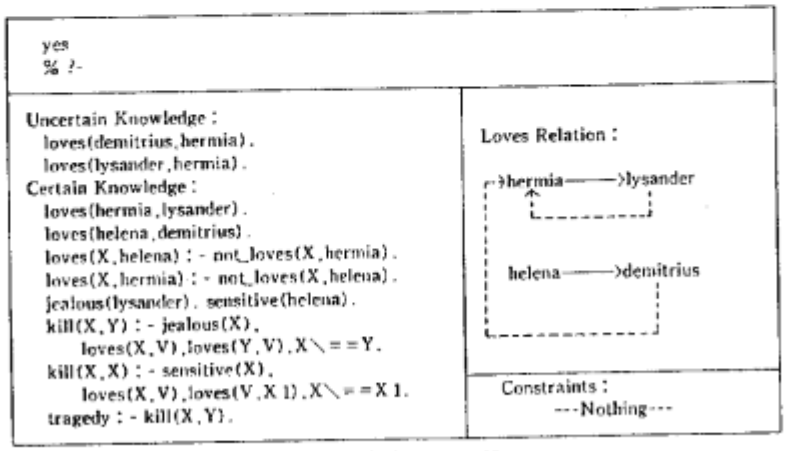


図 39 実行トレース 12

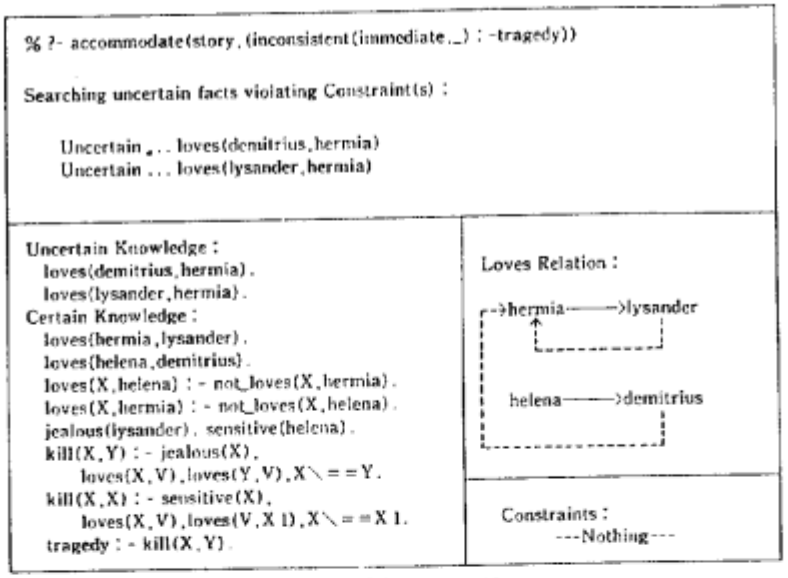


図 40 実行トレース 13

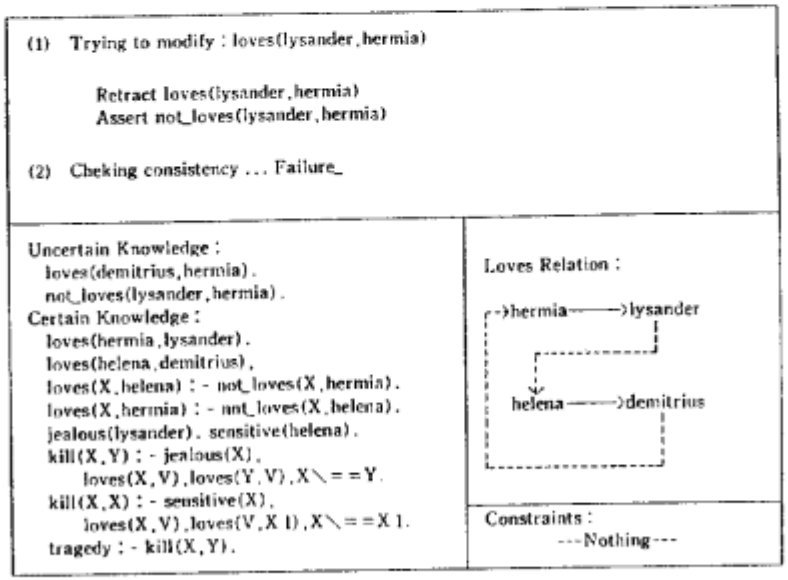


図 41 実行トレース 14

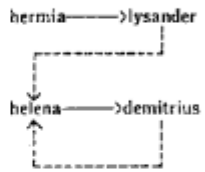
```
(1)  Trying to modify : (loves(demitrius,hermia)

        Retract loves(demitrius,hermia)
        Assert not_loves(demitrius,hermia)

(2)  Cheking consistency ... Failure_
```

```
Uncertain Knowledge :
  not_loves(demitrius,hermia).
  not_loves(lysander,hermia).
Cartain Knowledge :
  loves(hermia,lysander).
  loves(helena,demitrius).
  loves(X,helena) :- not_loves(X,hermia).
  loves(X,hermia) :- not_loves(X,helena).
  jealous(lysander). sensitive(helena).
  kill(X,Y) :- jealous(X),
       loves(X,V),loves(Y,V),X\==Y.
  kill(X,X) :- sensitive(X),
       loves(X,V),loves(V,X 1),X\==X 1
  tragedy :- kill(X,Y).
```

Loves Relation :



Constraints :
---Nothing---

図 42 実行トレース 15

```
        Assert loves(lysander,hermia)

(2)  Cheking consistency ... Success

Updating the Frame in the Knowledge Base.

            ---End Processing---
```

```
Uncertain Knowledge :
  not_loves(demitrius
  loves(lysander,herm
Certain Knowledge :
  loves(hermia,lysander).
  loves(helena,demitrius).
  loves(X,helena) :- not_loves(X,
  loves(X,hermia) :- not_loves(X,
  jealous(lysander).
  kill(X,Y) :- jealous
       loves(X,V),loves(Y,V),X 1=
  kill(X,X) :- sensitive(X),
       loves(X,V),loves(V,X 1),
       X 1=
  tragedy :- kill(X,Y)
```

Loves Relation :

Constraints :

```
inconsistent(immediate,_) :-
        tragedy.
```

図 43 実行トレース 16

つぎに，図 28 で示されるように，正しい文法ルールが同化されている．このときは，近似的な文法ルールが首尾よく同化されている．

(1-b) 知識調節

ここでは，第二番目の実行トレースの解説をする．この実行例では，正しい文章を与えることにより，文法ルールを調節することを考える．調節は，accomodation とも呼ばれ，確かな知識（premise-type knowledge）を知識ベースに追加することによって既存の知識を無矛盾に修正することを意味する．ここで与える正しい文章は，調節する以前の文法ルールが演繹できなかった文章である．この処理過程は，文法推論として知られている．

図 29 と 30 に，既存の辞書と文法ルールから組み立てられるすべての文章（例）を導いてそれらを表示している．

この表示された多数の文章には，前置詞句をもつ文章が一つもないので，これらの文法ルールには，その文章を推論する能力がないことがわかる．これらの文法ルールにその推論能力をもたせるためには，その文法ルールを修正する必要がある．そこで，図 31 に示す調節コマンドを利用して『hermia walks in forest』という文章を入力している．

この文章を認識するルールがこのフレーム内に全然ないので，"missing solution" からはじまるメッセージが示されている．このとき，このシステムは，『walks in forest』が動詞句か否かをユーザにきいてくる．この答えは，明らかに "yes" であるので，記号 "y" を入力している．さらに，システムはこの動詞句が自動詞か他動詞かをきいている．この答えは，明らかに，どちらも "no" であるので，両方に "n" を入力している．この質問応答を通して，このシステムは，動詞句として『walks in forest』を解釈している文法ルールが誤っているということを図 31 で検出している．そのため，この誤ったルールの調節を，実施していくことになる．

ここで，文法ルールをいかに推論するかということをもう一度，簡単に説明してみることにする．概略的にいうと，モデル推論システムは，最も一般的なモデルから出発し，段階的にそのモデルを精密化していくことにより，観測事実を説明するモデルを推論する．

その例として，各国の観光地で，日本の旅行者を識別するための問題について考えてみることにする．最初に思いつく仮説は，"もし，その旅行者が，日

本のカメラを持っていれば，その人は，日本人である" という仮説である．われわれは，各国の観光地で，日本のカメラを持っているアメリカ人，英国人，その他の国の人を見かけることができる．実際に日本人の旅行者をさらによく観察すると，まえに述べた仮説を精密化し，第二番目の仮説として，"旅行者が，日本のカメラを持つと同時にメガネをかけていれば，その人は，日本人である" という仮説を立てることができる．この仮説は，最初に立てた仮説よりも，さらにもっともらしい（精密な）仮説であることがわかる．

さて，本題にもどり，図32の端末に表示されているメッセージを説明していくことにしよう．

"Refining:" の後のルールは，精密化される仮説であり，"Checking:" の後のルールは，精密化された仮説であるので，チェック済みの仮説になっている．もし，この判定が，失敗に終われば，"Refuted:" というメッセージが，表示される．いくつかの判定のあとに，仮説の精密化が達成され，その結果が "Found clause:" というメッセージの後に表示されている．この新しく見つけられた仮説は，特殊な場合として，与えられた例を含む一般解になっている．われわれの例では，最初に "X と Y の動詞句が true である" というルールが精密化されたルールとして選択され，"X と nil の動詞句が true である" という最初の精密化の結果が判定されている．この判定結果は，成功しているので，画面上に "Found clause" というメッセージが表示されている．しかしながら，このルールは，動詞句として任意の入力Xを受け付けることができるので，このルールは，図33に示されているように，制約型知識に違反していることがわかる．これにより，図34で示されるように，さらに同じ過程を繰り返し，ルールの右側に前置詞句だけをもつ動詞句を認識するルールを見つけることになる．このルールは，再び，制約型知識に違反することになる．

最後に，図35で，右側の自動詞と前置詞句をもつルールを見つけることになる．このルールは，制約型知識を満足しており，画面上に示されているようなルールの集合が調節結果ということになる．

新しい文法ルールで組み立てられた文章の集合が，図36と37に示されている．これで，この文法ルールが，期待される文章を推論する能力があるということで，エキスパートとシステムの間の会話を終了することになる．

(2) メタ・レベルの知識獲得

最後に，第三番目の実行トレースの解説をしてみよう．この実行では，『真夏の夜の夢』で知られるウイリアム・シェクスピア作の物語を簡単に模擬することにする[4,7]．この状況は図38に表示されている通りである．二種類のルールとファクトが知識として示されている．それらの知識のなかには，不確かな知識と確かな知識の二種類がある．この恋愛関係の図式は，図39に示されている．ここで，実線は，確かな知識を示しており，点線は不確かな知識を示している．そのため，この例では，"Hermia loves Lysander" は true であるが，"Lysander loves Hermia" は，true ではないかも知れないのである．さらに，図39の下に，嫉妬をもつ人と失恋しやすい人について，それぞれ "kill" というルールが示されており，"kill" が悲劇 "tragedy" を起こすというルールが下に示されている．

この調節過程は，図40で示されるように，制約型知識として『悲劇は，この物語を矛盾にする』というメタ知識を与えることによって始めることになる．このシステムは，最初にこの制約型知識に違反するすべての不確かな知識をみつけようとしている．ここでは，二つの不確かな知識が，違反しているということがわかる．このとき，このシステムは，この二つの知識 "lysander loves hermia" を取り出し，それを修正する．この修正は，"loves" を "not_loves" あるいは，逆もまた同様に修正することで達成される．"loves" と "not_loves" の間の関係から，この修正は，パートナーの変更を恋愛関係にもとめていることになる．この場合，修正知識として，"lysander loves hermia" という新しい不確かな知識を得ることになる．それは，図41に示されている．この正当性は，この次の矛盾判定という段階で判定される．すなわち，次では，システムは，この新しい関係に矛盾が全然ないか否かを判定することになる．この結果は，失敗（矛盾）することになっている．その結果，システムは，同じような修正過程を繰り返すことになる．次の修正は，その他の知識に対して実施される．すなわち，それは，図42に示されるように，"demitrius loves hermia" に対して実施されることになる．この結果は，また，失敗（矛盾）している．このとき，このシステムは，最後の可能性を試みることになる．すなわち，それは "最初の修正をもとにもどす" ことである．

最後に，図43に示されるように，このシステムは，任意の矛盾を避けるこ

とに成功したことになる．以上の手続きから，図43に示されている知識ベースの“story”フレームが更新される．

おわりに

本稿では，知識獲得機能に重点をおいて，prologによる知識ベース管理の基本技術を述べ，次のような結論を得た．

（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，この関係に関する諸性質を系統的にルール化した．

（2） メタ推論を利用して制約用知識を容易に評価実行できた．その中でも，著者が考案したtrigger述語は，自然言語処理でよく使われる因果関係の表現を包含している．

（3） 制約用知識の評価実行に種々の可能性を設けるために，評価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した．

（4） 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，矛盾知識の抽出法とその効率的な修正法を明確にした．さらに，知識ベース管理システムとしての研究として，著者，独自のフレーム構成の有用性を示した．

（5） 知識獲得の概念を明確にし，著者，独自の知識獲得機能のダイヤグラムを確立した．これは，他に例をみていない．

（6） メタ推論機能は，多くの適用分野をもち得る事を例をもって示した．

（7） メタ推論により知識の同化や調節といったメカニズムを実現した．

知識同化プログラムや知識調節プログラムの実現の過程で，demo述語によるメタ・プログラミング技法が生まれた．

メタ・プログラミング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，知識プログラミング言語の設計，Prologと関係データベースとのインターフェース機構の実現，等においても，その有用性が確認されつつある．しかも並列型Prologでのメタ・プログラミングの適用例も，最近急激に増えつつある．

メタ・プログラミング技法の唯一の欠点は，インタプリーティブに走る所が多いので，処理性能が遅いことである．しかしながら最近，部分計算法（partial evaluation method）によるメタ推論システムのコンパイル化技術を開発し，二〜三の例題でその有用性を確かめた[28,36]．論理ベースのエキスパート・システム，ボトムアップパーザ，トレースを行うPrologインタプリタ等に部分計算プログラムを適用し，メタ推論の高速化に成功した．この方法を適用すれば多くのメタ推論システムの最適化が行える．

さらに，発想推論の問題も見逃せない問題である．発想とは，ある驚くべき事実に遭遇した人間が，その事実を説明する仮説を直観的に得る人間の思考過程である．前述したように帰納推論は，ある特定の概念についてのモデル推論であり，仮説生成メカニズムが重要なメカニズムになっている．しかしながら，本当に発想推論と呼べるものは，既存の知識群に関する知識（ファクトとルール）が不十分なので，他の既知の知識群との間になんらかの翻訳逆翻訳（創造工学の用語では等価変換[4]と呼んでいる）を行いつつ，その事実を説明する仮説を生成する．既知の知識群と未知の知識群との等価変換的写像関係の発見には，部分同型に基づく類推による思考形式が活用されている．部分同型そのものを発見するのに，連想に基づくパターン・マッチング機構は，一つの有効な方法である[13]．発想的推論については，研究すべきことがあまりにも多くあるので，今後の進展を期待したい．

最後に本稿で述べたことはPrologによる知識獲得向きの知識ベース管理問題のうちで，割合に論理的に性質のよいものについて，その解決の指針を述べたに過ぎないことを強調しておく．実際，この管理問題は，より難しくより複雑なたくさんの問題をかかえている．

付録1． フレーム内の知識例

ここでは，フレーム内に存在する種々のカテゴリーの知識のうち，構造化知識，プログラム化知識，制約型知識を例示する．例としては，生物（life），動物（animal），植物（plant），哺乳類（mammal），鳥（bird），果物（fruit）などを対象にする．メソッド型知識，辞書型知識に関しては，本文中で例示されている．なおここでは簡単のため，“KID”，フレーム名で，確信の程度，についての扱いを省略する．

(1-1) 構造化知識 (Structuralized Knowledge)

ここで扱う例に関して，is_a，property関係を図44に図示する．構造化知識とし

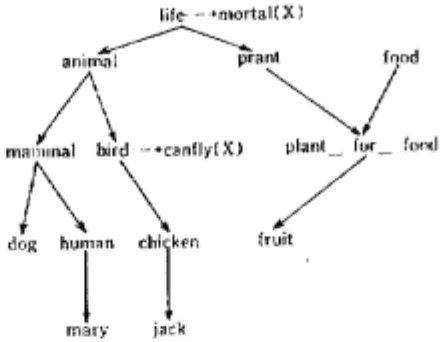


図 44 is_a, property関係の構造
矢印⟶はis_a関係を示し、矢印⋯→はproperty関係を示す

```
/* Structuralized Knowledge */
is_a(animal,life).
is_a(plant,life).
property(life,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 45 構造化知識の知識表現例


```
/* Programmed Knowledge */
mortal(X) :-
        super_concept(X,life).

not_canfly(jack).
canfly(X) :-
        super_concept(X,bird),
        /+not_canfly(X).

hungry(jack).

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

図 46 プログラム化知識の知識表現例

```
/* Constraints */
trigger(immediate,insert(frame,give(Agent,Receiver,Information))) :-
        food(Information),hungry(Receiver),!,
        insert(frame,eat(Reciever,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,life).
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 47 制約型知識の知識表現例

て表現された知識は，図45に示されている．

図45で，"mary"，"jack" の項は，個体を表わすが，その他の項は，概念を表わしている．

(1-2) プログラム化知識 (Programmed Knowledge)

ここでは，初期状態として，"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1例である (super_concept (jack, bird))" などの知識が，図46に示されている．

(1-3) 制約型知識 (Constraint-type Knowledge)

```
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).
maintain_constraints(Mode,true) : -!.
maintain_constraints(Mode,(P,Q)) : -
        maintain_constraints(Mode,P),maintain_constraints(Mode,Q).
maintain_constraints(Mode,P) : -
        operation_type(P),!,
        \+check_inconsistency(Mode,P),maintenance(Mode,P).
maintain_constraints(Mode,P) : -
        system(P)->true ;
        clause(P,Q),maintain_constraints(Mode,Q).
```

図 48 トランザクションを実行するプログラム例

図 47 にこの知識の知識表現例を示す．五つの知識のうち，最初の二つの知識はトリガーと呼ばれるメタ知識であり，知識ベースの更新があるたびに知識ベースの一貫性を保持するために，知識ベースをメンテナンスするメタ知識である．最後の三つの知識は，更新により変換する知識ベースが，ユーザの意図に反する方向に変換しないように，更新オペレーションそのものに対して制約を与えている．

付録 2. トランザクションの実行プログラム例

図 48 にトランザクションの実行プログラムの中心部分を示してあるが，lock 述語と unlock 述語の詳細については，ここでは重要でないので，無視して考えてもらいたい．

参 考 文 献

1) J.F.Allen (1983) Recognizing Intention from Natural Language Utterances, in M.Brady et al (ed.) Computational Model of Discourse, MIT Press.
2) 有川節夫 (1984) 帰納推論と類推の応用，『昭和 58 年度先端的コンピュータに関する調査研究報告書『技術動向編』-II』日機連および ICOT 編，pp.22-33.
3) D.L.Bowen (1981) DECsystem-10 PROLOG USER's Manual, DAI, University of Edinburgh.
4) K.A.Bowen and R.A.Kowalski (1981) Amalgamation Language and Meta-Language in Logic Programming, TR 4/81, Syracuse University.
5) P.R.Cohen and E.A.Feigenbaum 著 (1984) 田中幸吉，淵一博 共訳，『人工知能ハンドブック第三巻』，共立出版
6) J.Doyle (1978) Truth Maintenance System for Problem Solving, MIT Technical Report AI-TR-419.
7) 同上 (1979) Truth Maintenance System, Artificial Intelligence, Vol.12, No.3.
8) Mario Bunge (1959) Causality the Place of the Causal Principle in Modern Science, The President & Fellows of Harvard College, 日本語訳では岩波書店，黒崎宏訳，『因果性』(1972) をみよ．
9) D.D.Chamberlin et al. (1976) SEQUEL 2 : A Unified Approach to Definition, Manipulation, and Control, IBM J. RES. DEVELOP.
10) 淵一博 (1978) 問題解決と推論，『情報処理』，Vol.19, No.10, pp.936-943.
11) K.Furukawa, A.Takeuchi, and S.Kunifuji (1983) Mandala : A Knowledge Representation System in Concurrent Prolog, Information Society of Japan, Preprints of WG on Knowledge Engineering and Artificial Intelligence.
12) 波多野誼余夫 (1982)『認知心理学講座』，第 4 巻，東京大学出版会．
13) M.Haraguchi (1984) An Analogy as a Partial Identity, "Proc. of the Logic Programming Conference '84, 11-2, ICOT, Mar.
14) 市川亀久彌 (1970)『創造性の科学』，日本放送出版協会．
15) 北上 始，麻生隆敏，國藤 進，宮地泰造，古川康一 (1983) 知識同化構造の一実現法，『情報処理学会知識工学と人工知能研究会資料 30-2』．
16) Kitakami, H., Kunifuji, S., Miyachi, T., and Furukawa, K. (1984) 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, also available from ICOT as ICOT Technical Report TR-037 (1984).
17) 北上 始，國藤 進，宮地泰造，古川康一 (1984) Kaiser/KIM Architecture, ICOT Technical Report TM-0063.
18) 北上 始，國藤 進，宮地泰造，古川康一(1984)大規模な知識ベース管理システムをめざして，『情報処理知識工学と人工知能研究会』．
19) 北上 始，宮地泰造，古川康一，國藤 進 (1984) 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．
20) 北上 始，國藤 進，宮地泰造，古川康一 (1984)『特集 ロジックプログラミング：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』，日経エレクトロニクス，11. 5 号．
21) H.Kitakami, S.Kunifuji, T. Miyachi, K.Furukawa, A.Takeuchi, T.Miyazaki, S.Ishii, T.Takewaki, M.Ohki (1984) Demonstration of the KAISER System at the ICOT Open House in FGCS '84, ICOT TR-0081.
22) 北上 始，國藤 進，宮地泰造，古川康一 (1985) 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，『情報処理学会誌投稿』11月．
23) 北川敏男 (1968)『統計学の認識』，白揚社．
24) 同上 (1983) 知識情報処理システムへの情報処理学的接近．『富士通（株）国際情報社会科学研究所研究報告第10号』，3月．
25) S.Kunifuji and H.Yokota (1982) PROLOG and Relation Data Bases for Fifth Generation Computer Systems, Proc. of CERT Workshop on "Logic base for Databases".
26) 関藤 進，森生盛敏，竹内彰一，坂井 公，宮地泰造，北上 始，横田治夫，安川秀樹，古川康一 (1983) 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』．
27) 関藤 進，北上 始，宮地泰造，古川康一 (1985) 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．
28) 関藤 進，竹内彰一，北上 始，宮地泰造，大木 優，武脇敏晃，古川康一 (1985) 論理プログラミングと知識情報処理—メタプログラミングによるアプローチ—，『情報処理学会シンポジウム』，9月．
29) M.Minsky (1975) Framework for Representing Knowledge, in Winston (ed.) : The Psychology of Computer Vision, McGraw-Hill.
30) T.Miyachi, S.Kunifuji, H.Kitakami, K.Furukawa (1984) A Knowledge Assimilation Method for Logic Databases, New Generation Computing, 2-4, 385/404.
31) 宮地泰造，関藤 進，古川康一，北上 始 (1984) 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．
32) J.Ong, D.Fogg, and M.Stonebraker (1984) Implementation fo Data Abstraction in the Relational Database System INGRES, ACM SIGMOD RECORD.
33) K.R.Popper (1968) Conjecture and Refutations, Herper Torchbooks, New York.
34) E.Y.Shapiro (1981) Inductive Inference of Theories From Facts, Yale University Research Report 192.
35) E.Y.Shapiro (1982) Algorithmic Program Debugging, An ACM Distinguished Dessertation 1982. The MIT Press.
36) 竹内彰一，近藤浩康，大木 優，古川康一(1985)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』．
37) 田中 謙他 (1983) 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回自動制御学会知識工学シンポジウム資料』．
38) 田中穂積 (1984) Prologによる構文解析，『Computer Today』，No.1，pp.40-47，5月．
39) W.Weyhrauch (1980) Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, 13, 13/170.
40) H.Yokota, S.Kunifuji, T.Kakuta, N.Miyazaki, S.Shibayama, and K.Murakami (1982) An Enhanced Inference Mechanism for Generating Relational Algebra Queries, Proc. of Third ACM SIGACT-SIGMOD Symposium on Principles of Database System, Waterloo, Canada, April 2-4.
41) 米盛裕二 (1981)『パースの記号学』勁草書房．

（(株)富士通研究所）