解　説

論理型計算モデル†

後藤滋樹††　古川康一†††

1. はじめに——論理とプログラムとは同じものか

良くプログラミングの入門書などに，論理とプログラムとは似ているという記述が見られる．いわく，プログラムを書くということは数学の定理を証明するのと同じであり，一歩一歩論理的に積上げなければならない，というわけである．このような見方は，基本的には正しいと言える．確かに論理とプログラムとは雰囲気が似ているという以上に，部分的には全く同じものだと言って良い．

しかしながら，我々が通常の意味で論理と呼んでいるものがすべてプログラムの中に具現化されているかというと，例外は枚挙に暇がない．逆に，プログラムの一挙一動がことごとく論理的にうまく説明できるかと言えば，我々はまだそのような理想的な論理体系を手に入れてはいない．

結論めいたことを言えば，論理とプログラムとは遠くから見れば大体同じものであるが，近付いて見ると細部においてはかなり異なっていると言わざるを得ない．本稿はまさにその類似性と差異とについて解説を試みるものである．

論理型計算モデルというテーマに関連した話題として，現在PROLOGという新しい言語が注目されている．PROLOGはプログラミング言語でありながら，その構成要素が一種の論理式の形をしている．このような特徴は本稿に格好の話題を提供するものであるから次の2章ではまずPROLOGがどのように動くか，という点を解説し，引続いて3章でプログラムと論理との関係をより一般的に論じることにする．

2. 論理式に基づくプログラミング言語

以下ではプログラミング言語PROLOGの動作の仕組みを解説する．ただし，ここでは論理とプログラムとの対比をとることを主目的にしているので，PROLOGの基本的なメカニズムの紹介に重点を置く*．

なお2.1節ではいささか細かい定義が導入されるが，ここで定義された用語や記号は後の節でも引続いて用いられる．

2.1 項・述語・素論理式

軽い意味で「論理」という時には，そこで使われる記号や概念について特に取り決めはしない．しかし，論理体系というのは，それこそ無数に存在しているのであるから，多少とも正確に話を進めようとするならば，まず形式的な取り決めをしてから話を始めるのが無難である．

さて，我々が普通に論理と呼んでいるのは，正確に言うと一階述語論理ということになる**．述語論理では論理式（英語ではwell formed formula）という表現形式が用いられるが，これはほぼ日常語の文章を記号化したものである．例えば，

【日常語】 1+2は3に等しい．

は次のような論理式に対応づけられる．

【論理式】 equal (1+2, 3)

述語論理の「**述語**」(predicate) というのは日常語の方では「等しい」の部分を言い，論理式では“equal”の部分を指す．この対応は良くとれている．日常語では述語の他に主語とか修飾語とかいう分類も行われているが，論理式では，述語の他はひとまとめに**項**(term) として扱われている．

［例］　equal (1+2, 3)
　　述語　項　項

項はさらに，定数，変数，関数から構成される．

［定数の例］ 1, 2, 3, …

［変数の例］ $x, y, z, \cdots$

定数あるいは変数は，それ自身が単独で項となる．また関数を用いてより複雑な項を作ることもできる．

［関数を使った項］ $1+2, 3+y, z \div w, \cdots$

† Computation Models for Logic Programming Languages by Shigeki GOTO (Yokosuka Electrical Communication Laboratory, N.T.T.) and Koichi Furukawa (Institute for New Generation Computer Technology).

†† 日本電信電話公社横須賀電気通信研究所
††† 新世代コンピュータ技術開発機構

* 現実にインプリメントされたPROLOGの処理系では論理の世界をはみ出すメカニズムも種々組込まれている．

** 「一階」という言葉についてはここでは無視して良い．

ここに，＋，÷は関数記号である．関数を使った項は次のようにも書くことができる．

$\underset{\text{関数}}{+}\ (\underset{\text{項}}{1}, \underset{\text{項}}{2}),\qquad \underset{\text{関数}}{\div}\ (\underset{\text{項}}{x}, \underset{\text{項}}{w})$

このように関数記号を先頭に出した書き方を prefix 形式という．また 1+2 の＋のように項の間に書くのを infix 形式という．実は，関数と述語とはほぼ同類のものと考えられるので，述語でも infix 形式で書かれるものもある．

［例］ $\underset{\text{項}}{1+2} = \underset{\text{項}}{3}$　　ここに＝は等号の意味．
述語

＋，－，×，÷，＝のように馴染み深い記号は良く infix 形式で使われるが，infix 形式は2項間に書く場合に限られるので，prefix 形式のほうが一般的である．（2項以上の項を持つ述語・関数もいろいろある．）

さて，論理式に話を戻そう．先の例，

equal (1+2, 3)

のような論理式は，論理式全体のうちでは単純な部類に属するもので，**素論理式** (atomic formula) と呼ばれる．一般の論理式は，上のような素論理式に ∧(AND), ∨ (OR), ¬ (NOT) などの論理記号 (connective) を組合せて構成される*．言い換えれば，素論理式とは論理記号をひとつも含まない論理式のことである．

2.2 プログラミング言語 PROLOG

ここでは，簡単なプログラムの実例を通じて，新しいプログラミング言語 PROLOG の紹介をする．PROLOG の提案は Colmerauer の論文[1]により行われ，Kowalski の論文[2]により論理的な裏付けがなされた．

従来の PROLOG の解説書では文献[2]に由来する論理式の手続き的 (procedural) 解釈をとるものが多かったが，ここでは Colmerauer[3] を参考にして一種の「書き換え規則」として眺めてみたい．

簡単なプログラムの例を作ろう**．

［例］ PROLOG による加算プログラム

① add $(0, y, y)$:—

② add $(S(x), y, S(z))$:—add (x, y, z)

③ :—add $(S(S(0)),\ S(S(S(0))), w)$

まず外観的な特徴を述べよう．このプログラムは3つの「規則」から成っている．一般に規則は，

左辺 :— 右辺

という形をしているが，左辺には高々1個（0個でも良い）の素論理式が書かれ，一方右辺は任意個（有限個，0個でも良い）の素論理式で構成される．この形の規則の意味するところは，左辺の形の素論理式を右辺の素論理式の並びで置換えよ，ということである*．もし右辺が0個の規則が適用されれば，その時の左辺の形は消去してしまって良い．また左辺が0個の規則は，無から右辺が生まれることを意味しており，この形の規則の適用はプログラムの実行の最初に一度だけ行われる．この形（左辺が0個）の規則を特に **goal** と呼ぶ．

すぐ後の実行例からもわかるように，プログラムの実行は，goal により生成された素論理式の並び（右辺）が他の規則により変形・消滅され，最終的に再び空（0個）になった時に正常終了する．

［プログラムの実行例］

最初に規則③により次が生成される．

add $(S(S(0)),\ S(S(S(0))), w)$　　(1)

これに規則②が適用される．規則②には変数 x, y, z が含まれている．また素論理式(1)にも変数 w が含まれている．これらについて次のような代入を施す．（代入を矢印←で表わすことにする．）

$x \leftarrow S(0),\ y \leftarrow S(S(S(0))),\ w \leftarrow S(z)$

この代入によって(1)は次の形となり，

add $(S(S(0)),\ S(S(S(0))),\ S(z))$　　(1)′

規則②は次の形となる．

add $(S(S(0)),\ S(S(S(0))),\ S(z))$:—add $(S(0),\ S(S(S(0))), z)$

結局 (1)′ と代入後の規則②の左辺とが全く同じ形になったので，規則を適用して右辺で置換える．

add $(S(0), S(S(S(0))), z)$　　(2)

この(2)に対しても規則②が適用される．今度の代入は次のようである．

$x \leftarrow 0,\ y \leftarrow S(S(S(0))),\ z_{(2)} \leftarrow S(z_{②})$

ここに $z_{(2)}$ と $z_{②}$ という見慣れない記号は，それぞれ，次の意味で使われている．

$z_{(2)}$……素論理式(2)の中の z を指す．
$z_{②}$……規則②の中の z を指す

PROLOG では，変数は各規則の中だけに閉じて使われる約束である．したがって，同じ z という名前が

---

* 述語論理ではこの他に，∀(ALL), ∃(EXIST) の記号も使われる：$\forall x A(x)$…すべての x について $A(x)$ が成り立つ．$\exists y B(y)$…$B(y)$ が成り立つような y が少なくともひとつは存在する．

** PROLOG の表記法については，特に標準となるものが定まっているわけではない．世の中にはここに書いたものと見掛け上かなり異なる表記をとる処理系もある．

* 規則を構成する素論理式に変数 $(x, y, z, \cdots)$ が含まれる時には，適用に当って変数に適当な項を代入して良いものとする．また書換えの対象となる素論理式中の変数についても同じことが言える．

使われていても，上の2つのzは区別される．結局（2）は次の形に置換えられる．($z_{②}$を単にzと書いた．)

add $(0, S(S(S(0))), z)$　　(3)

今度は規則①が適用される．代入は下のようになる．

$y \leftarrow S(S(S(0)))$, $z \leftarrow y$ すなわち $z \leftarrow S(S(S(0)))$

規則①には右辺が無い（0個）ので，(3)は消去されてプログラムの実行は正常終了する．

[実行例終り]

この例のように，PROLOG のプログラムでは実行が正常終了する時には最後には何も無くなってしまう．実際には，プログラムの「計算」としての動作が変数への項の代入のところで行われていたので，再び同じプログラムの実行例について見直してみよう．

[代入の追跡]

最初に規則③ (goal) により生成された(1)には変数wが含まれていた．ここでは代入の結果を追跡するために次のような表記を用いる．

add $(S(S(0)), S(S(S(0))), w) \mid w=w$　　(1)

縦棒とその右側の $w=w$ が新たに付け加えられたものである．縦棒の左側は先の [実行例] と全く同じである．

さて，(1)に対しては $w \leftarrow S(z)$ という代入が行われて (1)′ となった．

add $(S(S(0)), S(S(S(0))), S(z)) \mid w=S(z)$　(1)′

この (1)′ は規則②によって次のように置換えられた．

add $(S(0), S(S(S(0))), z) \mid w=S(z)$　　(2)

置換えの時には代入は行われないので，$w=S(z)$ は不変である．

(2)に対しても規則が適用されるが，今度は $z_{(2)} \leftarrow S(z_{②})$ という代入が施される．先の実行例では (2)′ に相当する表記は省略していたが，強いて書けば，次のようになっていた筈である．

add $(S(0), S(S(S(0))), S(z_{②})) \mid w=S(S(z_{②}))$　(2)′

そして (2)′ は置換えられて (3) となる．

add $(0, S(S(S(0))), z) \mid w=S(S(z))$　　(3)

(3)に対しては，$z \leftarrow S(S(S(0)))$ という代入が施される．

add $(0, S(S(S(0))), S(S(S(0)))) \mid$
$w=S(S(S(S(S(0)))))$　　(3)′

(3)′に対して規則①が適用されて縦棒の左の素論理式は消滅する．縦棒の右側には実行途中の代入の結果が残る．

$\mid w=S(S(S(S(S(0)))))$

これが，加算を意味していることは次に説明する．

[追跡終り]

上の例題で使われた add という述語は add (x, y, z) で $x+y=z$ を意味する．また関数Sは successor の意味で次のように自然数と対応づけられている．

$S(x) \leftrightarrow x+1$　　xは変数

$S(0) \leftrightarrow 1$

$S(S(0)) \leftrightarrow 2$

$S(S(S(0))) \leftrightarrow 3$

⋮　⋮

最初に使われた規則③ (goal) は下の形であったが，

:—add $(S(S(0)), S(S(S(0))), w)$

上の普通の自然数の記法に直すと

:—add $(2, 3, w)$

ということになる．一方，代入の追跡の結果 $w=S(S(S(S(S(0)))))$ が得られたが，これは $w=5$ ということであるから，$2+3=w$ という goal に対して $w=5$ という答が得られたことになる．

ここでは，説明の都合上規則による素論理式の生成・消滅と変数への代入を二段階に分けて見せたため，いささか煩雑な印象を与えたかも知れないが，実際の PROLOG の処理系では上に示した操作がすべて自動的に行われ，プログラムを走らせれば直ちに$w=5$ という答が返ってくる．

本稿では，以下の説明の材料として PROLOG の内部の動作を紹介したが，プログラミング言語としての PROLOG の特徴については文献[4),5)]等を参照されたい．

2.3 ホーン節とその特殊性

前節 2.2 で登場した PROLOG の「規則」は，素論理式を構成要素とするという点で，論理式と深いつながりを持っていた．以下では，規則それ自身も論理式として解釈できることを説明する．さらに規則を表わす論理式はある形式上の特徴を持つことも容易に示せる．

ここでは，一旦「規則」の形を離れて論理式の表記法を導入した後，規則が実は論理式に他ならないことを示す．さらに，その特殊性についても言及する．

(1)　素論理式を用いた論理式の構成

日常語の場合でも，単純な文（素論理式に相当）ひとつだけでは豊富な情報を表現するのは難しい．論理式の場合も事情は同じで，素論理式を論理記号で組合せることによって，初めて充分な表現力を得ることが

できる．論理記号の選び方にもいろいろな流儀がある．代表的には，∧(AND)，∨(OR)，¬(NOT)の3種類の記号に，変数に関する ∀(ALL)，∃(EXIST)を加えたものが良く使われているが，ここでは次の形をしたホーン節と呼ばれる論理式を使うことにする*．

結論←条件 1，条件 2，…，条件 n

ここに，結論，条件 1，条件 2，…，条件 n はすべて素論理式である．n は0であっても良い．そのような場合は条件が無い．すなわち結論が無条件にいつでも成り立つということを表わす論理式となる．

[例] add (x, y, z) が $x+y=z$ を表わす素論理式とすると，下のホーン節がいずれも成り立つ．

① add $(2, 3, 5)$←

② add $(0, y, y)$←

いずれも条件が0個の例である．

[例終り]

n が2個以上のときには，条件 1，条件 2，…，条件 n がすべて同時に成り立つというのが全体の条件となる．すなわち普通の論理記号で言えば，条件 1∧条件 2∧…∧条件 n と ∧(AND)の記号で結ばれていることになる．

[例]

① father (x, y)←parent (x, y), male (x)

② odd (x)←prime (x), greater $(x, 2)$

矢印の左側にある結論を表わす素論理式は有っても無くても良い．ただし有る場合でも結論は1個しか許されない．条件の側は任意個（有限個，0個でも良い）であったのと比べると対照的であり，これがホーン節と呼ばれる論理式を特徴づけている．

実は一般の論理式を上のような矢印式で表わす場合には，結論も複数個（有限個）あっても良い．

結論 1，結論 2，…結論 m←条件 1，…，条件 n この場合の全体の結論は，結論1∨結論2∨…∨結論 m ということを意味する．つまり矢印の右辺（条件側）の素論理式の並びは ∧(AND) を意味するのに対し，左辺（結論側）は ∨(OR) を意味するというわけである．

ホーン節では結論を高々1つ（0個でも良い）しか認めないということで，一般の理論式の ∨(OR) の表現力に制限を付けた形になっている．

[例] 一般の論理式では次の表現が許される．

greater (x, y), equal (x, y), less (x, y)←

これは無条件（条件部が0）に上の3つの素論理式のうちどれかが成り立つことを主張している．

次に結論側の素論理式が無い（0個）場合を考えてみよう．正直に言うと，この場合の説明は少し難しい．一般の論理式では矢印の右辺も左辺も有限個が許された．

$A_1, A_2, \cdots, A_m \leftarrow B_1, B_2, \cdots, B_n$

これを書き直すと下のようになる．

$$\bigvee_{i=1}^{m} A_i \leftarrow \bigwedge_{j=1}^{n} B_j$$

ここに各 A_i, B_j は素論理式を表わすものとする．さて，仮定 (B_j) が0個の時についてはすでに説明をしたのであるが，この表記法でもう一度考えてみよう．

$\bigwedge_{j=1}^{n} B_j$ というのは n が大きくなるにつれて ∧(AND) で結ばれている素論理式が増えるので成り立ちにくくなる．逆に n が小さいと成り立ち易くなり，極端な場合として $n=0$ では常に真を表わす論理式（$\boldsymbol{T}$ と書く）となる．

$$\bigwedge_{j=1}^{0} B_j \equiv \boldsymbol{T} \qquad (\equiv \text{は同値を表わす})$$

これと双対に $\bigvee_{i=1}^{m} A_i$ の方も考えてみると，$\bigvee_{i=1}^{m} A_i$ は m が大きくなると成り立ち易くなり，m が小さくなると成り立ちにくくなる．極端な場合として $m=0$ では常に偽を表わす論理式（$\boldsymbol{F}$）となる．

$$\bigvee_{i=1}^{0} A_i \equiv \boldsymbol{F}$$

論理式の矢印の右辺（条件側）が $\boldsymbol{T}$ の場合というのは，条件が常に成り立つというのだから，論理式全体としては無条件に結論が成り立つというのと同じことになる．これは前の記述と正に一致する．これと双対的に左辺（結論側）が $\boldsymbol{F}$ というのは結論が常に成り立たない（常に偽）というのだから，矢印で表わされた式全体としては，条件を満たすものがあると矛盾をきたす，という意味になる．

[例]

←equal $(2, 3)$

定数2と3とが等しいと仮定すると矛盾をきたす．

特に矢印の右辺にも左辺にも何も無い時は，無条件に矛盾が成り立つということになり，単に「**矛盾**」と呼んでいるものと同じものになる．

上の例からもうかがい知れるように，矢印の右辺と

---

* 従来の慣例に従ってホーン節の表記に矢印を使うが，この矢印は前出の代入を表わす矢印とは異なることに注意されたい．

左辺との間で素論理式を入換えると否定 (NOT) の意味を表わすことができる．すなわち

←equal (2, 3)

は下の式と等価である．

¬equal (2, 3)←

¬(NOT) は否定を表わす論理記号である．ただし，我々のホーン節の定義に照らすと，¬equal (2, 3) は素論理式ではない（論理記号を含んでいる）ので，ホーン節ではこの表現（¬equal (2, 3)←）は認められない．

（2） PROLOG と論理式との対応

いよいよ「規則」と「論理式」との突き合せを行う．例として再び同じものを用いる．

規則① add (0, y, y)←

これは← を ← に読み変えれば，ホーン節の一種となり，「無条件に $0+y=y$ が成り立つ」ことを表わす．以下，← と ← とは特に断わらない限り同一視する．

規則② add ($S(x)$, y, $S(z)$)←add (x, y, z)

これは，$x+y=z$ が成り立てば(条件側)，$S(x)+y=S(z)$ が成り立つ（結論側）ということを表わしている．

規則③ ←add ($S(S(0))$, $S(S(S(0)))$, w)

これは ¬(2+3=w) という意味を表わしている．ここで ¬(NOT) が用いられる理由はすぐ後で説明する．

この例のように，PROLOG のプログラムは複数個の規則の集合として表わされている．各規則はそれぞれ論理式と見なすことができるので，結局 PROLOG のプログラムというものは論理式の集合ということになる．以前に ←add ($S(S(0))$, $S(S(S(0)))$, w) という形の規則 (goal) は実行の最初に 1 度だけ使われると言ったが，それはすなわち goal の形の規則はプログラム中に 1 個しか無いことを示している．ところで上の最後の規則は（← と ← とを同一視すれば）論理式としては ←add ($S(S(0))$, $S(S(S(0)))$, w) というホーン節であり，意味的には ¬add ($S(S(0))$, $S(S(S(0)))$, w)← という論理式になる．これは次のように説明される．

一般に論理式の集合 S から定理 T*を導くには，論理式の集合に推論規則を繰返し適用して定理に至る道筋をたどる．(図-1)

これと多少異なった証明法として**背理法**（帰謬法）

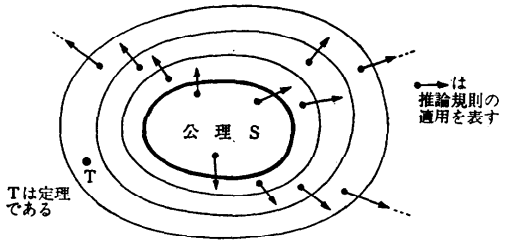

図-1 公理と定理

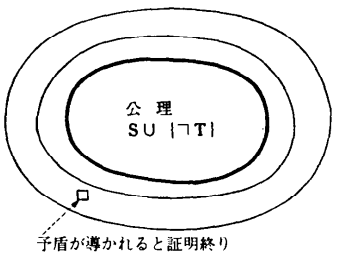

図-2 背理法による T の証明

という方法が知られている．これは導くべき定理 T を否定し（¬T），それを S に加えて（$S\cup\{\neg T\}$）その全体から矛盾を導くという方法である．(図-2)

PROLOG を論理的に眺めると，与えられたプログラムは論理式の集合であり，goal は背理法の仮定すなわち求める定理の否定であることがわかる．プログラムの実行の過程は，論理式の証明の過程に他ならない．プログラムを書き換え規則として眺めた場合には，goal により生成された素論理式の並びが変形・消滅されて，最終的に消去された時にプログラムは正常終了するのであった．これを論理式で考えると，消去されて何も無くなった状態は，

←

というホーン節で表わされる(右辺も左辺も 0 個)．これはすでに説明したように「矛盾」を表わす論理式であるから，背理法の証明で矛盾が見つかったことに相当する．

以上のように，PROLOG のプログラムの実行は論理式の集合からの定理の証明と同じものであることが判った．このような見方をした時に，ホーン節という論理式の特殊性はどのように生かされているのであろうか．

ホーン節の著しい特徴として，形式的な制約から結論部（左辺）に ∨(OR) の意味を持たせられないということがある．もし，一般の論理式のように OR を許

---

* 論理学の用語では「証明可能な論理式」のことを「定理」と呼ぶ．ここでもその意味で使っている．

してしまうと，どのような現象が起こるであろうか，次は簡単な実験である．

［例］ 非ホーン節を含むプログラム

規則① result $(x, 1)$:—greater $(x, 100)$

規則② result $(x, 0)$:—equal $(x, 100)$

規則③ result $(x, -1)$:—less $(x, 100)$

規則④ greater (x, y), equal (x, y), less (x, y):—

規則⑤ :—result $(100, w)$

言うまでもなく，規則④はホーン節の形をしていない．プログラムの実行は前と同様に⑤から始まる．

次の素論理式が生成される．

$$\text{result } (100, w) \mid w = w \qquad (1)$$

(1)に対しては規則①，②，③のいずれもが適用可能であり，それぞれ次のように(1)が置換えられる．

$$\text{greater } (100, 100) \mid w = 1 \qquad (2)$$

$$\text{equal } (100, 100) \mid w = 0 \qquad (2)'$$

$$\text{less } (100, 100) \mid w = -1 \qquad (2)''$$

通常のPROLOGの実行法では，これ以上の置換えは起こらない．すなわち規則④は，(2),(2)′,(2)″のいずれに対しても適用できない．これは④がPROLOGの範囲をはみ出す形をしているから当然予想されたことである．

そこで，PROLOGの実行法を少し拡大解釈する*．すなわち，前述(2.3節(1)の最後)のように矢印(:—と ← とは同一視)の左右で素論理式を動かすと ¬(NOT)が付くので，規則④は次のように同値変形できる．

規則④′ greater (x, y):—¬equal (x, y), ¬less (x, y)

④′の右辺は素論理式ではなくなるが，構わずに(2)に適用する．$x \leftarrow 100, y \leftarrow 100$ と代入されて，

$$\neg\text{equal } (100, 100), \neg\text{less } (100, 100) \mid w = 1 \qquad (3)$$

この(3)は論理的には次の規則⑥と同値である．

規則⑥ equal $(100, 100)$:—¬less $(100, 100) \mid w = 1$

⑥を(2)′に適用すると，

$$\neg\text{less } (100, 100) \mid w = 1, w = 0 \qquad (4)$$

となる．ここに縦棒の右の $w=1$ は規則⑥すなわち(2)に由来し，$w=0$ は(2)′から引継がれている．wに対する2通りの代入については後で論じることにし，実行を続ける．(4)は次の規則⑦と同値である．

規則⑦ less $(100, 100)$:—$\mid w = 1, w = 0$

⑦を(2)″に適用すると素論理式はすっかり消去される．

$$\mid w = 1,\ w = 0,\ w = -1$$

wに対する代入の方は(2)″の分が増えて3通りが並ぶことになった．これに対する解釈は以下に述べる．

［実験終り］

この例では，答が $w=1$，$w=0$，$w=-1$ の3通り併記されることになった．この併記の意味はもちろん∨(OR)で結ばれていることに相当する．($w=1 \wedge w=0 \wedge w=-1$ を満たす w は存在しないのは明らかであろう．)複数の答が出てしまうということは，非ホーン節の本質的な性質であるから，以下に少し詳しく説明しておこう．

上の例題は比較的単純であるから人間には答がどうなるか，すぐに判る．例えば，次の規則⑧が最初からプログラムの中に入っていれば，この規則⑧と(2)′とから素論理式は消去された筈である．

規則⑧ equal (x, x):—

この場合の答は $w=0$ ひとつである．ところが実際には例題のプログラムには規則⑧は無く，規則④だけが与えられていたのである．規則④は，

$$\text{greater } (x, y) \vee \text{equal } (x, y) \vee \text{less } (x, y)$$

という結論が無条件に成り立つことを述べているだけであり，変数 x, y に具体的な値が代入された時に3つの素論理式 greater (x, y), equal (x, y), less (x, y) のうちどれが実際に成り立つかを教えてはくれない．とにかく3つのうちどれかが成り立つ，という全体としての性質だけを主張している．

今の例題は説明用の人工的なプログラムであったが，一般に左辺に複数個の素論理式を認めてしまうと，$A \vee B$ のどちら側が実際に成り立つのか判らないという結果をもたらす．このような性質を認めてしまうと，プログラムとして実行した場合に条件分岐(if-then-else)に相当する動作が欠如してしまい，複数の答が併記されてしまう．このような理由で，PROLOGではホーン節という条件を課して，各規則の左辺(結論側)に複数個の素論理式が現われるのを禁止している．

ホーン節を論理学の側面から特徴づけると，一種のHarrop formulaということになる[6]．Harrop formula(と呼ばれる論理式の集合)の世界では，$A \vee B$ が成り立つ時にはいつでも A か B のうちどちらが本当に成り立つのかを知ることができる．

* この拡大解釈は，一般の論理式に対する resolution principle として知られているものである．3.1 節(1)参照．

3. 計算機構と推論機構

冒頭に触れたように，計算（プログラム）と証明とは比較的近い関係にあると思われている．前節では，PROLOG で書かれたプログラムが実は論理式の集合と見なせること，またその論理式というのには一定の枠がはめられており，一般の論理式がすべて PROLOG の実行法に従って解釈できるわけではないことを見た．

ここでは，計算と証明とのつながりをさらに広い観点から概観してみよう．**図-3** に著者の抱いているイメージのスケッチを示す．

3.1 計算と証明との相互関係

（1） 計算機構による証明機構の実現

1965年に Robinson によって効率的な自動証明の技法（resolution principle と呼ばれる）が提案されて以来，計算機による証明プログラムはかなり能率よく実行されるようになった．前述のように PROLOG の処理系は一種の定理証明システムとして眺めることができるのであるが，大部分の PROLOG インタプリタはこの resolution principle に基づいて動いているのが実状である．

もちろん，resolution principle の他にも多くの証明技法が知られており，計算機による実行にも数々の工夫が凝らされている．もともと定理の証明に使われる推論規則の処理というのは，本来の計算機の能力からすると易しい仕事であって，記号処理の観点からはもっと複雑な処理は他にいくらでもある．また，論理学の教科書には論理体系をゲーデル数の手法によって算術化して表現する方法が書かれているが，それによれば推論が全く機械的な計算に過ぎないことは昔から判っていたことだと言えないこともない．

ただし，ここで忘れてはならない重要なポイントが1つある．それは証明可能性の問題［論理式の集合 S と論理式 A とが与えられた時，S から A が証明可能か否かを決定する問題］は部分的に計算可能（semi-decidable, partially solvable）であるという事実である．すなわち，いかに優れた自動証明のアルゴリズムでも次の限界を破ることはできない．

［自動証明のアルゴリズム］

論理式の集合 S と論理式 A とに対してアルゴリズムを適用した結果は次のようになる．

1. S から A が証明可能な場合には，アルゴリズムは停止し “Yes” という答を出す．
2. S から A が証明できない場合には次の2つの場合のいずれかになる．
   a）アルゴリズムは停止して “No” という答を出す．
   b）アルゴリズムは停止しない（無限ループ）．

すなわち，A が証明不可能の場合にはアルゴリズムが止まる場合もあるし，止まらない場合もある．このような限界は理論的なもので，広くあてはまる．例えば PROLOG のプログラムは正常終了すべき goal を与えた時には素直に止まるが，そうでない goal に対しては無限ループに陥る可能性がある．さらに，プログラムが何時まで待っても止らない時には，果してそのうち止まって goal が無事達成されるのか，それとも無限に止まらないのか，一般には判別できないこともわかる．これは次の理由による．もしも，無限ループに陥るか否かを判定するアルゴリズムがあるとすると，そのアルゴリズムを用いることにより，上の限界が破れることになる．この種の問題についての詳しい記述は文献[7]を参照されたい．

（2） 証明機構による計算機構の解明

上の話とは逆向きの関係もある．例えば，論理式を用いてプログラムの検証[7]をするという話は良く知られている．また，プログラムの仕様（多くは入力と出力との関係）を論理式で表わしておき，その論理式を証明する過程からプログラムを作り出すというプログラムの合成の話題もある[8),9)]．

ここでは次の事実に注意しておきたい．すなわち，プログラムと論理式との対応関係は **表-1** のようになっている．どのようなプログラムでも 表-1 のプログラムの3要素から組立てられるというのは，ストラクチャードプログラミング[10]以来有名な話である．

表-1 の証明の中の推論規則の欄は上から下へと高度になっており，特に数学的帰納法の適用は一般には

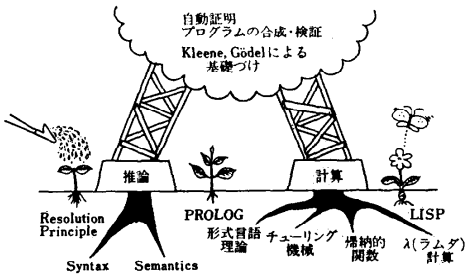

図-3 計算と推論

表-1 プログラムと論理式との対応関係

プログラムの構成要素	証明の中の推論規則
直線状に逐次実行	三段論法
条件分岐	場合分けの証明
ループ（再帰呼出し）	数学的帰納法

（説明）
- 三段論法：PROLOG の中で使われている推論
- 場合分けの証明 (case analysis)：証明の途中で2つ以上の場合に分けてそれぞれ証明する方法
- 数学的帰納法：本文で説明

機械的に行うことが難しい．ところで，プログラムの側で考えてみるとループあるいは再帰呼出しを含まないようなプログラムというのは単純至極であって，取り立てて検証や合成をする必要のないものが多い．そうなると，結局数学的帰納法の難しさが問題全体の難しさを支配してしまうわけである．

前述の PROLOG による加算プログラムの例では，:—add $(S(S(0)), S(S(S(0))), w)$ という goal に対して規則②が2回適用されて正常終了したのであったが，これは第1項の $S(S(0))$ が $S(x)$ という形の項であったからである．もし第1項が C という定数の項であったとすると :—add $(C, S(S(S(0))), w)$ という goal は通常の PROLOG の実行法では処理できない．すなわち，goal は次の素論理式を生成するが，

add $(C, S(S(S(0))), w)$

この素論理式には規則①も②も適用できない．

ところで :—add $(C, S(S(S(0))), w)$ という goal の論理式としての意味を考えてみると，

$$\forall x \exists w.\ \text{add}\ (x, S(S(S(0))), w) \qquad (5)$$

という定理を背理法で証明しようとして否定をとったものである*．

ところで，規則①，規則②をもう一度眺めてみると，

規則① add $(0, y, y)$:—

規則② add $(S(x), y, S(z))$:—add (x, y, z)

①は $0+y=y$ を主張しており，②は $x+y=z$ ならば $S(x)+y=S(z)$ であることを主張している．以下では PROLOG としての規則の適用にこだわらずに①と②から導ける事実を考察してみる．まず①の主張により，

$$0+y=y$$

である．これに規則②を勘案すると，

$$S(0)+y=S(y)$$

が得られ，再度これを規則②の条件側と思えば，

$$S(S(0))+y=S(S(y))$$

以下同様に反復すれば，

$$S^n(0)+y=S^n(y) \qquad (6)$$

を得る．ここに $S^n(0)$ とは $S(S(S\cdots(0))))$（n 回）を表わす．これ以上は何回②を適用しても(6)の形を脱することはできないが，このような場合に数学的帰納法 (mathematical induction) の推論規則を適用すると次の形の論理式が得られる*．

$$\forall x \exists w.\ x+y=w \qquad (7)$$

(7)と先の論理式(5)とを見比べると，$y=S(S(S(0)))$ を代入すれば全く同じものになる．すなわち，(5)は数学的帰納法を用いれば本来は正しく証明されるべき論理式なのであった．

このように，PROLOG では数学的帰納法は陽には実行することができないため，上述の加算プログラムで 100+200=300 を実行しようとすれば実に 100 回も規則②を適用しなければならない．もちろん，現実の PROLOG の処理系では加算などの算術演算は組込み述語として一瞬に計算することになっているし，集合の表記などの高位の概念も用意されているので，上の実例のような単純な問題で困ることはあり得ない．しかし，数学的帰納法の推論規則が PROLOG に含まれていないという点は，論理的な観点からは本質的な特徴として映る．

3.2 Logic の計算モデル

以上の記述では PROLOG を主な題材にしてきたが，ここでは範囲を広げて代表的なものをまとめておこう．

(1) PROLOG 再論，unification

プログラミング言語として PROLOG のが，すべての計算可能な関数を表現できることは，Tärnlund[11] によって示された．彼の証明法は万能チューリング機械 (Universal Turing Machine) を PROLOG で記述するという方法であった．PROLOG のプログラムはホーン節によって構成され，そのホーン節は論理式

* $\forall x$ や $\exists w$ の取扱いは本稿では充分に説明していないが，次のように意味的に変形して考えることができる．:—add$(C, S(S(S(0))), w)$ は $\neg$add$(C, S(S(S(0))), w)$ であり，これは $\exists x \forall w.\ \neg$add$(x, S(S(S(0))), w)$ を意味する．後者は $\neg\forall x \exists w.$ add$(x, S(S(S(0))), w)$ と同値である．

* 数学的帰納法は論理式 $p(0)$ と $p(x)$ ならば $p(S(x))$ から $\forall x.\ p(x)$ を導くものである．ここでは，$p(x)$ を

$$p(x) \equiv \exists w.\ \text{add}(x, y, w)$$

と定めた．$p(0)$ は規則①により add$(0, y, y)$ が言えるから $\exists w$ の w として y をとれば $\exists w.$ add$(0, y, w)$ となるから良い．また $p(x)$ ならば $p(S(x))$ の方は規則②から add(x, y, z) ならば add$(S(x), y, S(z))$ が言えて，上と同様の操作によって $\exists w.$ add(x, y, w) ならば $\exists w.$ add$(S(x), y, w)$ となるから良い．

としては制約条件を課せられているにも拘らず，計算モデルとしては万能であるという結果は興味深い．

前述のように (2,2節)，PROLOG の実行過程では変数への代入操作が中心的な役割を果している．この代入操作は **unification** (統一化あるいは単一化と訳す) と呼ばれる．つまり，2つの項を代入によって unify するという意味である．unification についての面白い話題としては，二階 unification (関数変数を含む項を扱う) がある意味で「計算」になっているという事実がある．これは Goldfarb の論文[12]の中で紹介されている．彼は二階の unification の決定不可能性 (undecidability) を証明するために，ヒルベルトの第10問題を二階の unification に帰着させた．その証明の中で加算と乗算を実現 (原文では simulate) する unification について述べている．

(2) FOL[13]

FOL という名前は First Order Logic に由来している．FOL は通常の分類では証明チェッカ (proof checker) という範疇に属するものであるが，単に人間が与えた証明をチェックしてくれるという機能に留まらず，人間と計算機とがインタラクティブに論理体系を作り上げて行く，という視点に立った設計が行われている．

FOL の特徴はいろいろあるが，まず記号操作としての証明の部分と，論理式の構成要素を現実世界の対象物 (FOL では LISP の世界) に対応づける部分とが明確に分離されている点がある．本来，論理学では syntax (記号操作) と semantics (モデルとの対応づけ) とは独立した概念であるにも拘らず，往往にして混同されている．特に resolution principle (3.1節(1)) は両者の混合物といった色彩を帯びており，PROLOG にもその傾向は受継がれている．

FOL の第2の特徴としては，高階の操作 (原文では metatheory) についての機能が用意されていることである．PROLOG にも高階の機能が一部使われてはいるが，どちらかというと例外的な扱いを受けている．

(3) Tableau, Normalization

先に述べたように (2.3節(2))，PROLOG は一種の証明システムである．通常の PROLOG の処理系では，自動証明のアルゴリズムとして resolution principle が使われている (3.1節(1))自動証明のアルゴリズムとしては resolution principle 以外の手法も知られている．例えば，文献[14]は **tableau** 法と呼ばれる証明技法で PROLOG の処理系を構成した語である．

必ずしも「自動」を目差さなくても良ければ，選択の範囲はさらに広がる．因に前出の FOL は natural deduction (自然演繹法) という古くから知られている証明法に準拠している．

RPOLOG では証明を作り上げながら同時に計算が進行するが，これと異なる方法として，まず証明を完成させておき，後から計算 (変数への代入) を追跡するという二段構えも考えられる．例えば，3.2節(2)で触れたように数学的帰納法の推論規則を適用しようと思えば，まず変数の項のままで証明を進めておき，必要になった所で適切な項を代入することになる．このような方法に関連して，**説明図の normalization** という概念がある．詳しくは文献[6]に譲るが，大まかに言えば，normalization は証明図をあたかも λ (ラムダ) 式のように扱うものである．すなわち，変数を含む証明図に具体的な項を与えると，λ 式と類似の簡約規則が動き出し，最終的には PROLOG と同様の「答」に到達するというものである．

4. おわりに——拡張の可能性，一階述語論理を超えて

先に細かい定義を導入した際 (2.1節) に，関数と述語とは大体同類であると書いた．関数というのは普通には集合から集合への対応関係を言う．

[関数] $f: X \to Y$

このような見方をすると述語というのは true/false の値だけを取る関数と見なせる．

[述語] $p: X \to \{\text{true, false}\}$

述語を関数と思ってしまえば，素論理式は一種の項の形をしている．

$$\underbrace{\text{add}}_{\text{関数と思う}}\ (\underbrace{2}_{\text{項}},\ \underbrace{3}_{\text{項}},\ \underbrace{w}_{\text{項}})$$

この種の項を用いて再び素論理式を組立てることができる．例えば add (2, 3, w) が論理式であるというのを，

$$\underbrace{\text{formula}}_{\text{述語}}\ (\underbrace{\text{add}\ (2, 3, w)}_{\text{項}})$$

と書く場合がそうである．これは形の上では今までの素論理式と何ら変らないけれども，本当は，

$$\underbrace{\text{formula}}_{\text{述語}}\ (\underbrace{\text{add}\ (2, 3, w)}_{\text{素論理式}})$$

なのであるから，大袈裟に言えば「論理式についての

論理式」の一例になっている．

このようにして，高位の概念というものを考えることができる．高位の概念などというと何か恐しげであるが，実際は日常的にありふれたものである．例えば，常識的に考えれば「論理学」の教科書には「論理」そのものが書かれているように思うけれども，良く考えてみれば，教科書では「論理」を対象として扱っている．つまり，教科書の内容は，日本語（あるいは英語）で書かれた「『論理』について部分論理」に他ならない．

また，人間自身を一種の論理体系として捉えると，自分自身の推論の仕方についての反省とか，新しい推論法の獲得（論理体系の拡大に相当する）を常時行っている．したがって，人間に匹敵するとは言わないまでも，人間と自然に対話できるような論理システムを構成するには，高位の概念が不可欠である．

ただし，高位のものは何と言っても難しいので，今までのところ手が付けられているのは極く一部である．本稿も主としてPROLOGとその周辺の一階述語論理の話題に終始したが，上に述べたような遠大な目標（人間並み）に比べると解説し得た部分は，ささやかな第一歩に過ぎない．

参 考 文 献

1) Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P.: Un system de Communication Homme-machine en Français, Rapport de recherche, Groupe d'Intelligence Artificielle, UER de Luminy, Universite d'Aix Marseille (1973).
2) Kowalski, R.: Predicate Logic as Programming Language, IFIP-74, pp. 569-574.
3) Colmerauer, A.: PROLOG and Infinite Trees, Logic Programming, pp. 231-251, Academic Press (1982)
4) 横井俊夫，淵 一博：推論機構を内蔵した述語論理型言語 Prolog，日経エレクトロニクス，「人工知能」特集号 (No. 300) 1982年9月27日号．
5) 中島秀之：Prolog 入門，bit, Vol. 14, No. 5～No. 7. pp. 685-691, pp. 771-776, pp. 856-863, 1982年4月～6月号連載．
6) Troelstra, A. S.: Metamathematical Investigation of Intuitionistic Arithmetic and Analysis, Lecture Notes in Mathematics, No. 344, Springer-Verlag (1973).
7) Manna, Z.: Mathematical Theory of Computation, McGraw-Hill (1974).
8) Manna, Z. and Waldingr, R.: A Deductive Approach to Program Synthesis, ACM TOPLAS, Vol. 2, No. 1, pp. 90-121, (1980).
9) 後藤滋樹：プログラム・シンセシス，情報処理，Vol. 22, No. 3, pp. 218-225 (3, 1981).
10) O.-J.-Dahl, Dijkstra, E. W. and Hoare, C. A. R.（野下浩平，川合 慧，武市正人訳）：構造化プログラミング，サイエンス社 (1975).
11) S-A. Tarnlund: Horn Clause Computability, BIT, Vol. 17, pp. 215-226 (1977).
（注）この BIT と文献[5]の bit とは別の雑誌である．
12) Goldfarb, W. D.: The Undecidability of the Second-Order Unification Problem, Theoretical Computer Science, Vol. 13, pp. 225-230 (1981).
13) Weyhrauch, R. W.: Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, Vol. 13, No. 1, 2 (合併号). pp. 133-170 (1980).
14) Broda, K.: The Relation Between Semantic Tableaux and Resolution Theorem Provers, Logic Programming Workshop, pp. 293-304 (1980).

（昭和57年11月16日受付）