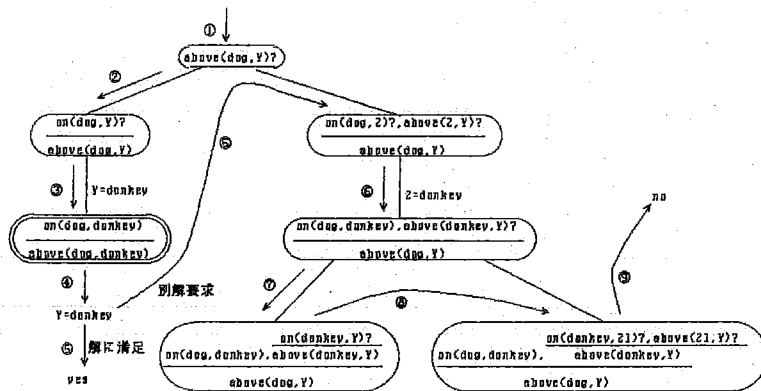
論理とＰｒｏｌｏｇ

（情報処理概論Ｄ Ｉ～ＩＩ，
木曜４限，１９９３，講義ノート）



新潟大学教養部
情報処理教室
元木 達也

目　次

第１章　命題論理 ……………………………………………1

1.1　文，言明と命題 ……………………………………1
1.2　真理関数的結合詞と複合命題 ……………………………2
1.3　論理構造の明確化 …………………………………6
1.4　命題論理式 ……………………………………………8
1.5　命題論理式の解釈と真理表 ……………………………9
1.6　論理的帰結 ……………………………………………13
1.7　命題論理式の代数 ……………………………………14
1.8　標準形 ……………………………………………………17
1.9　命題の真理関数的正しさと同値性 ……………………21
　　　演習問題１ ……………………………………………25

第２章　述語論理 ……………………………………………27

2.1　命題論理の能力の限界 ………………………………27
2.2　限量詞 ……………………………………………………29
2.3　論理構造の明確化 ……………………………………31
2.4　一階論理式　―述語表現と関数表現― …………………35
2.5　一階論理式の解釈 ……………………………………40
2.6　論理的帰結 ……………………………………………43
2.7　一階論理式の代数 ……………………………………45
2.8　標準形 ……………………………………………………48
2.9　命題の限量的正しさと同値性 …………………………50
　　　演習問題２ ……………………………………………52

第３章　公理化　―知識の体系化―

第３章　公理化　―知識の体系化―　……………………………55

３.１　公理と演繹による数学の体系化　………………………55
３.２　演繹の公理化　…………………………………………58
３.３　命題論理の公理系　……………………………………59
３.４　一階論理の公理系　……………………………………63
３.５　知識ベース　―計算機内での知識の体系化―　……………66
演習問題３　…………………………………………………71

第４章　Ｐｒｏｌｏｇのプログラムとその実行

第４章　Ｐｒｏｌｏｇのプログラムとその実行　……………………………73

４.１　Ｐｒｏｌｏｇのプログラムと公理系　…………………73
４.２　Ｐｒｏｌｏｇプログラムの一般形　……………………75
４.３　Ｐｒｏｌｏｇプログラムの動作原理　…………………78
４.４　Ｐｒｏｌｏｇプログラムの作成／実行　………………84
演習問題４　…………………………………………………88

第５章　Ｐｒｏｌｏｇ初級プログラミング

第５章　Ｐｒｏｌｏｇ初級プログラミング　…91

５.１　単一化　………………………………………………91
５.２　算術計算　……………………………………………94
５.３　プログラムの手続き的解釈と箱モデル　………………102
５.４　デバッガ　……………………………………………108
５.５　入出力のための基本述語　……………………………114
５.６　Ｐｒｏｌｏｇによるパズルの解法　……………………118
演習問題５　…………………………………………………127

第６章　Ｐｒｏｌｏｇプログラミング技法

第６章　Ｐｒｏｌｏｇプログラミング技法　‥131

６.１　リストの扱い　………………………………………131
６.２　実行制御のための述語　………………………………148

6.3　知識ベース操作のための述語　……………………162
演習問題6　……………………………………175

付録　組込み関数，組込み述語一覧　…………179
文献　……………………………………………………187
索引　……………………………………………………191

第1章　命題論理

論理学は“必然的な推理”の科学であり、論理的思考は日常会話の中に多く含まれる。［実際、ｌｏｇｉｃという単語は言葉や理性や思想を意味するギリシャ語のλογος（ロゴス）を語源とする。］１～２章では、日常会話の中に現れる論理的推論とその形式化について議論しよう。

１．１　文，言明と命題

日常言語における文のうち、その真偽が決まっているものがある。例えば、“鉄は金属である”の様に正しいことを言っている文は真(true)であるといい、“犬は翼を持つ”の様に誤ったことを言っている文は偽(false)であるという。論理学では、この様に真偽の決まっている文を扱う。

定義１．１　　平叙文（あるいは平叙文をつなげてできる文章）のうち、真か偽かのいずれかに決まっているものを**言明**(statement,**表述**)といい、言明の意味／内容を**命題**(proposition)という。また、命題に対して本来定まっている真偽性をその命題の**真理値**(truth value)という。

例１．２　　次の文のうちで言明は(1)～(16)だけである。このうち(1)～(6)の言明は真の命題を表し、(7)～(9)は偽の命題を表す。もちろん、(4)～(6)は言明としては異なるが命題としては同じである。また、(10)の（表す）命題の真理値は不明であり、(11)～(16)の（表す）命題の真理値は下線部の語の意味に応じて真になったり偽になったりする。

(1) 月曜日の次の日は火曜日である。
(2) 平成５年度は、新潟大学への入学式は４月９日（金）に行われた。
(3) 珊瑚は動物である。
(4) 山形県は新潟県と接している。
(5) 新潟県は山形県と接している。
(6) 新潟県と山形県は隣り合っている。
(7) 犬は翼を持っている。

(8) １と１の和は３である。

(9) シャーロック・ホームズは実在の人物であった。

(10) 円周率の小数点以下１兆桁目は３である。

(11) 大型間接税は国民のためになる。

(12) 今日は月曜日だ。

(13) 新潟から東京は遠い。

(14) 私は法蓮草が好きだ。

(15) ここは暑い。

(16) 彼女は美人だ。

(17) 新潟大学の所在地.

(18) 何時ですか？

(19) ドアを閉めろ。

(20) ああ、私がもっと若ければなあ！

１．２　真理関数的結合詞と複合命題

日常言語では、複数の言明を“そして”，“または”，“それゆえ”，“しかし”という接続詞，“(もし)〜なら”という助動詞，“〜(する前)に”という格助詞，“(たとえ)〜でも”という係助詞，“〜(である)ので”，“〜(である)が”という接続助詞でつなげたり、１つの言明に“〜ない”という否定の助動詞をつなげたりすることによって、より大きな言明を構成する。論理学では、これらの接続詞,助動詞や助詞のうちで特に“真理関数的”なものに注目する。

定義１.３　言明のうちで、より小さな言明を構成要素として含まないものを**原子言明**(atomic statement)といい、それ以外のものを**複合言明**(compound statement)という。そして、原子言明の表す命題を**原子命題**(atomic proposition)といい、それ以外の命題を**複合命題**(compound proposition)という。また、複合言明の中の接続詞,助動詞や助詞といった**結合詞**(connective)は、その複合言明の真理値がその構成要素である言明の真理値のみによって決まるときに、**真理関数的**(truth-functional)であるという。

次に真理関数的結合詞として“否定”，“連言”，“選言”，“排他的選言”，“含意”，“同値”の６種類をあげ、その次に真理関数的でない結合詞もいくつか示す。［以下で例示されるように、“または”という語は選言の結合詞としても連言の結合詞としても用いられる。従って、どの語がどんな意味で用いられるかは一般には決まっていない。］

否定(negation).　言明が任意に与えられた時、それを否定することによって別の言明を作ることができる。この結果できる複合命題の真理値は、元の言明の表す命題が真であるか偽であるかにのみ依存して、それぞれの場合に真，偽となる。例えば、

(1) 犬は翼を持たない，

(2) 犬が翼を持つことはない，

(3) 犬が翼を持つなんてあり得ない

は、すべて“犬は翼を持つ”という原子言明を否定して得られる言明であり、すべて同一の命題を表す。また、元の原子命題は偽であるから、その否定命題は真である。

連言(conjunction).　2つの言明が任意に与えられた時、それらがともに真であることを示す結合詞によって別の言明を作ることができる。この結果できる複合命題の真理値は、元の言明の表す命題が2つとも真である場合に真となり、どちらか一方でも偽であれば偽となる。例えば、

(4) 新潟県は山形県と接している，そして、新潟県は群馬県と接していない，

(5) 新潟県は山形県と接している，しかし、新潟県は群馬県と接していない，

(6) 新潟県は山形県と接しているが群馬県とは接していない

は、すべて“新潟県は山形県と接している”という原子言明と“新潟県は群馬県と接していない”という複合言明がともに真であることを表す言明であり、すべて同一の命題を表す。もちろん、元の2つの命題はそれぞれ真,偽であるから、これらの連言命題は偽である。

ここで、(5)の“しかし”は、日常会話では単に2つの構成要素である言明が真であることを言うだけではなく、2つの事実の対照性を強調したり、対照性に対する驚きを表すのに用いられる。しかし、論理的な観点からみると、その言明で述べられている事実の真偽性だけが重要なのであり、強調や驚きは無視される。また、(6)の言明は(5)の省略形と考えられる。

連言を表すのに“または”という語が使われることもある。例えば、次の2つの言明は同じ命題を表す。

(7) 日本国民または過去一年間日本に居住した者は、所得税の納入を義務づけられる。

(8) 日本国民は所得税の納入を義務づけられる。そして、過去一年間日本に居住した者も所得税の納入を義務づけられる。

選言(disjunction).　2つの言明が任意に与えられた時、それらのうちのいずれか一方（または両方）が真であることを示す結合詞によって別の言明を作ることができる。［このような結合詞を次の排他的選言と区別するときには、特に、**両立的**(inclusive)**選言**という。］この結果できる複合命題の真理値は、元の言明の表す命題が2つとも偽である場合に偽となり、また、どちらか一方でも真であれば真となる。例えば、

(9) 新潟県は山形県と接している，または、新潟県は群馬県と接している，

(10) 新潟県は山形県または群馬県と接している，

(11) 新潟県は山形県,群馬県のうちの少なくとも１つの県と接している

は、すべて“新潟県は山形県と接している”と“新潟県は群馬県と接している”の２つの原子言明のいずれか一方（または両方）が真であることを表す言明であり、すべて同一の命題を表す。もちろん、元の２つの原子命題は真であるから、これらの選言命題も真である。

排他的選言(exclusive disjunction).　２つの言明が任意に与えられた時、それらのうちのいずれか一方だけが真であることを示す結合詞によって別の言明を作ることができる。この結果できる複合命題の真理値は、元の２つの言明の表す命題のうち１つが真であり残りの１つが偽であるときに真となり、元の２つの命題が共に真であったり共に偽であるときには偽となる。例えば、

(12) 珊瑚は動物であるか、または、珊瑚は植物であるかのいずれかである,

(13) 珊瑚は動物か植物かのいずれかの界に分類される,

(14) 珊瑚は動物か植物かのどちらかである

は、すべて“珊瑚は動物である”と“珊瑚は植物である”の２つの原子言明のどちらか一方だけが真であることを表す言明であり、すべて同一の命題を表す。もちろん、元の２つの原子言明はそれぞれ真,偽であるから、これらの排他的選言命題は真である。

含意(implication).　２つの言明が任意に与えられた時、それらのうちの一方（**前件**,antecedent,という）が真なら他方（**後件**,cnsequent,という）も真であることを示す結合詞によって別の言明を作ることができる。この結果できる複合命題の真理値は、前件の命題が真で後件の命題が偽の時に偽となり、それ以外の時、すなわち、前件が偽であったり後件が真であるときには真となる。例えば、

(15) 新潟市に大雪が降った場合には、その日の新潟大学での大学入試センター試験は開始時刻を繰り下げて実施される,

(16) もし新潟市に大雪が降れば、その日の新潟大学での大学入試センター試験は開始時刻を繰り下げて実施される

は、ともに“新潟市に大雪が降る”と“その日の新潟大学での大学入試センター試験は開始時刻を繰り下げて実施される”の２つの原子言明をそれぞれ前件,後件とする含意の複合言明であり、互いに同じ命題を表す。(15)と(16)が平成４年１月１１日(土)～１２日(日)のことを言った文だとすると、実際には前件も後件も偽であったからこれらの含意命題は真である。

同値(equivalence).　２つの言明が任意に与えられた時、それらの真理値が同じであることを示す結合詞によって別の言明を作ることができる。この結果できる複合命題の真理値は、元の２つの言明の表す命題のうち１つが真であり残りの１つが偽であるときに偽となり、元の２つの命題が共に真であったり共に偽であるときには真となる。［従って、同値の複合命題の真理値は排他的選言の場合の逆である。］例えば、

(17) 新潟市の市長が女性である時、かつその時に限り、新潟市の市長は男性ではない

は、“新潟市の市長が女性である”という原子言明と“新潟市の市長は男性ではない”という複合言明が同じ真理値を持つことを表す言明である。もちろん、平成４年４月２３日(木)の時点では、元の２つの命題は偽であるからこの同値命題は真である。

真理関数的でない結合詞（原因・理由）． 例えば、

(18) 甲子園球場で雨が降ったので、その日の巨人－阪神戦は中止になった，

(19) その日の巨人－阪神戦は甲子園球場の雨のため中止になった

は、ともに“甲子園球場で雨が降った”と“その日の巨人－阪神戦は中止になった”の２つの原子言明を結合詞でつないで得られる複合言明であり、互いに同じ命題を表す。(18)と(19)が昭和６３年４月１２日(火)のことを言った文だとするとこの複合命題は真であるが、それは、これを構成する２つの原子命題が真であるためだけでなく、２つの原子命題の因果関係が正しいことにも由来する。“～(である)ので”，“従って”，“なぜなら”という結合詞を用いて得られる複合命題が真となるためには、それを構成する２つの命題がともに真であることのほかに、それらの因果関係が正しいことをも要求する。実際、２つの原子命題“鉄は金属である”と“その日の巨人－阪神戦は中止になった”がともに真であるにもかかわらず、これらから構成される複合命題

(20) 鉄は金属であるので、その日の巨人－阪神戦は中止になった

は、原子命題間の因果関係が正しいと認められないために、真と考えることはできない。(18)と(20)が昭和６３年４月１２日(火)のことを言った文だとすると、これらの言明は、両方とも真の命題をまったく同様に結合して得られたのに、片方は真の複合命題ができもう片方は偽の複合命題ができる。従って、ここで挙げた結合詞は真理関数的でないことになる。

真理関数的でない結合詞（叙想法,仮定法,subjunctive mood）． 例えば、

(21) 太平洋戦争の終わるのがもう１年遅くなっていたら、新潟に原爆が投下されていただろう

は、“太平洋戦争の終わるのがもう１年遅くなる”と“新潟に原爆が投下されていた”という２つの原子言明を結合詞でつないで得られる複合言明である。もちろん、この複合言明は真と考えられるが、それは、これを構成する原子命題が両方偽であるからではなく、様々な資料から真であることがわかるのである。実際、２つの原子命題“太平洋戦争の終わるのがもう１年遅くなる”と“ニューヨークに原爆が投下されていた”がともに偽であるにもかかわらず、これらから構成される複合命題

(22) 太平洋戦争の終わるのがもう１年遅くなっていたら、ニューヨークに原爆が投下されていただろう

は、明らかに偽と考えられる。(21)と(22)は、両方とも偽の命題をまったく同様に結合して得られたのに、片方は真の複合命題ができもう片方は偽の複合命題ができる。従って、ここで挙げた結合詞は真理関数的でないことになる。

１．３　論理構造の明確化

前節１．２では比較的簡単な言明だけを見てきたが、日常言語ではもっと複雑で結合詞が何重にもなった言明もある。複雑な言明が会話中に与えられた場合、それぞれの結合詞がどの言明とどの言明を結び付けているかを、我々は会話の流れなどから適切に判断する。［話をする方も、曖昧な言明にならないように心掛けて喋ることが多い。］しかし、この判断ができないこともある。例えば、単なる文字列として、

(1)　xを未知数とする方程式　$ax+b=0$　が解を持つための必要十分条件は、

$a\neq 0$　または　$a=0$　かつ　$b=0$

である

という言明が与えられたとしよう。これは、もちろん、３つの原子言明“xを未知数とする方程式　$ax+b=0$　が解を持つ”，“$a=0$”（２回現れる），“$b=0$”に否定，選言，連言，同値の結合詞が１回ずつ適用されて構成される複合言明であって、“または”と“かつ”がどの言明を結合するのかが曖昧になっている。このような曖昧さを避けるためには、通常、コンマ等を有効に用いるが、単に曖昧さを除去するだけなら括弧を用いるのが簡単である。例えば、括弧を用いると(1)は、

(2)　xを未知数とする方程式　$ax+b=0$　が解を持つための必要十分条件は、

（$a\neq 0$　または　$a=0$）　かつ　$b=0$

である

または

(3)　xを未知数とする方程式　$ax+b=0$　が解を持つための必要十分条件は、

$a\neq 0$　または　（$a=0$　かつ　$b=0$）

である

と明確化できる。［普通、我々は(1)が正しいことを言っていると考えて(3)の解釈を行うが、元々(1)が正しいことを言っている保証はないから(2)の解釈もあり得る。］

言明の結合の曖昧さを除去するには括弧を用いるのが有効で簡単な方法であることをこれまで説明したが、これだけで全ての言明の論理構造が明確になるとは言えない。なぜなら、１．２節で見た様に、同一の働きをする真理関数的結合詞が多数存在するからである。括弧を用いれば言明の構造は明確になるが、それだけでは命題の構造は明確にならないのである。例えば、１．２節(9)～(11)は全て同一の論理構造を持つと考えられるが、一見しただけではこれはわからない。そこで、言明の論理構造を明確化するために、さらに、否定,連言,選言,排他的選言,含意,同値の結合詞をそれぞれ　$\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$　という記号（結合子,connective,という）で置き換えよう。すなわち、

αでない	という意味の否定の複合命題を	$(\sim\alpha)$　で,
αかつβ	という意味の連言の複合命題を	$(\alpha\wedge\beta)$　で,
αまたはβ	という意味の選言の複合命題を	$(\alpha\vee\beta)$　で,

α, βのうち1つだけ成立　という意味の排他的選言の複合命題を　$(\alpha \oplus \beta)$　で,
αならばβ　という意味の含意の複合命題を　$(\alpha \to \beta)$　で,
αとβが同値　という意味の同値の複合命題を　$(\alpha \leftrightarrow \beta)$　で

置き換える。これによって、例えば(3)の複合命題は次の様に書き換えられる。

(4) (xを未知数とする方程式　$ax+b=0$　が解を持つ
　$\leftrightarrow$　$((\sim a=0) \vee (a=0 \wedge b=0)))$

この(4)は、日常文としては奇妙なものであるが、論理構造がはっきりしているという点で我々の目的（すなわち真理関数的結合詞に関する研究）に合った表現である。

複雑な表現で一挙に全ての結合詞を記号で置き換えられないものも当然あり得る。この様な場合、

(5) 記号で置き換えられていない結合詞のうちで最も外側のものを記号で置き換える

という操作を繰り返し適用すればよい。例えば、

(6) もし銀行の中央のコンピュータがシステムダウンすれば、その原因がすぐに発見されてかつすぐに修復可能なものでなければ、その銀行の窓口はパニック状態に陥る

という言明に(5)の操作を繰り返し適用すると次の様になる。[最後の(10)が(6)の全ての結合詞を記号で置き換えて得られる表現である。]

(7) (銀行の中央のコンピュータがシステムダウンする
　$\to$　その原因がすぐに発見されてかつすぐに修復可能なものでなければ、
　　その銀行の窓口はパニック状態に陥る　)

(8) (銀行の中央のコンピュータがシステムダウンする
　$\to$　(その原因がすぐに発見されてかつすぐに修復可能なものでない
　　$\to$　その銀行の窓口はパニック状態に陥る　))

(9) (銀行の中央のコンピュータがシステムダウンする
　$\to$　(($\sim$　その原因はすぐに発見されてかつすぐに修復可能なものである)
　　$\to$　その銀行の窓口はパニック状態に陥る　))

(10) (銀行の中央のコンピュータがシステムダウンする
　$\to$　(($\sim$(その原因はすぐに発見される
　　　$\wedge$　その原因はすぐに修復可能なものである))
　　$\to$　その銀行の窓口はパニック状態に陥る　))

1．4　命題論理式

命題論理学では、個々の原子命題の構造や真偽性については全く関心がなく、どんな複合命題のパターンがどんな時に結合子 $\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ の真理関数的性質によって真となるか，どんな推論パターンが結合子 $\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ の真理関数的性質によって常に正しくなるかを調べる。そのために、具体的な原子命題を $p, q, r, s, p_1, q_1, r_1, s_1, p_2, q_2, r_2, s_2, \ldots$ 等の文字（言明文字, statement letter, という）で置き換えて命題の真理関数的な骨組みにだけ着目する。この様な骨組みを“命題論理式”または“真理関数的型式”といい、次の様に形式化する。

定義1．4　**命題論理式**(propositional formula, **真理関数的型式**)のクラスを次の (i), (ii) により帰納的に定める。

(i) 言明文字はどれも命題論理式である。

(ii) 命題論理式 α, β に対して、$(\sim\alpha), (\alpha \wedge \beta), (\alpha \vee \beta), (\alpha \oplus \beta), (\alpha \rightarrow \beta), (\alpha \leftrightarrow \beta)$ の6つの文字列パターンはどれも命題論理式である。

例1．5　前節1．3(4)の複合命題において、

“xを未知数とする方程式 $ax+b=0$ が解を持つ” という原子命題を言明文字 pで，

“$a=0$” という原子命題を言明文字 qで，

“$b=0$” という原子命題を言明文字 rで

置き換えると、

(1)　$(p \leftrightarrow ((\sim q) \vee (q \wedge r)))$

という構造が得られる。これは定義1．4によって命題論理式である。なぜなら、

(2) 定義1．4(i)より p は命題論理式である。

(3) 定義1．4(i)より q は命題論理式である。

(4) 定義1．4(i)より r は命題論理式である。

(5) 定義1．4(ii)と(3)より　$(\sim q)$ は命題論理式である。

(6) 定義1．4(ii)と(3), (4)より $(q \wedge r)$ は命題論理式である。

(7) 定義1．4(ii)と(5), (6)より $((\sim q) \vee (q \wedge r))$ は命題論理式である。

(8) 定義1．4(ii)と(2), (7)より $(p \leftrightarrow ((\sim q) \vee (q \wedge r)))$ は命題論理式である。

例1．6　前節1．3(10)の複合命題において、

“銀行の中央のコンピュータがシステムダウンする” という原子命題を言明文字 pで，

“その原因はすぐに発見される” という原子命題を言明文字 qで，

“その原因はすぐに修復可能なものである” という原子命題を言明文字 rで，

“その銀行の窓口はパニック状態に陥る” という原子命題を言明文字 sで

置き換えると、

(9)　$(p \to ((\sim(q \wedge r)) \to s))$

という構造が得られる。(1)式と同様に、これも命題論理式と認められる。

略記法　通常の算術式の場合に倣って、命題論理式の括弧の省略を行う。その際、結合の優先順位は $\sim, \wedge, \vee, \oplus, \to, \leftrightarrow$ の順に高いものとする。

例1.7　(1),(9)の命題論理式で不要な括弧を除去すると、それぞれ、次の様になる。

(10)　$p \leftrightarrow \sim q \vee q \wedge r$

(11)　$p \to (\sim(q \wedge r) \to s)$

1.5　命題論理式の解釈と真理表

命題論理式は命題の真理関数的構造を表すものであって、その中に現れる個々の言明文字が具体的にどの言明文字を表すかは決まっていない。それゆえ、命題論理式はそれ自身固定された真偽性を持たない。2つの言明の真理関数的構造が同じ形の命題論理式によって表されていたとしても、それらの真偽性が異なることもある。

しかし、命題論理式の中のどの言明文字が真の命題を表し，どの言明文字が偽の命題を表すかを決めてしまえば、この制約付きの命題論理式を骨組みとする命題の真理値は一意的に決まってしまう。この意味で、各言明文字への真理値の割当てが個々の命題論理式の真理値を決定すると言ってもよい。従って、言明文字への真理値の割当てをどうすれは真の命題が得られ，どうすれば偽の命題が得られるかを問題にする場合には、命題論理式中の個々の言明文字にどちらの真理値を割り当てるかだけを考えれば十分である。［言明文字に原子命題を割り当てる方法は無限にあるが、真理値を割り当てる方法は有限であることに注目。］

定義1.8　言明文字の各々に真理値を割り当てることを解釈(interpretation)といい、解釈 I によって定まる命題論理式 α の真理値を $I(\alpha)$ で表す。また、$I(\alpha)=$真 の時 $\models_I \alpha$，$I(\alpha)=$偽 の時 $\not\models_I \alpha$ と書く。

例1.9　言明文字 p, q, r, s にそれぞれ 真,偽,真,偽 を割り当てる解釈 I によって、前節1.4(10)の命題論理式の真理値は次の様に決まる。

(1)　I の定義から　$I(p)=$真,

(2)　I の定義から　$I(q)=$偽,

(3)　(2)より　　　$I(\sim q)=$真,

(4)　(3)より　　　$I(\sim q\vee q\wedge r)=$真,

(5)　(1),(4)より　$I(p\leftrightarrow\sim q\vee q\wedge r)=$真,

従って、$\models_I p\leftrightarrow\sim q\vee q\wedge r$ である。また、前節1.4(11)の式については、同様にして $I(p\to(\sim(q\wedge r)\to s))=$偽, すなわち $\not\models_I p\to(\sim(q\wedge r)\to s)$ である。

解釈というものを用いると、特別な性質を持つ命題論理式をいろいろと考えることができる。例えば、結合子 $\sim,\wedge,\vee,\oplus,\to,\leftrightarrow$ の真理関数的性質によって常に真となる命題のパターンというのは、次の定義に定める恒真式にほかならない。

定義1.10　命題論理式αに関して、

(i) どんな解釈Iを与えても $\models_I\alpha$ となる時、αは**恒真**(valid)であるといい $\models\alpha$ と書く。また、恒真な命題論理式を**恒真式**(tautology)という。

(ii) どんな解釈Iを与えても $\not\models_I\alpha$ となる時、αは**充足不能**(unsatisfiable)または**矛盾**(inconsistent)であるという。また、矛盾な命題論理式を**矛盾式**(contradiction)という。

(iii) ある解釈Iに対して $\models_I\alpha$ となる時、αは**充足可能**(satisfiable)または**無矛盾**(consistent)であるという。

この定義から直接、次の命題が導かれる。[ここでは、“命題”という語を“容易に証明可能な正しい事実”というぐらいの意味で用いている。]

命題1.11　任意の命題論理式αに関して、

(i) αが恒真 $\Leftrightarrow$ $(\sim\alpha)$が矛盾,

(ii) αが矛盾 $\Leftrightarrow$ $(\sim\alpha)$が恒真,

(iii) αが無矛盾 $\Leftrightarrow$ αが矛盾でない

$\Leftrightarrow$ $(\sim\alpha)$が恒真でない

命題論理式が恒真であるかどうか，矛盾であるかどうかを調べるのは簡単である。命題論理式αが与えられたとき、αの中に現れる言明文字の個数は有限であって、$I(\alpha)$の値はこれらの言明文字への真理値の割当てにのみ依存している。従って、αがn個の異なる言明文字を含めば、これらの言明文字に対する2^n種類の真理値の割当てが可能で、2^n種類の場合を全て調べ尽くせば$I(\alpha)$がどんな解釈Iの下で真となり，どんな解釈Iの下で偽となるかが明らかになる。この“全ての場合を調べ尽くす表”を**真理表**(truth table)という。

例1．12　言明文字 p, q を含む命題論理式 $p\to((p\to q)\to q)$ の真理表は表1．1の通りである。この真理表の各行が p, q への真理値の割当て例に相当する。例えば、表1．1①の行は、

$I(p)=I(q)=$偽となるような解釈 I に対して

$I(p\to q)=$真，

$I((p\to q)\to q)=$偽，

$I(p\to((p\to q)\to q))=$真

となることを表す。この表から、$p\to((p\to q)\to q)$ は矛盾ではなく充足可能さらに恒真であることが分かる。従って、$\vDash p\to((p\to q)\to q)$ である。

表1．1　$p\to((p\to q)\to q)$ に対する真理表［簡単のため真を1で，偽を0で表す。］

	p	q	$p\to q$	$(p\to q)\to q$	$p\to((p\to q)\to q)$
①	0	0	1	0	1
②	0	1	1	1	1
③	1	0	0	1	1
④	1	1	1	1	1

例1．13　前節1．4(10)の命題論理式 $p\leftrightarrow\sim q\vee q\wedge r$ は3個の異なる言明文字 p, q, r を含むから、その恒真性／矛盾性を調べるためには、表1．2の様に $2^3=8$ 種類の解釈を全て調べ尽くせばよい。この真理表の各行が p, q, r への真理値の割当て例に相当する。例えば、⑥の行は例1．9で示された事実をまとめたものである。また、この真理表から $p\leftrightarrow\sim q\vee q\wedge r$ は矛盾でも恒真でもないが充足可能であることが分かる。［命題1．11より、矛盾でないものは充足可能である。］

表1．2　$p\leftrightarrow\sim q\vee q\wedge r$ に対する真理表［簡単のため真を1で，偽を0で表す。］

	p	q	r	$\sim q$	$q\wedge r$	$\sim q\vee q\wedge r$	$p\leftrightarrow\sim q\vee q\wedge r$
①	0	0	0	1	0	1	0
②	0	0	1	1	0	1	0
③	0	1	0	0	0	0	1
④	0	1	1	0	1	1	0
⑤	1	0	0	1	0	1	1
⑥	1	0	1	1	0	1	1
⑦	1	1	0	0	0	0	0
⑧	1	1	1	0	1	1	1

例1.14　命題論理式 α, β から構成される複合命題 $\sim(\alpha \to \sim\beta)$ を考える。 $I(\alpha)$と$I(\beta)$の真理値は、解釈Iがαとβの中の言明文字に真理値をどう割り当てるかによって様々に変わるが、基本的には、

① $I(\alpha)=$偽, $I(\beta)=$偽,

② $I(\alpha)=$偽, $I(\beta)=$真,

③ $I(\alpha)=$真, $I(\beta)=$偽,

④ $I(\alpha)=$真, $I(\beta)=$真

の４つの場合がある。［αやβの形によっては、これら４つの場合のうち一部しか起こり得ないこともある。］これら４つの場合について $\sim(\alpha \to \sim\beta)$ の真理値を調べれば表1.3が得られる。この真理表から、任意の解釈Iに対して、

$$I(\sim(\alpha \to \sim\beta))=\text{真} \Leftrightarrow I(\alpha)=I(\beta)=\text{真}$$
$$\Leftrightarrow I(\alpha \wedge \beta)=\text{真}$$

であること，すなわち

(6) $\sim(\alpha \to \sim\beta)$ は $\alpha \wedge \beta$ と全く同じ真理関数的振る舞いをする

ことが分かる。さらに、同様の方法／意味で、

(7) $\sim\alpha \to \beta$ は $\alpha \vee \beta$ と全く同じ真理関数的振る舞いをし，

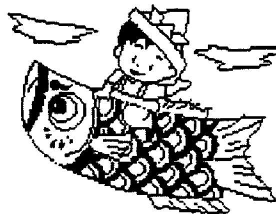
(8) $(\alpha \to \beta) \wedge (\beta \to \alpha)$ は $\alpha \leftrightarrow \beta$ と全く同じ真理関数的振る舞いをし，

(9) $\sim(\alpha \leftrightarrow \beta)$ は $\alpha \oplus \beta$ と全く同じ真理関数的振る舞いをする

ことが分かる。

表1.3　$\sim(\alpha \to \sim\beta)$ に対する真理表［簡単のため真を１で，偽を０で表す。］

	α	β	$\sim\beta$	$\alpha \to \sim\beta$	$\sim(\alpha \to \sim\beta)$
①	0	0	1	1	0
②	0	1	0	1	0
③	1	0	1	1	0
④	1	1	0	0	1



1．6　論理的帰結

いくつかの事実から別の事実を導き出す推論は様々なところで行われているが、そのうち多くのものが論理的に正しいものと考えられる。これらの論理的に正しい推論のうち特に1．4節の冒頭で述べた“結合子 $\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ の真理関数的性質によって常に正しくなる推論パターン”というのは、解釈の考え方に基づいて次の様に形式化できる。

定義1．15　命題論理式の集合Γと命題論理式αに関して、Γ内の全ての命題論理式を真にする解釈Iの下で常に$I(\alpha)$が真となるならば、αはΓの**論理的帰結**(logical consequence)であるといい $\Gamma \models \alpha$ と書く。 特にΓが有限で $\Gamma=\{\beta_1, \beta_2, \ldots, \beta_n\}$ の時、$\Gamma \models \alpha$ の代わりに $\beta_1, \beta_2, \ldots, \beta_n \models \alpha$ と略記する。

有限個の事実から別の事実を導き出す推論が真理関数的に正しいかどうか，すなわちΓが有限の場合に $\Gamma \models \alpha$ であるかどうかを判定する問題は、次の定理によって恒真性判定の問題に帰着できる。

定理1．16(⊨に関する演繹定理,deduction theorem)　任意の命題論理式 α, β と命題論理式の集合Γに関して、

$$\Gamma \cup \{\alpha\} \models \beta \quad \Leftrightarrow \quad \Gamma \models \alpha \rightarrow \beta$$

例1．17　定理1．16により

$$p,\ p \rightarrow q \models q \quad \Leftrightarrow \quad p \models (p \rightarrow q) \rightarrow q$$
$$\Leftrightarrow \quad \models p \rightarrow ((p \rightarrow q) \rightarrow q)$$

であり、また例1．12により $\models p \rightarrow ((p \rightarrow q) \rightarrow q)$ であることが分かるから、$p, p \rightarrow q \models q$, すなわち qは $\{p,\ p \rightarrow q\}$ の論理的帰結である。

例1．18　定理1．16により

$$q,\ p \rightarrow q \models p \quad \Leftrightarrow \quad q \models (p \rightarrow q) \rightarrow p$$
$$\Leftrightarrow \quad \models q \rightarrow ((p \rightarrow q) \rightarrow p)$$

であり、また表1．4の真理表から $q \rightarrow ((p \rightarrow q) \rightarrow p)$ が恒真式でないことが分かるから、pは $\{q,\ p \rightarrow q\}$ の論理的帰結ではない。従って、q と $p \rightarrow q$ からpを導き出す推論は真理関数的に何の根拠もない推論である。

表１．４　$q\to((p\to q)\to p)$ に対する真理表［簡単のため真を１で，偽を０で表す。］

p	q	$p\to q$	$(p\to q)\to p$	$q\to((p\to q)\to p)$
0	0	1	0	1
0	1	1	0	0
1	0	0	1	1
1	1	1	1	1

１．７　命題論理式の代数

任意の命題論理式に対して、それと（１.５節(6)～(9)の意味で）全く同一の真理関数的振る舞いをする命題論理式は無数にある。［実際、αは $\alpha\wedge\alpha,(\alpha\wedge\alpha)\wedge\alpha,((\alpha\wedge\alpha)\wedge\alpha)\wedge\alpha,\ \ldots$ と全く同一の真理関数的振る舞いをする。］従って、命題論理式が与えられたとき、それと同一の真理関数的振る舞いをしてより単純なものに変形する手法があれば有用である。特に、算術式の変形に類似したものだと親しみ易くて使いやすい。

そこで、この節では命題論理式の変形手法について調べる。まず、“同一の真理関数的振る舞いをする”というのを解釈の考えを用いて形式化しよう。

定義１.１９　全ての解釈 Iについて $I(\alpha)=I(\beta)$ となる時、２つの命題論理式 α,β は**論理的に同値**(logically equivalent)であるといい $\alpha\equiv\beta$ と書く。

算術では、等式の中の変数に式を同時に代入して別の等式を得たり、算術式の中の部分式をそれと等値なもので置き換えて式を変形することができる。例えば、等式

(1) $x^2-1=(x+1)(x-1)$

内の xの代わりに $x+1$ を使って等式

(2) $(x+1)^2-1=((x+1)+1)((x+1)-1)$

を得ることができ、また、この(2)の右辺内の部分式を

(3) $(x+1)+1=x+2,$

(4) $(x+1)-1=x$

を用いて置き換えて等式

(5) $((x+1)+1)((x+1)-1)=(x+2)x$

を得ることができる。さらに、等号の推移性（すなわち、$a=b,\ b=c$ から $a=c$ を導ける性質）を用いて、(2)と(5)から等式

(6)　$(x+1)^2-1=(x+2)x$
を導くことができる。

論理的同値の関係≡についても同様の変形が可能である。すなわち、次の定理1.20は算術の場合の(2)と(5)から(6)を導く推論を，定理1.21は算術の場合の(1)から(2)を導く推論を，定理1.22は算術の場合の(3)と(4)から(5)を導く推論を可能にする。

定理1.20　任意の命題論理式 α, β, γ に対して、
(i)　$\alpha\equiv\alpha$,
(ii)　$\alpha\equiv\beta$　ならば　$\beta\equiv\alpha$,
(iii)　$\alpha\equiv\beta, \beta\equiv\gamma$　ならば　$\alpha\equiv\gamma$

定理1.21(代入法則,principle of substitution)　2つの命題論理式 α, β が論理的に同値なとき、αとβの中に現れる言明文字pを全て命題論理式γで置き換えて得られる命題論理式 $\alpha[\gamma/p]$ と $\beta[\gamma/p]$ も論理的に同値である。［αやβの中にはpが現れなくてもよい。αの中にpが現れなければ $\alpha[\gamma/p]$ はαと同一の式になるだけである。］

定理1.22(置き換え法則,principle of replacement)　任意の命題論理式αに対して、“一部分（の命題論理式）をそれと論理的に同値な式で置き換える”という操作によってαから得られる式は、元の式αと論理的に同値である。

例1.23　基本的な等式
(7)　$q\wedge(p\vee\sim p)\equiv q$　(単位元律),
(8)　$(p\vee\sim p)\vee q\equiv p\vee\sim p$　(単位元律),
(9)　$(p\wedge q)\vee(p\wedge r)\equiv p\wedge(q\vee r)$　(分配律)
と定理1.20～22を用いて、任意の命題論理式 α, β に対して等式
(10)　$\alpha\vee(\alpha\wedge\beta)\equiv\alpha$　(吸収律)
を導くことができる。［もちろん、(7)～(9)は真理表を用いて容易に確かめることができる。また、等式 $p\vee(p\wedge q)\equiv p$ を真理表で確かめてこれに定理1.21を適用することによって、(10)を導くこともできる。］　まず、(7)式のqをαで置き換えることにより定理1.21を適用して
(11)　$\alpha\wedge(p\vee\sim p)\equiv\alpha$
が得られる。これに定理1.20(ii)を適用すれば
(12)　$\alpha\equiv\alpha\wedge(p\vee\sim p)$
となるから、この(12)式を基に定理1.22を適用して
(13)　$\alpha\vee(\alpha\wedge\beta)\equiv(\alpha\wedge(p\vee\sim p))\vee(\alpha\wedge\beta)$

が得られる。次に、(9)式の p, q, r をそれぞれ $\alpha, (p\vee\sim p), \beta$ で置き換え、また(8)式の q を β で置き換えることにより定理１．２１を適用して、それぞれ

(14) $(\alpha\wedge(p\vee\sim p))\vee(\alpha\wedge\beta)\equiv\alpha\wedge((p\vee\sim p)\vee\beta)$,

(15) $(p\vee\sim p)\vee\beta\equiv p\vee\sim p$

が得られる。さらに、(15)式を基に定理１．２２を適用して

(16) $\alpha\wedge((p\vee\sim p)\vee\beta)\equiv\alpha\wedge(p\vee\sim p)$

となるから、(13), (14), (16), (11)式に定理１．２０(iii)を適用して所要の(10)式が導かれる。以上の変形手順を算術式の変形に類似した形で書けば、次の様にまとめられる。

$\alpha\vee(\alpha\wedge\beta)\equiv(\alpha\wedge(p\vee\sim p))\vee(\alpha\wedge\beta)$	(7), 代入法則, ≡の対称性；置き換え法則
$\equiv\alpha\wedge((p\vee\sim p)\vee\beta)$	(9), 代入法則
$\equiv\alpha\wedge(p\vee\sim p)$	(8), 代入法則, 置き換え法則
$\equiv\alpha$	(7), 代入法則

先の例１．２３では定理１．２１～２２を適用するための基本等式として(7)～(9)を用いたが、一般には次の定理に挙げられている等式を基本的なものとして定理１．２０～２２と合わせて用いる。

定理１．２４　言明文字 p, q, r に関して次の(i)～(xv)が成り立つ。但し、ここでは $p\vee\sim p$ を true で，$p\wedge\sim p$ を false で表す。

(i) $p\vee p\equiv p$,	$p\wedge p\equiv p$	(べき等律)
(ii) $(p\vee q)\vee r\equiv p\vee(q\vee r)$,	$(p\wedge q)\wedge r\equiv p\wedge(q\wedge r)$	(結合律)
(iii) $p\vee q\equiv q\vee p$,	$p\wedge q\equiv q\wedge p$	(交換律)
(iv) $p\vee(q\wedge r)\equiv(p\vee q)\wedge(p\vee r)$,	$p\wedge(q\vee r)\equiv(p\wedge q)\vee(p\wedge r)$	(分配律)
(v) $q\vee$ true ≡ true,	$q\wedge$ true ≡ q	(単位元律)
(vi) $q\vee$ false ≡ q,	$q\wedge$ false ≡ false	(単位元律)
(vii) $q\vee\sim q\equiv$ true,	$q\wedge\sim q\equiv$ false	(補元律)
(viii) ～true ≡ false,	～false ≡ true	(補元律)
(ix) $\sim(\sim p)\equiv p$,		(対合律)
(x) $\sim(p\vee q)\equiv\sim p\wedge\sim q$,	$\sim(p\wedge q)\equiv\sim p\vee\sim q$	(ドモルガンの法則)
(xi) $p\rightarrow q\equiv\sim p\vee q$		(→の除去，導入)
(xii) $p\wedge q\equiv\sim(p\rightarrow\sim q)$		(∧の除去，導入)
(xiii) $p\vee q\equiv\sim p\rightarrow q$		(∨の除去，導入)
(xiv) $p\leftrightarrow q\equiv(p\rightarrow q)\wedge(q\rightarrow p)$		(↔の除去，導入)
(xv) $p\oplus q\equiv\sim(p\leftrightarrow q)$		(⊕の除去，導入)

1．8　標準形

どんな命題論理式も論理的同値性を保ったままある標準形に変形できるなら、様々な議論の簡素化につながる。なぜなら、命題論理式の真理関数的な面の議論では、標準形のものを考えるだけで十分になるからである。

そこでこの節では、前節1．7で論じた変形手法を基にして、任意の命題論理式を3種類の標準形に変形できることを示す。最初の標準形とそれへの変形手順は次の定理に示される。［これは一般には標準形と呼ばれていない。］

定理1．25(含意と否定への還元)　命題論理式が任意に与えられたとき、論理的同値性を保ったままそれを含意と否定以外の結合子を含まないものに変形できる。具体的には、前節の定理1．24(xii)～(xv)と定理1．20～22を用いて次の(i)～(iii)の順に変形を行えばよい。

(i) 定理1．24(xv)と定理1．20～22を用いて、結合子として $\sim, \to, \wedge, \vee, \leftrightarrow$ だけを含むものに変形する。(⊕の除去)

(ii) 定理1．24(xiv)と定理1．20～22を用いて、結合子として $\sim, \to, \wedge, \vee$ だけを含むものに変形する。(↔の除去)

(iii) 定理1．24(xii), (xiii)と定理1．20～22を用いて、結合子として $\sim, \to$ だけを含むものに変形する。(∧, ∨の除去)

もちろん、この定理1．25と同様にして連言と否定への還元や選言と否定への還元も可能である。しかし、同値と否定への還元，排他的選言と否定への還元や連言と選言への還元はどれも可能とは限らない。

例1．26　命題論理式 $\sim p \oplus (p \vee q)$ は定理1．25で述べられた手順により次の様に変形できる。

$\sim p \oplus (p \vee q)$

$\equiv \sim(\sim p \leftrightarrow (p \vee q))$　　(手順(i))

$\equiv \sim((\sim p \to (p \vee q)) \wedge ((p \vee q) \to \sim p))$　　(手順(ii))

$\equiv \sim((\sim p \to (\sim p \to q)) \wedge ((\sim p \to q) \to \sim p))$　　(手順(iii))

$\equiv \sim(\sim((\sim p \to (\sim p \to q)) \to \sim((\sim p \to q) \to \sim p)))$　　(手順(iii))

また、定理1．25の手順にこだわらなければ次の様にも変形できる。

$\sim p \oplus (p \vee q)$

$\equiv \sim(\sim p \leftrightarrow (p \vee q))$　　⊕の除去(定理1．24(xv))

$\equiv \sim((\sim p \to (p \vee q)) \wedge ((p \vee q) \to \sim p))$　　↔の除去(定理1．24(xiv))

$\equiv \sim((p\vee(p\vee q))\wedge(\sim(p\vee q)\vee\sim p))$	→の除去(定理１．２４(xi),(xiii))
$\equiv \sim((p\vee q)\wedge((\sim p\wedge\sim q)\vee\sim p))$	結合律,べき等律；ドモルガンの法則
$\equiv \sim((p\vee q)\wedge\sim p)$	吸収律(すなわち１．７節(10)式)
$\equiv \sim(p\wedge\sim p\vee q\wedge\sim p)$	分配律
$\equiv \sim(q\wedge\sim p)$	補元律,単位元律
$\equiv \sim q\vee p$	ドモルガンの法則
$\equiv q\rightarrow p$	→の導入(定理１．２４(xi))

残りの２つの標準形とそれへの変形手順は次の２つの定理によって示される。

定理１．２７(選言標準形)　命題論理式が任意に与えられたとき、論理的同値性を保ちながらそれを

(1) $\alpha_1\vee\alpha_2\vee\cdots\vee\alpha_n$

ここで、$n\geqq 1$であり、各α_iは言明文字,言明文字の否定,またはそれら(まとめてリテラル,leteral,という)を連言結合子∧で結合したものである

なる形(**選言標準形**,disjunctive normal form,という)のものに変形できる。具体的には、前節の定理１．２４と定理１．２０〜２２を用いて次の(i)〜(v)の順に変形を行えばよい。［途中で結合律,交換律,べき等律,補元律,単位元律を用いればより簡単な選言標準形が得られる。］

(i) 定理１．２４(xv)と定理１．２０〜２２を用いて、結合子として ～,∧,∨,→,↔ だけを含むものに変形する。(⊕の除去)

(ii) 定理１．２４(xiv)と定理１．２０〜２２を用いて、結合子として ～,∧,∨,→ だけを含むものに変形する。(↔の除去)

(iii) 定理１．２４(xi),(xiii)と定理１．２０〜２２を用いて、結合子として ～,∧,∨ だけを含むものに変形する。(→の除去)

(iv) 対合律,ドモルガンの法則と定理１．２０〜２２を用いて、否定結合子～を全て言明文字の直前にもってくる。

(v) 交換律,結合律,分配律 ($p\wedge(q\vee r)\equiv p\wedge q\vee p\wedge r$ の方) と定理１．２０〜２２を用いて、選言標準形に変形する。

定理１．２８(連言標準形)　命題論理式が任意に与えられたとき、論理的同値性を保ちながらそれを

(2) $\alpha_1\wedge\alpha_2\wedge\cdots\wedge\alpha_n$

ここで、$n\geqq 1$であり、各α_iは言明文字,言明文字の否定,またはそれらを選言結合子∨で結合したものである,

なる形(**連言標準形**,conjunctive normal form,という)のものに変形できる。具体的には、前節の定

理１．２４と定理１．２０～２２を用いて次の(i)～(v)の順に変形を行えばよい。［途中で結合律，交換律，べき等律，補元律，単位元律を用いればより簡単な連言標準形が得られる。］

(i) 定理１．２４(xv)と定理１．２０～２２を用いて、結合子として $\sim, \wedge, \vee, \rightarrow, \leftrightarrow$ だけを含むものに変形する。（$\oplus$の除去）

(ii) 定理１．２４(xiv)と定理１．２０～２２を用いて、結合子として $\sim, \wedge, \vee, \rightarrow$ だけを含むものに変形する。（$\leftrightarrow$の除去）

(iii) 定理１．２４(xi)，(xiii)と定理１．２０～２２を用いて、結合子として $\sim, \wedge, \vee$ だけを含むものに変形する。（$\rightarrow$の除去）

(iv) 対合律，ドモルガンの法則と定理１．２０～２２を用いて、否定結合子～を全て言明文字の直前にもってくる。

(v) 交換律，結合律，分配律（$p\vee(q\wedge r)\equiv(p\vee q)\wedge(p\vee r)$ の方）と定理１．２０～２２を用いて、連言標準形に変形する。

例１．２９　例１．２６で考えた命題論理式 $\sim p\oplus(p\vee q)$ は定理１．２７で述べられた手順により次の様に選言標準形に変形される。

$$
\begin{aligned}
&\sim p\oplus(p\vee q) \\
\equiv\ &\sim(\sim p\leftrightarrow(p\vee q)) && \text{(手順(i))} \\
\equiv\ &\sim((\sim p\rightarrow(p\vee q))\wedge((p\vee q)\rightarrow\sim p)) && \text{(手順(ii))} \\
\equiv\ &\sim((p\vee(p\vee q))\wedge(\sim(p\vee q)\vee\sim p)) && \text{(手順(iii))} \\
\equiv\ &\sim((p\vee q)\qquad\wedge(\sim(p\vee q)\vee\sim p)) && \text{べき等律，結合律による簡単化} \\
\equiv\ &\sim(p\vee q)\vee\sim(\sim(p\vee q)\vee\sim p) && \text{(手順(iv))} \\
\equiv\ &(\sim p\wedge\sim q)\vee(\sim\sim(p\vee q)\wedge\sim\sim p) \\
\equiv\ &(\sim p\wedge\sim q)\vee((p\vee q)\wedge p) \\
\equiv\ &(\sim p\wedge\sim q)\vee((p\wedge p)\vee(q\wedge p)) && \text{(手順(v))} \\
\equiv\ &(\sim p\wedge\sim q)\vee p\vee(q\wedge p) && \text{べき等律による簡単化}
\end{aligned}
$$

また、同じ命題論理式 $\sim p\oplus(p\vee q)$ は定理１．２８で述べられた手順により次の様に連言標準形に変形される。［この変形によって得られる式は選言標準形にもなっている。］

$$
\begin{aligned}
&\sim p\oplus(p\vee q) \\
\equiv\ &(\sim p\wedge\sim q)\vee((p\vee q)\wedge p) && \text{(手順(i)～(iv);選言標準形の場合と同様)} \\
\equiv\ &(\sim p\vee((p\vee q)\wedge p))\wedge(\sim q\vee((p\vee q)\wedge p)) && \text{(手順(v))} \\
\equiv\ &(\sim p\vee(p\vee q))\wedge(\sim p\wedge p)\wedge(\sim q\vee(p\vee q))\wedge(\sim q\wedge p) \\
\equiv\ &\sim q\vee p && \text{結合律，交換律，補元律，単位元律による簡単化}
\end{aligned}
$$

選言標準形と連言標準形は次のような利点を持つ。

(i) 選言標準形の命題論理式，特に結合律,交換律,べき等律,補元律と単位元律により簡単化されたものは、どんな解釈の下で真になるかがはっきりしている。すなわち、各α_iをリテラルまたはリテラルを連言結合子∧で結合したものとした時、

選言標準形の命題論理式 $\alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_n$ が真

⇔ $\alpha_1, \alpha_2, \cdots, \alpha_n$ の中のどれかが真

⇔ ある $i \in \{1,2, \ldots, n\}$ について、α_i 内のリテラルが全て真

である。従って、選言標準形の命題論理式を真にする解釈があるかどうか，すなわち充足可能かどうかの判定もすぐにできる。実際、その式が結合律,交換律,補元律と単位元律によりfalseに簡単化できれば矛盾であり、そうでなければ充足可能であると判断できる。

(ii) 連言標準形の命題論理式，特に結合律,交換律,べき等律,補元律と単位元律により簡単化されたものは、どんな解釈の下で偽になるかがはっきりしている。すなわち、各α_iをリテラルまたはリテラルを選言結合子∨で結合したものとした時、

選言標準形の命題論理式 $\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n$ が偽

⇔ $\alpha_1, \alpha_2, \cdots, \alpha_n$ の中のどれかが偽

⇔ ある $i \in \{1,2, \ldots, n\}$ について、α_i 内のリテラルが全て偽

である。従って、連言標準形の命題論理式を偽にする解釈があるかどうか，すなわち恒真でないかどうかの判定もすぐにできる。実際、その式が結合律,交換律,補元律と単位元律によりtrueに簡単化できれば恒真であり、そうでなければ恒真でないと判断できる。

例1.30 命題論理式

(3) $\sim p \oplus (p \vee q)$

は例1.29により選言標準形

$(\sim p \wedge \sim q) \vee p \vee (q \wedge p)$

に変形できるが、これを結合律,交換律,補元律と単位元律により簡単化してもfalseに変形できない。従って(3)式は充足可能であって矛盾ではない。実際、この選言標準形から、例えばpに真を割り当てる解釈の下で(3)式が真になることが分かる。また、例1.29より(3)式は連言標準形

$\sim q \vee p$

にも変形できるが、これを結合律,交換律,補元律と単位元律により簡単化してもtrueに変形できない。従って(3)式は恒真ではない。実際、この連言標準形から、$\sim q$とpに偽を（すなわちpに偽，qに真を）割り当てる解釈の下で(3)式が偽になることが分かる。

例1.31 例1.12から命題論理式

$p \to ((p \to q) \to q)$

が恒真であることが分かるが、これは次の様に連言標準形への変形によっても示すことができる。

$p \to ((p \to q) \to q)$
$\equiv \sim p \vee \sim (p \to q) \vee q$　　→の除去
$\equiv \sim p \vee \sim (\sim p \vee q) \vee q$　　→の除去
$\equiv \sim p \vee (p \wedge \sim q) \vee q$　　ドモルガンの法則,対合律
$\equiv (\sim p \vee p \vee q) \wedge (\sim p \vee \sim q \vee q)$　　分配律
$\equiv$ true∧true　　補元律,単位元律
$\equiv$ true　　単位元律

1.9　命題の真理関数的正しさと同値性

1.4節の冒頭で述べたように、命題論理学では、個々の原子命題の構造や真偽性については全く関心がなく、どんな複合命題のパターンがどんな時に結合子 ～,∧,∨,⊕,→,↔ の真理関数的性質によって真となるか，どんな推論パターンが結合子 ～,∧,∨,⊕,→,↔ の真理関数的性質によって常に正しくなるかを調べる。そのために、命題の中にある個々の原子命題を文字で置き換えることによって命題の真理関数的構造を抽出し、以後1.8節までこの様な構造(すなわち命題論理式)の恒真性,論理的帰結性や同値性について議論した。

従って、本来の意図からすると、命題から抽出された構造が恒真な命題論理式なら、その構造を持った元の命題は結合子 ～,∧,∨,⊕,→,↔ の真理関数的性質によって常に真となると考えるのが自然である。この考えに基づいて命題や推論の真理関数的正しさ，さらには命題間の真理関数的同値性についての形式化を行うと次の様になる。

定義1.32　(i) 命題$\mathcal{A}$の真理関数的構造を抽出して命題論理式αが得られるとする。この時、もしαが恒真なら元の命題$\mathcal{A}$は**真理関数的に真**(truth-functionally true)であるといい、またαが矛盾なら$\mathcal{A}$は**真理関数的に偽**(truth-functionally false)であるという。

(ii)　命題 $\mathcal{A}, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ の真理関数的構造を抽出して、それぞれ命題論理式 $\alpha, \beta_1, \beta_2, \ldots, \beta_n$ が得られるとする。[もちろん、$\mathcal{A}, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ 内で同一の原子命題は全て同一の言明文字で置き換えられるものとする。]　この時、もしαが $\beta_1, \beta_2, \ldots, \beta_n$ の論理的帰結，すなわち $\beta_1, \beta_2, \ldots, \beta_n \models \alpha$ であるなら、元の命題について$\mathcal{A}$は $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ の**真理関数的帰結**(truth-functional consequence)であるという。

(iii) 命題 $\mathcal{A}, \mathcal{B}$ の真理関数的構造を抽出して、それぞれ命題論理式 α, β が得られるとする。[もちろん、$\mathcal{A}, \mathcal{B}$ 内で同一の原子命題は全て同一の言明文字で置き換えられるものとする。]
この時、もしαとβが論理的に同値，すなわち $\alpha \equiv \beta$ であるなら、元の2つの命題$\mathcal{A}$と$\mathcal{B}$は**真理関数的に同値**(truth-functionally equivalent)であるという。

例１．３３　１．３節(3)の命題

xを未知数とする方程式　$ax+b=0$　が解を持つための必要十分条件は、

$a\neq 0$　または　（$a=0$　かつ　$b=0$）

である

は、各原子命題の意味を考えて初めて正しいと認められるものであり、真理関数的に真でも偽でもない。なぜなら、この命題の真理関数的構造を抽出すると例１．５に示されるように命題論理式

$$p\leftrightarrow\sim q\vee(q\wedge r)$$

が得られるが、これは例１．１３で見た様に恒真でも矛盾でもないからである。

例１．３４　２つの命題

(1) 公園は、晴れてて寒くない日は人でいっぱいになるが、晴れてても寒い日は人でいっぱいにはならない。もちろん、晴れてなければ公園は人でいっぱいにならない，

と

(2) 晴れてて寒くない日，かつその日に限り、公園は人でいっぱいになる，

は真理関数的に同値である。なぜなら、結合詞を記号で置き換え、さらに

“晴れている”　という原子命題を言明文字pで，

“寒い”　という原子命題を言明文字qで，

“公園は人でいっぱい”という原子命題を言明文字rで

置き換えることにより、(1),(2)の真理関数的構造としてそれぞれ命題論理式

(3) $(p\wedge\sim q\rightarrow r)\wedge(p\wedge q\rightarrow\sim r)\wedge(\sim p\rightarrow\sim r)$,

(4) $p\wedge\sim q\leftrightarrow r$

が得られるが、これらの式(3),(4)は論理的に同値であるからである。実際、

$(p\wedge q\rightarrow\sim r)\wedge(\sim p\rightarrow\sim r)$	
$\equiv(\sim(p\wedge q)\vee\sim r)\wedge(p\vee\sim r)$	→の除去
$\equiv(\sim p\vee\sim q\vee\sim r)\wedge(p\vee\sim r)$	ドモルガンの法則
$\equiv((\sim p\vee\sim q)\wedge p)\vee\sim r$	分配律
$\equiv(\sim p\wedge p\vee\sim q\wedge p)\vee\sim r$	分配律
$\equiv(\mathrm{false}\vee\sim q\wedge p)\vee\sim r$	補元律
$\equiv\ \sim r\vee p\wedge\sim q$	単位元律,交換律
$\equiv\ r\rightarrow p\wedge\sim q$	→の導入

であるから、置き換え法則により

$(p\wedge\sim q\rightarrow r)\wedge(p\wedge q\rightarrow\sim r)\wedge(\sim p\rightarrow\sim r)$	
$\equiv(p\wedge\sim q\rightarrow r)\wedge(r\rightarrow p\wedge\sim q)$	置き換え法則
$\equiv\ p\wedge\sim q\leftrightarrow r$	↔の導入

となる。

例1.35(C.-L.Chang&R.C.-T.Lee［6］)　3つの命題

(5) 公定歩合が上がれば株価は下がる,

(6) 株価が下がればほとんどの人は不幸になる,

(7) 公定歩合が上がる

から命題

(8) ほとんどの人が不幸になる

を導く推論は真理関数的に正しいものである。［すなわち、(8)は(5)〜(7)の真理関数的帰結である。］なぜなら、結合詞を記号で置き換え、さらに

“公定歩合が上がる”　という原子命題を言明文字 p で,
“株価が下がる”　という原子命題を言明文字 q で,
“ほとんどの人が不幸になる”という原子命題を言明文字 r で

置き換えることにより、(5)〜(8)の真理関数的構造としてそれぞれ命題論理式

(9) $p \to q$,

(10) $q \to r$,

(11) p,

(12) r

が得られるが、(8)の構造である(12)式が残りの(9)〜(11)式の論理的帰結，すなわち

(13) $p \to q,\ q \to r,\ p \vDash r$

であるからである。実際、定理1.16(⊨に関する演繹定理)によれば(13)は“$(p \to q) \to ((q \to r) \to (p \to r))$ が恒真”であること，すなわち

(14) $\vDash (p \to q) \to ((q \to r) \to (p \to r))$

と同等であり、(14)は連言標準形への変形手続きで

$$
\begin{aligned}
& (p \to q) \to ((q \to r) \to (p \to r)) & \\
\equiv\ & \sim(p \to q) \vee \sim(q \to r) \vee \sim p \vee r & \text{→の除去} \\
\equiv\ & \sim(\sim p \vee q) \vee \sim(\sim q \vee r) \vee \sim p \vee r & \text{→の除去} \\
\equiv\ & (p \wedge \sim q) \vee (q \wedge \sim r) \vee \sim p \vee r & \text{ドモルガンの法則} \\
\equiv\ & (p \vee q \vee \sim p \vee r) \wedge (p \vee \sim r \vee \sim p \vee r) & \text{分配律} \\
& \wedge (\sim q \vee q \vee \sim p \vee r) \wedge (\sim q \vee \sim r \vee \sim p \vee r) & \\
\equiv\ & \mathrm{true} & \text{交換律,補元律,単位元律}
\end{aligned}
$$

となることにより示される。もちろん、(14)は表1.5の真理表によっても説明できる。

表１．５　$(p\to q)\to((q\to r)\to(p\to r))$ に対する真理表［簡単のため真を１で,偽を０で表す。］

p	q	r	$p\to r$	$q\to r$	$(q\to r)\to(p\to r)$	$p\to q$	$(p\to q)\to((q\to r)\to(p\to r))$
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	1	0	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	1	0	0	1
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	1	1	1	1	1	1

例１．３６　　１つの命題

(15) もし飛行機事故に遭遇すれば、運がよくなければ助からない

から命題

(16) 飛行機事故に遭遇しても、運がよければ助かる

を導く推論は真理関数的に正しいものではない。［すなわち、(16)は(15)の真理関数的帰結ではない。］なぜなら、結合詞を記号で置き換え、さらに

“飛行機事故に遭遇する”　　　　という原子命題を言明文字 p で,

“(事故に遭遇した人の)運がよい”という原子命題を言明文字 q で,

“(事故に遭遇した人が)助かる”　という原子命題を言明文字 r で

置き換えることにより、(15),(16)の真理関数的構造としてそれぞれ命題論理式

(17)　$p\to(\sim q\to\sim r)$,

(18)　$p\to(q\to r)$

が得られるが、(16)の構造である(18)式が残りの(17)式の論理的帰結でないからである。実際、p と q に真を割り当て r に偽を割り当てる解釈の下では(17)式は真になるが(18)式は偽になってしまう。

演習問題１

1．1　次の文の中で、下線部の結合詞“～というのを言い換えると、…ということになる”は、真理関数的か？

新潟市の市長が男性でない<u>というのを言い換えると</u>、新潟市の市長が女性<u>ということになる</u>。

1．2（L．チャイカ［4］）　日本語の言明

もし、完全雇用は経済が成長している時にのみ行われるのであり、かつ経済が成長している場合には物価水準が上昇するのであれば、完全雇用が行われていてしかも物価水準が上昇しない、といったことはない

について、

(1)　この言明は、1．3節で示した論理構造明確化の手順によって、どの様に変形されてゆくか？

(2)　この言明の論理構造は、どんな命題論理式で表されるか？

1．3　命題1．11を証明せよ。

1．4　次の各々の命題論理式について、恒真かどうか，矛盾かどうか，充足可能かどうかを調べよ。

(1)　$((p\to q)\to q)\to q$

(2)　$p\wedge\sim(p\vee q)$

(3)　$(p\leftrightarrow q)\leftrightarrow(p\leftrightarrow(q\leftrightarrow p))$

(4)　$((p\oplus q)\oplus p)\oplus\sim q$

(5)　$(p\to q)\to((q\to r)\to(p\to r))$

1．5　1．5節の(7)～(9)の事柄を確かめよ。すなわち、任意の解釈 I に対して、

①　$I(\alpha\vee\beta)=$真　⇔　$I(\sim\alpha\to\beta)=$真

②　$I(\alpha\leftrightarrow\beta)=$真　⇔　$I((\alpha\to\beta)\wedge(\beta\to\alpha))=$真

③　$I(\alpha\oplus\beta)=$真　⇔　$I(\sim(\alpha\leftrightarrow\beta))=$真

となることを確かめよ。

1．6　定理1．16を証明せよ。

1．7　定理１．２０～２２，定理１．２４を証明せよ。

1．8　命題論理式

$$(p \to q) \to (\sim p \to \sim q)$$

について、

(1)　真理表を作ることによって、恒真かどうかを判定せよ。

(2)　連言標準形に変形することによって、恒真かどうかを判定せよ。

1．9　上の問題１．４で考えた命題論理式(1)～(5)をそれぞれ選言標準形，連言標準形に変形せよ。

1．10（C.-L.Chang＆R.C.-T.Lee［６］）　３つの命題

① もし議会が新しい法案を可決するのを拒否すれば、ストライキは、１年以上続いてかつ会社の社長が退陣することがない限り終わらない，

② 議会が新しい法案を可決するのを拒否した，

③ ストライキは（始まったばかりで）まだ１年以上続いていない

から命題

④ ストライキは終わらない

を導く推論は真理関数的に正しいものであるか？　すなわち、④は①～③の真理関数的帰結であるか？

1．11　２つの命題

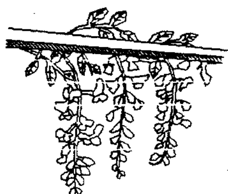
① すべての人間はいつかは死ぬ，

② ソクラテスは人間である

から命題

③ ソクラテスはいつかは死ぬ

を導く推論は真理関数的に正しいものであるか？　すなわち、③は①～②の真理関数的帰結であるか？



第2章　述語論理

前章では、日常会話の中に現れる言葉の中で真理関数的結合詞を論理的な言葉と考え、これらの結合詞の意味に基づく論理的推論を形式化した。しかし、実際の日常会話の中には、前章で形式化されていないが論理的と思える推論もかなり見受けられる。

そこで、この章では、真理関数的結合詞に加えて“すべての(every)”や“ある(some)”なども論理的な言葉と考え、これらの言葉の意味に基づく論理的推論の形式化を前章と同じ順序で進めてみよう。

2.1　命題論理の能力の限界

命題論理学では、個々の原子命題の構造や真偽性については全く関心がなく、どんな複合命題のパターンがどんな時に結合子 $\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ の真理関数的性質によって真となるか，どんな推論パターンが結合子 $\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ の真理関数的性質によって常に正しくなるかを調べる。従って、命題論理学には個々の原子命題の構造を分析する能力がなく、このために、論理的に正しいと思える推論でも真理関数的に正しくないことがある。実際、次に例示するように、全く新しい原子命題を導く推論は論理的に正しいと思えるものであっても正当化されない。

例2.1　2つの言明

(1) すべての人間はいつかは死ぬ，

(2) ソクラテスは人間である

から新しい原子言明

(3) ソクラテスはいつかは死ぬ

を導く推論は論理的に正しいものと考えられる。なぜなら、この推論の正しさは“人間”，“いつかは死ぬ”，“ソクラテス”という特定の言葉の意味によるものではなく、これら3つの言葉を別の言葉で置き換えて得られる推論，例えば、2つの言明

(4) すべての魚は泳げる，

(5) マンボウは魚である

から新しい原子言明

(6) マンボウは泳げる

を導く推論も(1)～(3)の場合と同様の理由で正しいと認められるからである。すなわち、(1)～(3)の推論も(4)～(6)の推論も２つの型式(schema)

(7) すべての□□は○○○，

(8) △△は□□である

から型式

(9) △△は○○○

を導く推論パターンを持っており、この様な推論パターン自体が正しいと言えるからである。

しかしながら、(3)は(1)と(2)の真理関数的帰結ではない。なぜなら、(1),(2),(3)の真理関数的構造を命題論理式の形で抽出しようとしても、命題論理の立場では(1),(2),(3)は互いに異なる原子命題としてしか考えられないから、各々を異なる言明文字で置き換えるしかない。そこで、(1)～(3)の真理関数的構造として例えばそれぞれ p, q, r が得られることになるが、明らかに、r は p と q の論理的帰結ではない（すなわち $p, q \not\models r$ である）。

例2.2　２つの原子言明

(10) W.A.Mozart はザルツブルグで生まれた，

(11) W.A.Mozart は作曲家である

から新しい言明

(12) ある作曲家はザルツブルグで生まれた

を導く推論は論理的に正しいものと考えられる。なぜなら、この推論の正しさは“W.A.Mozart”，“ザルツブルグで生まれた”，“作曲家”という特定の言葉の意味によるものではなく、これら３つの言葉を別の言葉で置き換えて得られる推論，例えば、２つの原子言明

(13) 鯨は海で生活する，

(14) 鯨は哺乳動物である

から新しい言明

(15) ある哺乳動物は海で生活する

を導く推論も(10)～(12)の場合と同様の理由で正しいと認められるからである。すなわち、(10)～(12)の推論も(13)～(15)の推論も２つの型式

(16) □□は○○○，

(17) □□は△△である

から型式

(18) ある△△は○○○

を導く推論パターンを持っており、この様な推論パターン自体が正しいと言えるのである。

しかしながら、先の例2.1(1)〜(3)の場合と同様の理由で、(12)は(10)と(11)の真理関数的帰結ではない。　すなわち、(10)〜(12)の真理関数的構造を命題論理式の形で抽出しようとしても結局(10)〜(12)の各々を異なる3つの言明文字で置き換えるしかなく、(10)〜(12)の真理関数的構造として例えばそれぞれ p, q, r が得られることになるが、明らかに、rはpと qの論理的帰結ではない（すなわち $p, q \not\models r$ である）。

2．2　限量詞

前節2.1では、論理的には正しいと思えるが真理関数的には正しくない推論の例を2つ挙げた。これらの推論パターン2.1節(7)〜(9)と(16)〜(18)はそれ自体論理的に正しいと思えるから、これら2つの推論パターンが命題論理で正しいとされない原因としては、2.1節(7)〜(9)もしくは(16)〜(18)の中に論理的だが命題論理で分析されていない言葉があると考えられる。

そこで、この様な（すなわち、真理関数的結合詞以外の論理的な）言葉を2.1節(7)〜(9)や(16)〜(18)の中から探し出してみよう。 まず、2.1節(7)〜(9)の中には“すべての”,主語の次に置く“は”と“である”と言う3つの言葉しか含まれていない。 この内“である”という言葉を省いても2.1節(7)〜(9)はやはり論理的に正しいと思え、また、“は”という助詞は単に2.1節(7)〜(9)が文に見える働きをするだけと考えられるから、2.1節(7)〜(9)を論理的に正しい推論にしているのは“すべての”という言葉である。同様に、2.1節(16)〜(18)を論理的に正しい推論にしているのは“ある(some)”という言葉である。

“すべての(every)”，“ある(some)”,もしくはこれらと同等の働きをする言葉を**限量詞**(quanti-fier)という。以下この節では、論理的だが命題論理で分析されていない言葉として、どの様な限量詞が日常会話の中に現れるか見てみよう。

全称限量詞．　“すべての(every)”,もしくはこれと同等の働きをする限量詞を特に**全称限量詞**(universal quantifier)という。全称限量詞は“考えられる全てのものについて何かある事柄が正しい”ことを主張する時に用いられるものである。従って、全称限量化された命題はこの主張が本当に正しい時に真となり、また、それ以外の時には偽となる。例えば、9つの言明

(1) すべての鳥は空を飛べる，

(2) あらゆる鳥は空を飛べる，

(3) それぞれの鳥は空を飛べる，

(4) 各々の鳥は空を飛べる，

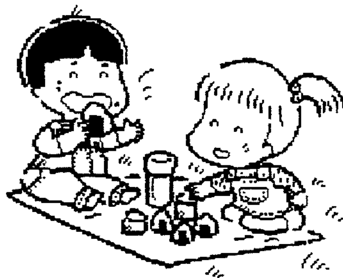
(5) 任意の鳥は空を飛べる，
(6) どの鳥も空を飛べる，
(7) いかなる鳥も空を飛べる，
(8) 鳥はどれも空を飛べる，
(9) 鳥は空を飛べる

は、どれも同じ命題を表す。［(9)の言明には全称限量詞が見当たらない。この様に、普遍的な原則を示すときには全称限量詞が省略されることもある。］もちろん、空を飛べない鳥としてペンギンや駝鳥が思い浮かぶので、この命題は偽である。

存在限量詞.　“ある(some)”，もしくはこれと同等の働きをする限量詞を特に**存在限量詞**(existential quantifier)という。存在限量詞は“ある事柄を満たすものが世の中に存在する”ことを主張するときに用いられるものである。従って、存在限量化された命題はこの主張が本当に正しい時に真となり、また、それ以外の時には偽となる。例えば、６つの言明

(10) ある県はチューリップを県花としている，
(11) 少なくとも１つの県ではチューリップが県花になっている，
(12) ある県の県花はチューリップである，
(13) チューリップを県花としている県が少なくとも１つ存在する，
(14) チューリップを県花としている県が実在する，
(15) チューリップを県花としている県がある

は、どれも同じ命題を表す。富山県と新潟県がチューリップを県花としているから、この命題はもちろん真である。



2.3 論理構造の明確化

命題論理では、例えば

(1) すべての実数 x について、$x(x-1)=0$ ならば $x=0$ または $x=1$ である

という言明は、真理関数的結合詞“ならば”と“または”を含むにもかかわらず、言明の最小単位(すなわち原子言明)として扱われる。なぜなら、(1)の言明の最も外側の論理的な言葉は限量詞の“すべての”であって、“ならば”でも“または”でもないからである。これに対し述語論理では、真理関数的結合詞のほかに限量詞も論理的な語として注目するから、(命題論理)より細かく言明を分析することになる。例えば、(1)の言明は“xは実数である”，“$x(x-1)=0$”，“$x=0$”，“$x=1$”という4つの構成要素に分解される。

当然、論理構造明確化の手順は命題論理の場合に比べて複雑になる。しかし、これに伴う手間としては、単に“言明をより細かく分解する”というものだけでなく、“限量詞が文中のどの名詞を修飾するかを明確に示す”というものも新たに生じる。[従って、形式化の際には、例えば(1)の基本構成要素“$x(x-1)=0$”を命題論理の場合のように1つの文字で置き換えることはできない。なぜなら、“$x(x-1)=0$”が全称限量詞で修飾されたもの x を含むからである。]

具体的な論理構造明確化の手順は、命題論理の場合の基本方針，すなわち

(i) 結合の曖昧さを除去するために括弧を用いる，

(ii) 記号で置き換えられていない結合詞の中で、最も外側のものから順に記号で置き換えてゆく，

(iii) 意味の同じ真理関数的結合詞を統一するために、否定，連言，選言，排他的選言，含意，同値の結合詞をそれぞれ $\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ という記号で置き換える

というものを拡張して得られる。まず、限量詞を新しく論理的な語と考えることに伴う変更を(ii)に加えなければならないが、このためには単に(ii)を

(II) 記号で置き換えられていない結合詞と限量詞の中で、最も外側のものから順に記号で置き換えてゆく

と変更するだけでよい。[元々の考え方は不変である。]

次に、(iii)に倣って限量詞も記号で置き換えなければならない。その際、単に同一の働きをする限量詞が多数存在するだけでなく、2.2節で見た様に限量詞の現れる位置も頭部，中間部，末尾と多様であるから、限量詞の現れる位置を固定する必要がある。また、限量詞で修飾された語は言明中の他の箇所から指示代名詞などで参照されるから、この参照の方式も統一する必要がある。そこで、全称と存在の限量詞をそれぞれ $\forall, \exists$ という記号(限量子，quantifier，という；“All”，“Exist”の頭文字を逆様にしたもの)で置き換え、限量詞で修飾されるものに $u, v, w, x, y, z, u_1, v_1,$

$w_1, x_1, y_1, z_1, u_2, v_2, w_2, x_2, y_2, z_2, \ldots$ 等の文字（変項,variable,という）で名前をつける。［もちろん、異なるものには異なる名前をつける。］そして、最も外側の論理的言葉が限量詞である様な言明については、それを

(2)　$\boldsymbol{Q}$ $\boldsymbol{x}$ (　$\mathcal{A}$　)

ここで、$\boldsymbol{Q}$ は当該限量詞の記号（すなわち、∀ または ∃）,

$\boldsymbol{x}$ は当該限量詞で修飾されるものの名前（すなわち、変項）,

$\mathcal{A}$ は、元の言明中に“当該限量詞で修飾されたもの”を参照する指示代名詞（もしくは、それと同等のもの）があれば、それらをすべて $\boldsymbol{x}$ で置き換え、さらに当該限量詞を除去することによって、元の言明から得られるもの

という形に統一的に書き換える。［すなわち、限量詞に関わる言明は“すべての x について ... である”とか“ある x について ... である”という言い方に統一することになる。］

以上をまとめて、述語論理の場合の論理構造明確化の手順は次の基本方針(I)〜(IV)に従ったものとなる。

(I) 結合の曖昧さを除去するために括弧を用いる。

(II) 記号で置き換えられていない結合詞と限量詞の中で、最も外側のものから順に記号で置き換えてゆく。

(III) 意味の同じ真理関数的結合詞を統一するために、否定,連言,選言,排他的選言,含意,同値の結合詞をそれぞれ 〜,∧,∨,⊕,→,↔ という記号で置き換える。

(IV) 最も外側の論理的言葉が限量詞である様な言明については(2)の形の式に置き換える。

例2.3　論理構造明確化の基本方針(I)〜(IV)に従えば、言明

(3) すべてのものは生命を持つか持たないかのいずれかである

は次の様に変形されてゆく。

(4)　∀ x (xは生命を持つか持たないかのいずれかである)

(5)　∀ x (xは生命を持つか、または xは生命を持たないかのいずれかである)

(6)　∀ x (xは生命を持つ ⊕ xは生命を持たない)

(7)　∀ x (xは生命を持つ ⊕ (〜 xは生命を持つ))

例2.4(限定された全称限量詞)　論理構造明確化の基本方針(I)〜(IV)に従えば、(1)の言明

すべての実数 x について、$x(x-1)=0$ ならば $x=0$ または $x=1$ である

は次の様に変形されてゆく。

(8)　∀ x (実数 x について、$x(x-1)=0$ ならば $x=0$ または $x=1$ である)

(9)　∀ x (実数 x が与えられた時、$x(x-1)=0$ ならば $x=0$ または $x=1$ である)

(10)　∀ x (xは実数である → $x(x-1)=0$ ならば $x=0$ または $x=1$ である)

(11) $\forall x$(xは実数である → ($x(x-1)=0$ → $x=0$ または $x=1$))

(12) $\forall x$(xは実数である → ($x(x-1)=0$ → ($x=0 \vee x=1$)))

この様に、

(13) すべての□□ xについて、 ...... である

という言明の論理構造を明確にすると

(14) $\forall x$(xは□□である → ...... である)

という風になり、“〜について”という言葉が含意記号 → で置き換えられる。

例2.5(限定された存在限量詞) 論理構造明確化の基本方針(I)〜(IV)に従えば、2.2節(12)の言明

ある県の県花はチューリップである

は次の様に変形されてゆく。

(15) $\exists x$(県 xの県花はチューリップである)

(16) $\exists x$(xは県であり、 xの県花はチューリップである)

(17) $\exists x$(xは県である ∧ xの県花はチューリップである)

より一般には、

(18) ある□□ xについて、 ...... である

という言明の論理構造を明確にすると

(19) $\exists x$(xは□□である ∧ ...... である)

という風になり、“〜について”という言葉が今度は連言記号 ∧ で置き換えられる。

例2.6(言葉の省略;W. O. クワイン[1]) 論理構造明確化の基本方針(I)〜(IV)に従って言明

(20) 日本人はいつも箸で食事をする

を機械的に変形してゆくと、次の様になる。

(21) $\forall x$(日本人xはいつも箸で食事をする)

(22) $\forall x$(xは日本人である → xはいつも箸で食事をする)

(23) $\forall x$(xは日本人である → $\forall y$(時刻 yには xは箸で食事をする))

(24) $\forall x$(xは日本人である → $\forall y$(yは時刻である → yには xは箸で食事をする)))

しかし、(23),(24)の言明は単に日本人が大食漢であることを言っているわけで、(20)〜(22)の本来の意味と異なることを言っている。従って、(20)〜(22)には言葉が省略されていると考えられ、これを補って変形すると次の様になる。

(25) $\forall x$(xは日本人である → xは食事の時にはいつも箸で食事をする)

(26) $\forall x$(xは日本人である → $\forall y$(xの食事の時刻 yには、 xは箸で食事をする))

(27) $\forall x$(xは日本人である →
　　$\forall y$(yは xの食事の時刻である → yには xは箸で食事をする))

(28) $\forall x$(xは日本人である →
　　$\forall y$(yは時刻である ∧ yには xは食事をする → yには xは箸で食事をする))

例2.7(複雑な言明；W. O. クワイン［１］)　論理構造明確化の基本方針(I)～(IV)に従えば、言明

(29) 新聞の勧誘員が新聞の予約購読の大嫌いな人にその申込をさせれば、その時以降は彼は新聞勧誘術を会得している

は次の様に変形されてゆく。

(30) $\forall x$(xは新聞の勧誘員 → xが新聞の予約購読の大嫌いな人にその申込をさせれば、
　　その時以降は xは新聞勧誘術を会得している)

(31) $\forall x$(xは新聞の勧誘員 →
　　$\forall u$(uは時刻 → xが uにおいて新聞の予約購読の大嫌いな人にその申込をさせれば、
　　u以降の時刻においては xは新聞勧誘術を会得している))

(32) $\forall x$(xは新聞の勧誘員 →
　　$\forall u$(uは時刻 → (xが uにおいて新聞の予約購読の大嫌いな人にその申込をさせる
　　→ u以降の時刻においては xは新聞勧誘術を会得している)))

(33) $\forall x$(xは新聞の勧誘員 →
　　$\forall u$(uは時刻 →
　　(∃ y(uにおいて新聞の予約購読の大嫌いな人 yについて、
　　xは uにおいて yに新聞の予約購読の申込をさせる)
　　→ u以降の時刻においては xは新聞勧誘術を会得している)))

(34) $\forall x$(xは新聞の勧誘員 →
　　$\forall u$(uは時刻 →
　　(∃ y(yは uにおいて新聞の予約購読の大嫌いな人
　　∧ xは uにおいて yに新聞の予約購読の申込をさせる)
　　→ u以降の時刻においては xは新聞勧誘術を会得している)))

(35) $\forall x$(xは新聞の勧誘員 →
　　$\forall u$(uは時刻 →
　　(∃ y(yは人 ∧ yは uにおいて新聞の予約購読が大嫌い
　　∧ xは uにおいて yに新聞の予約購読の申込をさせる)
　　→ u以降の時刻においては xは新聞勧誘術を会得している)))

(36) ∀x(xは新聞の勧誘員 →
　∀u(uは時刻 →
　　(∃y(yは人 ∧ yはuにおいて新聞の予約購読が大嫌い
　　　∧ xはuにおいてyに新聞の予約購読の申込をさせる)
　　→ ∀v(u以降の時刻vについて、
　　　　xはvにおいて新聞勧誘術を会得している))))

(37) ∀x(xは新聞の勧誘員 →
　∀u(uは時刻 →
　　(∃y(yは人 ∧ yはuにおいて新聞の予約購読が大嫌い
　　　∧ xはuにおいてyに新聞の予約購読の申込をさせる)
　　→ ∀v(vはu以降の時刻
　　　　→ xはvにおいて新聞勧誘術を会得している))))

(38) ∀x(xは新聞の勧誘員 →
　∀u(uは時刻 →
　　(∃y(yは人 ∧ yはuにおいて新聞の予約購読が大嫌い
　　　∧ xはuにおいてyに新聞の予約購読の申込をさせる)
　　→ ∀v(vは時刻 ∧ vはu以降
　　　　→ xはvにおいて新聞勧誘術を会得している))))

2．4　一階論理式 ― 述語表現と関数表現 ―

命題論理では、命題の論理構造を明確化するために真理関数的結合詞を全て記号で置き換え、さらに命題の特殊性を排除し真理関数的構造だけを抽出するために命題に含まれる各々の原子命題をp, q, r, s, ... 等の言明文字で置き換えた。ここでは、原子言明の構造を分析する必要はなく、言明文字で置き換える際に原子言明のどれとどれが同一の命題を表すかの判断ができれば十分であった。

これに対し述語論理では、命題の論理構造明確化の段階で、真理関数的結合詞の記号化の他に限量詞の記号化や変項の導入も行われ、原子言明や**原子開放言明**(atomic open statement;すなわち、変項を含み、変項全てに何らかの語を入れると原子言明になるもの)を ～, ∧, ∨, ⊕, →, ↔, ∀, ∃ という論理記号や括弧，変項でつなげた表現が得られる。

原子開放言明は主語や目的語などの部分に変項を含みこれらの変項は各々を修飾する限量詞と対になって論理構造の一部をなすから、命題から特殊性を排除しその論理構造だけを抽出する段階で

は、原子開放言明に置き換わる文字(列)には元々の原子開放命題に含まれる変項は全てそのまま含まれなければならない。従って、原子(開放)言明を系統的に文字列で置き換えるというのなら、原子(開放)言明の中の主語や目的語などの様に変項で置き換わり得るところを各々独立に何らかの文字(列)で置き換えなければならず、これによって得られた文字列は元の原子(開放)言明の主語や目的語に相当する部分を持つことになる。

また、原子(開放)言明内の主語や目的語に相当する部分の表記には、直接的に変項や対象物,人の名前を用いる他に、例えば

(1) Sebastian の父親,

(2) 数 x と数 2 の和, すなわち $x+2$

の様に、Sebastian, 2, x といった名前や変項を用いて間接的に別の対象物や人を指す, いわゆる "関数的" な表現を用いることもある。系統的な置き換えを行うというのなら、当然、原子(開放)言明内の(1),(2)などの部分はその関数構造を保存する文字列で置き換えることになる。

結局、原子(開放)言明を系統的に文字列で置き換えようとすれば、その文字列には元の原子(開放)言明の構造も、さらには言明内の対象物,人を表す関数構造も何らかの形で残ることになる。そこで、言明内の対象物,人を表す関数構造を

(3) □(○,○, ... ,○)

ここで、○の部分は対象物や人を表す文字,変項,またはより小さな関数構造を表す文字列,
□の部分はこの関数を表す文字

という形の文字列で、さらに原子(開放)言明を

(4) △(○,○, ... ,○)

ここで、○の部分は元の原子(開放)言明の主語や目的語などに置き換わるもので、対象物や人を表す文字,変項,または関数構造を表す文字列,
△の部分は○の箇所に置き換わらない残りの部分, すなわち述部に相当する部分を表す文字

という形の文字列で置き換えることで統一しよう。文字の使い方に関しては、対象物や人を表し○の位置に入る文字（**個体文字**, individual letter, という）として $a, b, c, d, a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2, \ldots$ などを用い、関数を表し□の位置に入る文字（**関数文字**, function letter, という）として $f, g, h, f_1, g_1, h_1, f_2, g_2, h_2, \ldots$ などを用い、また述部を表し△の位置に入る文字（**述語文字**, predicate letter, という）として $A, B, \ldots, Z, A_1, B_1, \ldots, Z_1, A_2, B_2, \ldots, Z_2, \ldots$ の様に英大文字またはそれに添字をつけたものを用いることで統一する。［関数文字, 述語文字は各々に固定された個数の引数(ひきすう)(argument; すなわち(3),(4)の○に相当する部分)を持つものとする。］

述語論理の場合には、命題から特殊性を排除して得られる論理構造を“閉じた一階述語論理式”，“閉じた一階論理式”，または“閉じた限量型式”といい、次の様に形式化する。

定義２．８　　項(term)のクラスを次の (i),(ii) により帰納的に定める。

(i) 変項と個体文字はどれも項である。

(ii) n引数の関数文字 $\boldsymbol{f}$と n個の項 $t_1, \ldots, t_n$ に対して、$\boldsymbol{f}(t_1, \ldots, t_n)$ も項である。

定義２．９　　**一階論理式**(first-order formula)のクラスを次の (i),(ii) により帰納的に定める。

(i) n引数の述語文字 $\boldsymbol{P}$と n個の項 $t_1, \ldots, t_n$ に対して、$\boldsymbol{P}(t_1, \ldots, t_n)$ は一階論理式である。これによって定まる一階論理式を特に**原子論理式**(atomic formula,**基本論理式**,**素論理式**)という。

(ii) 一階論理式 α, β と変項 $\boldsymbol{x}$に対して、$(\sim\alpha), (\alpha\wedge\beta), (\alpha\vee\beta), (\alpha\oplus\beta), (\alpha\rightarrow\beta), (\alpha\leftrightarrow\beta), \forall\boldsymbol{x}\alpha, \exists\boldsymbol{x}\alpha$ の８つの文字列パターンはどれも一階論理式である。

略記法．　通常の算術式や命題論理式の場合にならって、一階論理式の括弧の省略を行う。その際、結合の優先順位は $\sim, \forall, \exists, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ の順に高いものとする。

定義２．１０　　限量子 $\forall, \exists$ で修飾されていない変項を**自由変項**(free variable)といい、自由変項を持つかどうかに応じて一階論理式をそれぞれ **開いている**(open),**閉じている**(closed) という。

例２．１１　　一階論理式

(5) $\forall x P(x, y) \rightarrow P(x, y)$

の中には自由変項として x, y の２つが含まれるから、これは開いた一階論理式である。また、一階論理式

(6) $\forall x \exists y (\forall x P(x, y) \rightarrow P(x, y))$

は自由変項を含まないから閉じている。

例２．１２　　前節２．３(7)の言明

(7) $\forall x$(xは生命を持つ $\oplus$ ($\sim$ xは生命を持つ))

において、

“xは生命を持つ”という原子開放言明を原子論理式 $P(x)$ で

置き換えると

(8) $\forall x(P(x) \oplus (\sim P(x)))$

という構造が得られる。定義２．９～１０によれば、これは閉じた一階論理式である。

例2.13　前節２.３(17)の言明

(9) $\exists x$(xは県である ∧ xの県花はチューリップである)

において

“xの県花”　という関数構造を項 $f(x)$ で,

“xは県である”　という原子開放言明を原子論理式 $P(x)$ で,

“uはチューリップである”という原子開放言明を原子論理式 $T(u)$ で

置き換えると

“xの県花はチューリップである”という原子開放言明は $T(f(x))$ で

表されるから、結局(9)の論理構造として

(10) $\exists x(P(x) \wedge T(f(x)))$

という閉じた一階論理式が得られる。

例2.14　前節２.３(12)の言明

(11) $\forall x$(xは実数である → ($x(x-1)=0$ → ($x=0$ ∨ $x=1$)))

において

“０”という個体の名前を個体文字 a で,

“１”という個体の名前を個体文字 b で,

“uからvの減算結果”という関数構造を項 $f(u, v)$ で,

“uとvの積”　という関数構造を項 $g(u, v)$ で,

“xは実数である”という原子開放言明を原子論理式 $R(x)$ で,

“uとvは等しい”という原子開放言明を原子論理式 $E(u, v)$ で

置き換えると

“$x(x-1)$”という関数構造は項 $g(x, f(x, b))$ で

表され、さらに

“$x(x-1)=0$”という原子開放言明は論理式 $E(g(x, f(x, b)), a)$ で,

“$x=0$”　という原子開放言明は論理式 $E(x, a)$ で,

“$x=1$”　という原子開放言明は論理式 $E(x, b)$ で

表されるから、結局(11)の論理構造として

(12) $\forall x(R(x) \to (E(g(x, f(x, b)), a) \to (E(x, a) \vee E(x, b))))$

という閉じた一階論理式が得られる。

例2.15　２．１節(1)～(3)の推論を１つの言明で表すと、例えば

(13) ソクラテスは人間であり全ての人間はいつかは死ぬと言うのなら、ソクラテスもいつかは死ぬことになる

となる。これは、論理構造明確化の手順によって

(14) ソクラテスは人間である ∧ ∀x(xは人間である → xはいつかは死ぬ)
→ ソクラテスはいつかは死ぬ

と変形される。ここで

"ソクラテス"という個体の名前を個体文字 a で,
"xは人間である" という原子開放言明を原子論理式 $P(x)$ で,
"xはいつかは死ぬ"という原子開放言明を原子論理式 $M(x)$ で

置き換えると

"ソクラテスは人間である" という原子言明は $P(a)$ で,
"ソクラテスはいつかは死ぬ"という原子言明は $M(a)$ で

表されるから、(14)の論理構造として

(15) $P(a) \land \forall x(P(x) \to M(x)) \to M(a)$

という閉じた一階論理式が得られる。

例2.16　2.1節(10)〜(12)の推論を1つの言明で表すと、例えば

(16) W.A.Mozart がザルツブルグ生まれの作曲家であることから、ザルツブルグ生まれの作曲家の存在は明かである

となる。これは、論理構造明確化の手順によって

(17) W.A.Mozart はザルツブルグで生まれた ∧ W.A.Mozart は作曲家である
→ ∃x(xはザルツブルグで生まれた ∧ xは作曲家である)

と変形される。ここで

"W.A.Mozart"という個体の名前を個体文字 a で,
"xはザルツブルグで生まれた"という原子開放言明を原子論理式 $Z(x)$ で,
"xは作曲家である" という原子開放言明を原子論理式 $C(x)$ で

置き換えると

"W.A.Mozart はザルツブルグで生まれた"という原子言明は $Z(a)$ で,
"W.A.Mozart は作曲家である" という原子言明は $C(a)$ で

表されるから、(17)の論理構造として

(18) $Z(a) \land C(a) \to \exists x(Z(x) \land C(x))$

という閉じた一階論理式が得られる。

2.5 一階論理式の解釈

命題論理式の中で意味の決まっていない文字は言明文字しかないから、命題論理式に具体的な意味を与えるためにはその中の各々の言明文字に原子言明を割り当てればよい。［もちろん、(割当てによって)得られた命題の真理値は、どの言明文字に真の原子言明を割り当て，どの言明文字に偽の原子言明を割り当てるかだけに依存して決まる。］

これに対して、閉じた一階論理式の中には意味の決まっていない文字として個体文字，関数文字，述語文字の３種類があるから、閉じた一階論理式に具体的な意味を与えるためにはその中の個体文字に具体的な物または人を，関数文字に関数構造を持つ言い回しを，そして述語文字に原子開放言明を割り当てなければならない。［開いた一階論理式の場合には個体文字，関数文字，述語文字への割当てを行っても自由変項の部分の意味が決まらない。従って、自由変項にも個体文字と同様に具体的な物または人を割り当てなければならない。］

どんな割当てによって一階論理式から真の命題が得られ，どんな割当てによって偽の命題が得られるかを問題にする場合には、結局は命題論理の場合と同様に具体的な割当てを考える必要はなく次の定義の様に

関数文字に割り当てられる言い回しの“関数としての振る舞い”，

述語文字に割り当てられる原子開放言明の“関数としての振る舞い”

についてだけ注目すれば十分である。

定義2.17 (i) １個以上の個体からなる集合Dを設定し，

(ii) 各個体文字，各自由変項にDの要素を割り当て，

(iii) 各(n引数)関数文字$\boldsymbol{f}$とDの各要素 $e_1, \ldots, e_n$ に対し $\boldsymbol{f}(e_1, \ldots, e_n)$ にDの要素を割り当て，

(iv) 各(n引数)述語文字$\boldsymbol{P}$とDの各要素 $e_1, \ldots, e_n$ に対し $\boldsymbol{P}(e_1, \ldots, e_n)$ に真または偽を割り当てる

ことを**解釈**(interpretation)という。 そして、命題論理の場合と同様に一階論理式αの真理値を$I(\alpha)$で表し、$I(\alpha)=$真 の時 $\models_I \alpha$，$I(\alpha)=$偽 の時 $\not\models_I \alpha$ と書く。

例2.18 個体の集合を $D=\{e_1, e_2\}$ とし、

$f(e_1), f(e_2)$ にそれぞれ e_2, e_1 を割当て，

$P(e_1), P(e_2)$ にそれぞれ 真，偽 を割当て，

$T(e_1), T(e_2)$ にそれぞれ 真，偽 を割当てる

解釈Iに対しては、

$$
\begin{aligned}
I(P(e_1)\wedge T(f(e_1))) &= I(P(e_1)\wedge T(e_2)) \\
&= I(\text{真}\wedge\text{偽}) \\
&= \text{偽}, \\
I(P(e_2)\wedge T(f(e_2))) &= I(P(e_2)\wedge T(e_1)) \\
&= I(\text{偽}\wedge\text{真}) \\
&= \text{偽}
\end{aligned}
$$

であるから、

$$I(\exists x(P(x)\wedge T(f(x))))=\text{偽},$$

すなわち

$$\not\models_I \exists x(P(x)\wedge T(f(x)))$$

である。［ここで、$P(e_1)\wedge T(f(e_1))$ は一階論理式でないから、上の定義２．１７に従えば厳密には $I(P(e_1)\wedge T(f(e_1)))$ という表記は誤りである。しかし、“部分的に解釈済み”の式 α に対しても $I(\alpha)$ の表記をこの様に自然に拡張して使うことにする。］

また、

$T(e_1)$, $T(e_2)$ にそれぞれ 偽,真 を割当てる

こと以外は全く I と同じ解釈 J に対しては、

$$
\begin{aligned}
J(P(e_1)\wedge T(f(e_1))) &= J(P(e_1)\wedge T(e_2)) \\
&= J(\text{真}\wedge\text{真}) \\
&= \text{真}, \\
J(P(e_2)\wedge T(f(e_2))) &= J(P(e_2)\wedge T(e_1)) \\
&= J(\text{偽}\wedge\text{偽}) \\
&= \text{偽}
\end{aligned}
$$

であるから、

$$J(\exists x(P(x)\wedge T(f(x))))=\text{真},$$

すなわち

$$\models_J \exists x(P(x)\wedge T(f(x)))$$

である。

恒真性,充足可能性の考え方は命題論理の場合と全く同じである。結合子 $\sim, \wedge, \vee, \oplus, \rightarrow, \leftrightarrow$ と限量子 $\forall, \exists$ の論理的意味によって常に真と認められる命題のパターンが、恒真な(閉じた)一階論理式に他ならない。形式的な定義のためには定義１．１０を少し手直しするだけでよい。

定義２．１９　　一階論理式 α に関して、

(i) どんな解釈 I を与えても $\models_I \alpha$ となる時、α は恒真(valid)であるといい $\models \alpha$ と書く。

(ii) どんな解釈 I を与えても $\not\models_I \alpha$ となる時、α は充足不能(unsatisfiable)または矛盾(incon-

sistent)であるという。

(iii) ある解釈 I に対して $\models_I \alpha$ となる時、α は**充足可能**(satisfiable)または**無矛盾**(consistent)であるという。

命題論理の場合と同様に、この定義から直接、次の命題が導かれる。

命題２.２０　任意の一階論理式 α に関して、

(i) α が恒真 $\Leftrightarrow$ $(\sim\alpha)$ が矛盾,

(ii) α が矛盾 $\Leftrightarrow$ $(\sim\alpha)$ が恒真,

(iii) α が無矛盾 $\Leftrightarrow$ α が矛盾でない

$\Leftrightarrow$ $(\sim\alpha)$ が恒真でない

例２.２１　前節２.４(10)の閉じた一階論理式

(1) $\exists x(P(x) \wedge T(f(x)))$

は例２.１８より恒真でも矛盾でもなく、従って充足可能である。

例２.２２　前節２.４(8)の閉じた一階論理式

(2) $\forall x(P(x) \oplus \sim P(x))$

は恒真である。実際、解釈 I をどの様に与えたとしても、そこで考えている任意の個体 e に関して

$$I(P(e) \oplus \sim P(e)) = 真$$

となるから、

$$I(\forall x(P(x) \oplus \sim P(x))) = 真$$

が導かれる。

例２.２３　前節２.４(18)の閉じた一階論理式

(3) $Z(a) \wedge C(a) \rightarrow \exists x(Z(x) \wedge C(x))$

は恒真である。なぜなら、仮にある解釈 I の下で

(4) $\not\models_I Z(a) \wedge C(a) \rightarrow \exists x(Z(x) \wedge C(x))$

とすると、含意結合子→の真理関数的性質から

(5) $\models_I Z(a) \wedge C(a)$,

(6) $\not\models_I \exists x(Z(x) \wedge C(x))$

が得られる。しかし、(5)式からは

(7) $\models_I \exists x(Z(x) \wedge C(x))$

という、(6)と矛盾した式が導かれる。

例２．２４　前節２．４(15)の閉じた一階論理式

(8) $P(a) \land \forall x(P(x) \to M(x)) \to M(a)$

は恒真である。なぜなら、仮にある解釈 I の下で

(9) $\not\models_I P(a) \land \forall x(P(x) \to M(x)) \to M(a)$

とすると、結合子 →, ∧ の真理関数的性質から

(10) $\models_I P(a)$,

(11) $\models_I \forall x(P(x) \to M(x))$,

(12) $\not\models_I M(a)$

が得られる。しかし、(5), (7)式からは

(13) $\not\models_I P(a) \to M(a)$

従って

(14) $\not\models_I \forall x(P(x) \to M(x))$

が導かれる。この(14)式は(11)と矛盾する。

一階論理式が恒真であるかどうか,矛盾であるかどうかを調べるのは、命題論理の場合と異なり厄介である。命題論理の場合に倣って考えられる解釈を全て調べ尽くそうとしても、個体の集合 D の大きさを決めるだけで無限の可能性が生じてしまう。しかも、D を１つ固定したとしてもそれが無限集合なら、与えられた一階論理式に対して本質的に無限個の解釈が可能になる。D を１つの有限集合に固定すれば、与えられた一階論理式に対して本質的に有限個の解釈しかなくなるが、ほとんどの場合それら全てを調べ尽くすのは困難である。［例えば、n 引数述語文字 P については、

$P(e_1, \ldots, e_n)$　　但し、各 $e_I \in D$

という形の式にそれぞれ独立に真理値を割り当てることになるから、全部で

$2^{|D|^n}$ 通り　　但し、$|D|$ は D の中の要素の個数

の割当て方法が可能になる。］結局、一階論理式の恒真性判定の際の困難さを克服する手段はなく、“全ての一階論理式に対して系統的に恒真性を判定するための方法がない”ことが理論的にも証明されている。系統的な方法として我々が与えることのできる最良のものは“一階論理式が恒真の時には恒真であると断定し、そうでない時には恒真でないと断定するか永久に停止しない”というものであり、例えば背理法に基づく方法が古くから知られている。

２．６　論理的帰結

論理的帰結の考え方も命題論理の場合と全く同じである。 結合子 ～, ∧, ∨, ⊕, →, ↔ と限量子 ∀, ∃ の論理的意味によって正しいと認められる推論パターンを形式化したものが論理的帰結の考え方に他ならない。形式的な定義のためには定義１．１５を少し手直しするだけでよい。

定義２．２５　一階論理式の集合Γと一階論理式αに関して、Γ内の全ての一階論理式を真にする解釈Iの下で常に$I(\alpha)$が真となるならば、αはΓの**論理的帰結**(logical consequence)であるといい $\Gamma \models \alpha$ と書く。 特にΓが有限で $\Gamma = \{\beta_1, \beta_2, \ldots, \beta_n\}$ の時、$\Gamma \models \alpha$ の代わりに $\beta_1, \beta_2, \ldots, \beta_n \models \alpha$ と略記する。

$\models$に関する演繹定理も、やはり命題論理の場合の定理１．１６と同様に成り立つ。これによって、Γが有限の場合に $\Gamma \models \alpha$ であるかどうかを判定する問題は、恒真性判定の問題に帰着できる。

定理２．２６（$\models$に関する演繹定理）　任意の一階論理式 α, β と一階論理式の集合Γに関して、

$$\Gamma \cup \{\alpha\} \models \beta \quad \Leftrightarrow \quad \Gamma \models \alpha \to \beta$$

例２．２７　定理２．２６により

$$Z(a), C(a) \models \exists x(Z(x) \wedge C(x))$$
$$\Leftrightarrow Z(a) \wedge C(a) \models \exists x(Z(x) \wedge C(x))$$
$$\Leftrightarrow \models Z(a) \wedge C(a) \to \exists x(Z(x) \wedge C(x))$$

であり、また例２．２３により $\models Z(a) \wedge C(a) \to \exists x(Z(x) \wedge C(x))$ であることが分かるから

$$Z(a), C(a) \models \exists x(Z(x) \wedge C(x)),$$

すなわち $\exists x(Z(x) \wedge C(x))$ は $Z(a)$ と $C(a)$ の論理的帰結であると結論できる。これによって、２．１節(10)～(12)の推論が論理的に正当化されたことになる。

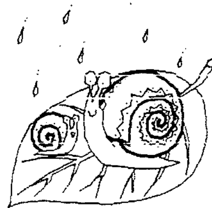
例２．２８　定理２．２６により

$$P(a), \forall x(P(x) \to M(x)) \models M(a)$$
$$\Leftrightarrow P(a) \wedge \forall x(P(x) \to M(x)) \models M(a)$$
$$\Leftrightarrow \models P(a) \wedge \forall x(P(x) \to M(x)) \to M(a)$$

であり、また例２．２４により $\models P(a) \wedge \forall x(P(x) \to M(x)) \to M(a)$ であることが分かるから

$$P(a), \forall x(P(x) \to M(x)) \models M(a),$$

すなわち $M(a)$ は $P(a)$ と $\forall x(P(x) \to M(x))$ の論理的帰結であると結論できる。これによって、２．１節(1)～(3)の推論が論理的に正当化されたことになる。



2.7 一階論理式の代数

論理的同値の考え方も命題論理の場合と全く同じであり、その形式的定義のためには定義1.19を少し手直しするだけでよい。

定義2.29 全ての解釈 I について $I(\alpha)=I(\beta)$ となる時、2つの一階論理式 α,β は**論理的に同値**(logically equivalent)であるといい $\alpha\equiv\beta$ と書く。

命題論理式の変形に関する3つの基本定理のうち、定理1.20と定理1.22(置き換え法則)は一階論理式の場合にもほとんどそのままの形で与えることができる。

定理2.30 任意の一階論理式 α,β,γ に対して、
(i) $\alpha\equiv\alpha$,
(ii) $\alpha\equiv\beta$ ならば $\beta\equiv\alpha$,
(iii) $\alpha\equiv\beta,\beta\equiv\gamma$ ならば $\alpha\equiv\gamma$

定理2.31(置き換え法則) 任意の一階論理式 α に対して、“一部分(の一階論理式)をそれと論理的に同値な式で置き換える”という操作によって α から得られる式は、元の式 α と論理的に同値である。

また、定理1.21(代入法則)を一階論理式用に拡張するには、代入に関する考え方を整理しておかなければならない。なぜなら、述語文字へ一階論理式を代入するためには、述語の引数と一階論理式内の変項を結合しなければならないからである。

表記法2.32 一階論理式 γ に関して、その中の自由変項が $\boldsymbol{x}_1,\ \ldots\ ,\boldsymbol{x}_n$ であることを明示する時には γ の代わりに $\gamma(\boldsymbol{x}_1,\ \ldots\ ,\boldsymbol{x}_n)$ と書く。また、$\gamma(\boldsymbol{x}_1,\ \ldots\ ,\boldsymbol{x}_n)$ 内の自由変項 $\boldsymbol{x}_1,\ \ldots\ ,\boldsymbol{x}_n$ をそれぞれ項 $t_1,\ \ldots\ ,t_n$ で置き換えて得られる一階論理式を $\gamma(t_1,\ \ldots\ ,t_n)$ で表す。

定義2.33 一階論理式 $\alpha,\gamma(\boldsymbol{x}_1,\ \ldots\ ,\boldsymbol{x}_n)$ と m 引数述語文字 $\boldsymbol{P}$ が与えられているとする。α 内の $\boldsymbol{P}(t_1,\ \ldots\ ,t_m)$ という形の原子論理式をそれぞれ

$\gamma(t_1,\ \ldots\ ,t_m,\boldsymbol{x}_{m+1},\ \ldots\ ,\boldsymbol{x}_n)$ ($m<n$ の時)

または

$\gamma(t_1,\ \ldots\ ,t_n)$ ($m\geqq n$ の時)

で置き換えて得られる一階論理式は、2つの条件

(i) どの $\mathbb{P}(t_1, ..., t_m)$ についても、その中の変項が置き換えによって新たにγ内の限量子で修飾されることはない,

(ii) どの $\mathbb{P}(t_1, ..., t_m)$,但し$m<n$, についても、$\boldsymbol{y}_{m+1}, ..., \boldsymbol{y}_n$ の中の変項が置き換えによって新たにα内の限量子で修飾されることはない

を満たす場合に、“α内の述語文字$\mathbb{P}$に一階論理式 $\gamma(\boldsymbol{y}_1, ..., \boldsymbol{y}_n)$ を代入(substitute)して得られる一階論理式”であるといい、$\alpha[\gamma(\boldsymbol{y}_1, ..., \boldsymbol{y}_n)/\mathbb{P}]$ で表す。

例２.３４　　一階論理式

(1) $\forall y P(x, y) \to P(y, x)$,

(2) $\exists z Q(u, v, w, z) \to R(u, v, w)$

をそれぞれ $\alpha, \gamma(u, v, w)$ で表す。すると、α内の $P(t_1, t_2)$ という形の原子論理式をそれぞれ $\gamma(t_1, t_2, w)$ で置き換えて得られる一階論理式

(3) $\forall y(\exists z Q(x, y, w, z) \to R(x, y, w)) \to (\exists z Q(y, x, w, z) \to R(y, x, w))$

は定義２.３３の条件(i),(ii)を満たす。それゆえ、この(3)式はα内の述語文字Pに $\gamma(u, v, w)$ を代入して得られる一階論理式であり、$\alpha[\gamma(u, v, w)/P]$ で表すことができる。［定義２.３３の条件(i),(ii)を満たさないものとしては、例えば、α内の $P(t_1, t_2)$ という形の原子論理式をそれぞれ

(4) $\forall x Q(t_2, x, t_1)$

で置き換える場合を考えればよい。実際、この置き換えによって

(5) $\forall y \underline{\forall} x Q(y, x, \underline{x}) \to \underline{\forall} x Q(\underline{x}, x, y)$

という一階論理式が得られる。この(5)式中の下線部の変項xは元々α内でPの引数であったにもかかわらず“置き換え”によって新たに(4)式中の限量子で修飾されるので、条件(i)は満たされない。

また、α内の $P(t_1, t_2)$ という形の原子論理式をそれぞれ

(6) $Q(t_2, y, t_1)$

で置き換えれば

(7) $\underline{\forall} y Q(y, \underline{y}, x) \to Q(x, y, y)$

という一階論理式が得られる。この(7)式中の下線部の変項yは元々(6)式の自由変項であったにもかかわらず“置き換え”によって新たにα内の限量子で修飾されるので、この場合は条件(ii)は満たされない。］

以上の準備を経て、定理１.２１を次の様に一階論理式用に拡張できる。

定理２.３５(代入法則)　　２つの一階論理式 α, β が論理的に同値なとき、αとβ内の述語文字$\mathbb{P}$に一階論理式 $\gamma(\boldsymbol{y}_1, ..., \boldsymbol{y}_n)$ を代入して得られる一階論理式 $\alpha[\gamma(\boldsymbol{y}_1, ..., \boldsymbol{y}_n)/\mathbb{P}]$ と $\beta[\gamma(\boldsymbol{y}_1, ..., \boldsymbol{y}_n)/\mathbb{P}]$ も論理的に同値である。

一階論理式の変形に有用な基本等式としては、命題論理式の場合の定理１．２４を拡張したものがすぐに思い浮かぶ。次の通り。

定理２．３６　任意の一階論理式 α, β, γ に関して次の(i)〜(xv)が成り立つ。但し、ここでは、ある０引数述語文字 $\boldsymbol{P}$ を用いて $\boldsymbol{P}() \vee \sim \boldsymbol{P}()$ を true で，$\boldsymbol{P}() \wedge \sim \boldsymbol{P}()$ を false で表す。

(i) $\alpha \vee \alpha \equiv \alpha$,　$\alpha \wedge \alpha \equiv \alpha$　(べき等律)

(ii) $(\alpha \vee \beta) \vee \gamma \equiv \alpha \vee (\beta \vee \gamma)$,　$(\alpha \wedge \beta) \wedge \gamma \equiv \alpha \wedge (\beta \wedge \gamma)$　(結合律)

(iii) $\alpha \vee \beta \equiv \beta \vee \alpha$,　$\alpha \wedge \beta \equiv \beta \wedge \alpha$　(交換律)

(iv) $\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$,　$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$　(分配律)

(v) $\beta \vee \mathrm{true} \equiv \mathrm{true}$,　$\beta \wedge \mathrm{true} \equiv \beta$　(単位元律)

(vi) $\beta \vee \mathrm{false} \equiv \beta$,　$\beta \wedge \mathrm{false} \equiv \mathrm{false}$　(単位元律)

(vii) $\beta \vee \sim \beta \equiv \mathrm{true}$,　$\beta \wedge \sim \beta \equiv \mathrm{false}$　(補元律)

(viii) $\sim \mathrm{true} \equiv \mathrm{false}$,　$\sim \mathrm{false} \equiv \mathrm{true}$　(補元律)

(ix) $\sim (\sim \alpha) \equiv \alpha$,　(対合律)

(x) $\sim (\alpha \vee \beta) \equiv \sim \alpha \wedge \sim \beta$,　$\sim (\alpha \wedge \beta) \equiv \sim \alpha \vee \sim \beta$　(ドモルガンの法則)

(xi) $\alpha \rightarrow \beta \equiv \sim \alpha \vee \beta$　(→の除去，導入)

(xii) $\alpha \wedge \beta \equiv \sim (\alpha \rightarrow \sim \beta)$　(∧の除去，導入)

(xiii) $\alpha \vee \beta \equiv \sim \alpha \rightarrow \beta$　(∨の除去，導入)

(xiv) $\alpha \leftrightarrow \beta \equiv (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$　(↔の除去，導入)

(xv) $\alpha \oplus \beta \equiv \sim (\alpha \leftrightarrow \beta)$　(⊕の除去，導入)

また、限量子に関する基本的な等式をまとめて次の定理が得られる。

定理２．３７　任意の一階論理式 α, β と任意の変項 $\boldsymbol{x}$ に関して、次の(i)〜(iv)が成り立つ。

(i) $\boldsymbol{x}$ が β の自由変項でなければ

$\forall \boldsymbol{x} \alpha \vee \beta \equiv \forall \boldsymbol{x} (\alpha \vee \beta)$,　$\exists \boldsymbol{x} \alpha \vee \beta \equiv \exists \boldsymbol{x} (\alpha \vee \beta)$,

$\forall \boldsymbol{x} \alpha \wedge \beta \equiv \forall \boldsymbol{x} (\alpha \wedge \beta)$,　$\exists \boldsymbol{x} \alpha \wedge \beta \equiv \exists \boldsymbol{x} (\alpha \wedge \beta)$

である。

(ii) $\sim \forall \boldsymbol{x} \alpha \equiv \exists \boldsymbol{x} \sim \alpha$,　$\sim \exists \boldsymbol{x} \alpha \equiv \forall \boldsymbol{x} \sim \alpha$

(iii) $\forall \boldsymbol{x} \alpha \wedge \forall \boldsymbol{x} \beta \equiv \forall \boldsymbol{x} (\alpha \wedge \beta)$,　$\exists \boldsymbol{x} \alpha \vee \exists \boldsymbol{x} \beta \equiv \exists \boldsymbol{x} (\alpha \vee \beta)$

(iv) α が β の別形(variant)，すなわち α が

β 内の $\boldsymbol{Q} \boldsymbol{z} \gamma(\ldots, \boldsymbol{z}, \ldots)$ なる形の式を $\boldsymbol{Q} \boldsymbol{y} \gamma(\ldots, \boldsymbol{y}, \ldots)$ で置き換える

但し、$\boldsymbol{Q}$ は限量子，

$\boldsymbol{z}$ と $\boldsymbol{y}$ は変項を表し、

$\boldsymbol{y}$ は γ 内の自由変項でないとする

という操作によってβから得られる式であるなら、$\alpha \equiv \beta$ である。

２．８　標準形

一階論理式の標準形としては、次の定理に示されたものが一般的である。

定理２．３８(冠頭標準形)　一階論理式が任意に与えられたとき、論理的同値性を保ちながらそれを

(1) $Q_1 \boldsymbol{x}_1 Q_2 \boldsymbol{x}_2 \ \ldots \ Q_n \boldsymbol{x}_n \ \alpha$

ここで、各Q_iは限量子,

各$\boldsymbol{x}_i$は変項を表し、

αは限量子を含まない一階論理式である

という形(**冠頭標準形**,prenex normal form,という；この内$Q_1 \boldsymbol{x}_1 Q_2 \boldsymbol{x}_2 \ \ldots \ Q_n \boldsymbol{x}_n$の部分を**前置部**,prefix,といい、$\alpha$の部分を**母式**,matrix,という)に変形できる。　具体的には、前節の定理２．３０～３１と定理２．３６～３７を用いて次の(i)～(vi)の順に変形を行えばよい。［途中で結合律,交換律,べき等律,補元律,単位元律を用いればより簡単な冠頭標準形が得られる。］

(i) 定理２．３６(xv)と定理２．３０～３１を用いて、結合子として ～,∧,∨,→,↔ だけを含むものに変形する。（⊕の除去）

(ii) 定理２．３６(xiv)と定理２．３０～３１を用いて、結合子として ～,∧,∨,→ だけを含むものに変形する。（↔の除去）

(iii) 定理２．３６(xi),(xiii)と定理２．３０～３１を用いて、結合子として ～,∧,∨ だけを含むものに変形する。（→の除去）

(iv) 対合律,ドモルガンの法則,定理２．３７(ii)と定理２．３０～３１を用いて、否定結合子～を全て限量子の内側にもってくる。

(v) 定理２．３７(iv)と定理２．３０～３１を用いて変項名の付け替えを行い、限量子に続く変項が“他の限量子に続く変項”とも自由変項とも異なる様にする。

(vi) 定理２．３７(i),交換律と定理２．３０～３１を用いて、冠頭標準形に変形する。

もちろん、冠頭標準形の一階論理式が与えられたときには、分配律を用いてその母式を選言標準形や連言標準形に変形できる。その結果得られる式の形をそれぞれ**冠頭選言標準形**(prenex-disjunctive normal form), **冠頭連言標準形**(prenex-conjunctive normal form)という。

例2.39 一階論理式 $\sim\exists x P(x,y)\oplus(\exists x P(x,y)\vee\forall x Q(x))$ は定理2.38で述べられた手順により次の様に変形できる。

$\sim\exists x P(x,y)\oplus(\exists x P(x,y)\vee\forall x Q(x))$

$\equiv (\sim\exists x P(x,y)\wedge\sim\forall x Q(x)) \vee ((\exists x P(x,y)\vee\forall x Q(x))\wedge\exists x P(x,y))$

(手順(i)〜(iii),手順(iv)の一部；例1.29において選言標準形への変形手順(i)〜(iv)を適用して $\sim p\oplus(p\vee q)$ から $(\sim p\wedge\sim q)\vee((p\vee q)\wedge p)$ を導いたのと同様)

$\equiv (\forall x\sim P(x,y)\wedge\exists x\sim Q(x)) \vee ((\exists x P(x,y)\vee\forall x Q(x))\wedge\exists x P(x,y))$

(手順(iv)の残り)

$\equiv (\forall x\sim P(x,y)\wedge\exists z\sim Q(z)) \vee ((\exists u P(u,y)\vee\forall v Q(v))\wedge\exists w P(w,y))$

(手順(v))

$\equiv \forall x(\sim P(x,y)\wedge\exists z\sim Q(z)) \vee ((\exists u P(u,y)\vee\forall v Q(v))\wedge\exists w P(w,y))$

(手順(vi))

$\equiv \forall x((\sim P(x,y)\wedge\exists z\sim Q(z)) \vee ((\exists u P(u,y)\vee\forall v Q(v))\wedge\exists w P(w,y)))$

$\equiv \forall x(\exists z(\sim P(x,y)\wedge\sim Q(z)) \vee ((\exists u P(u,y)\vee\forall v Q(v))\wedge\exists w P(w,y)))$

$\equiv \forall x\exists z((\sim P(x,y)\wedge\sim Q(z)) \vee ((\exists u P(u,y)\vee\forall v Q(v))\wedge\exists w P(w,y)))$

$\vdots$

$\equiv \forall x\exists z\exists u\forall v\exists w((\sim P(x,y)\wedge\sim Q(z)) \vee ((P(u,y)\vee Q(v))\wedge P(w,y)))$

また、定理2.38の手順にこだわらなければ次の様に変形できる。[定理2.37(iii)をうまく用いれば手順(v)〜(vi)をそのまま用いるより限量子の個数が少なくてすむことに注目せよ。]

$\sim\exists x P(x,y)\oplus(\exists x P(x,y)\vee\forall x Q(x))$

$\equiv \sim\forall x Q(x) \vee \exists x P(x,y)$

(例1.26や例1.29において $\sim p\oplus(p\vee q)\equiv\sim q\vee p$ と変形できたのと同様)

$\equiv \exists x\sim Q(x) \vee \exists x P(x,y)$ 定理2.37(ii),置き換え法則

$\equiv \exists x(\sim Q(x) \vee P(x,y))$ 定理2.37(iii),置き換え法則

2．9　命題の限量的正しさと同値性

命題論理では命題の真理関数的構造（すなわち、真理関数的結合子のなす構造）に注目した。そして、この構造を用いて命題や推論の真理関数的（すなわち、結合子の真理関数的性質による）正しさについての形式化を行い、“真理関数的に真”，“真理関数的帰結”，“真理関数的同値”という考え方を導入した。

これに対し述語論理では、真理関数的結合子だけでなく限量子にも注目するから、命題の持つ論理構造（すなわち一階論理式で表される構造）を**限量的構造**(quantificational structure)と呼ぶ。しかし、命題や推論の限量的（すなわち結合子の真理関数的性質と限量子の意味による）正しさについて形式化するためには、命題論理の場合と全く同じ考えに基づいて、定義1．32を少し手直しするだけでよい。

定義2．40　(i) 命題$\mathcal{A}$の限量的構造を抽出して一階論理式αが得られるとする。この時、もしαが恒真なら元の命題$\mathcal{A}$は**限量的に真**(quantificationally true)であるといい、またαが矛盾なら$\mathcal{A}$は**限量的に偽**(quantificationally false)であるという。

(ii)　命題 $\mathcal{A}, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ の限量的構造を抽出して、それぞれ一階論理式 $\alpha, \beta_1, \beta_2, \ldots, \beta_n$ が得られるとする。[もちろん、$\mathcal{A}, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ 内で同一の個体は同一の個体文字で，同一の関数構造は同一の関数文字で，同一の述部は同一の述語文字で置き換えられるものとする。]　この時、もしαが $\beta_1, \beta_2, \ldots, \beta_n$ の論理的帰結，すなわち $\beta_1, \beta_2, \ldots, \beta_n \models \alpha$ であるなら、元の命題について$\mathcal{A}$は $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ の**限量的帰結**(quantificational consequence)であるという。

(iii) 命題 $\mathcal{A}, \mathcal{B}$ の限量的構造を抽出して、それぞれ一階論理式 α, β が得られるとする。[もちろん、$\mathcal{A}, \mathcal{B}$ 内で同一の個体は同一の個体文字で，同一の関数構造は同一の関数文字で，同一の述部は同一の述語文字で置き換えられるものとする。]　この時、もしαとβが論理的に同値，すなわち $\alpha \equiv \beta$ であるなら、元の2つの命題$\mathcal{A}$と$\mathcal{B}$は**限量的に同値**(quantificationally equivalent)であるという。

例2．41　　2．3節(3)の命題

(1) すべてのものは生命を持つか持たないかのいずれかである

は限量的に真である。なぜなら、2．4節の例1．12により(1)の限量的構造として

(2) $\forall x(P(x) \oplus (\sim P(x)))$

という一階論理式が得られ、2．5節の例2．22によりこの(2)式が恒真であることが分かるからである。

例２．４２　　２．２節(12)の命題

(3) ある県の県花はチューリップである

は、個々の言葉の意味を考えて初めて正しいと認められるものであり、限量的に真でも偽でもない。なぜなら、２．４節の例２．１３により(3)の限量的構造として

(4) $\exists x(P(x) \wedge T(f(x)))$

という一階論理式が得られ、この(4)式は２．５節の例２．２１で見たように恒真でも矛盾でもないからである。

例２．４３　　２．１節(3)の命題

(5) ソクラテスはいつかは死ぬ

は２．１節(1)～(2)の命題

(6) すべての人間はいつかは死ぬ,

(7) ソクラテスは人間である

の限量的帰結である。なぜなら、２．４節の例２．１５と同様にすれば、(5)～(7)の限量的構造としてそれぞれ

(8) $M(a)$,

(9) $\forall x(P(x) \rightarrow M(x))$,

(10) $P(a)$

という一階論理式が得られ、これらの(8)～(10)に関しては２．６節の例２．２８に示すように(8)式が(9)～(10)の論理的帰結になっているからである。

例２．４４　　２．１節(12)の命題

(11) ある作曲家はザルツブルグで生まれた

は２．１節(10)～(11)の命題

(12) W.A.Mozart はザルツブルグで生まれた,

(13) W.A.Mozart は作曲家である

の限量的帰結である。なぜなら、２．４節の例２．１６と同様にすれば、(11)～(13)の限量的構造としてそれぞれ

(14) $\exists x(Z(x) \wedge C(x))$,

(15) $Z(a)$,

(16) $C(a)$

という一階論理式が得られ、これらの(14)～(16)に関しては２．６節の例２．２７に示すように(14)式が(15)～(16)の論理的帰結になっているからである。

演習問題２

2.1　日本語の言明

① 全てのペンギンは翼を持つが空を飛べない，

② 月曜日の次の日は火曜日である，

③ 父親の母親は祖母である，

④ 全ての自然数に対して、それより大きな自然数が存在する

について、

(1) それぞれの言明は、２．３節で示した論理構造明確化の手順によって、どの様に変形されてゆくか？

(2) それぞれの言明の論理構造は、どんな一階論理式で表されるか？

2.2　命題２．２０を証明せよ。

2.3　次の各々の一階論理式について、恒真かどうか，矛盾かどうか，充足可能かどうかを調べよ。

(1) $\forall x P(x) \leftrightarrow \sim\exists x \sim P(x)$

(2) $\forall x P(x) \wedge \exists x \sim P(x)$

(3) $\forall x (P(x) \wedge Q(x)) \to (\forall x P(x) \wedge \forall x Q(x))$

(4) $\forall x (P(x) \vee Q(x)) \to (\forall x P(x) \vee \forall x Q(x))$

(5) $(\forall x P(x) \vee \forall x Q(x)) \to \forall x (P(x) \vee Q(x))$

(6) $\forall x \exists y T(x, y) \to \exists y \forall x T(x, y)$

(7) $\exists y \forall x T(x, y) \to \forall x \exists y T(x, y)$

2.4　定理２．２６を証明せよ。

2.5　定理２．３０～３１，定理２．３５を証明せよ。

2.6　定理２．３６～３７を証明せよ。

2.7　上の問題２．３で与えた一階論理式をそれぞれ冠頭標準形，冠頭選言標準形，冠頭連言標準形に変形せよ。

２．８　３つの命題

① ２０歳以上の人は誰でも選挙権を持っている，

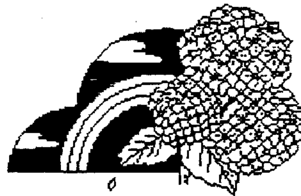
② Ａ氏の息子で選挙権を持つ人はいない，

③ ２０歳未満の人は飲酒を許されていない

から命題

④ Ａ氏の息子で飲酒を許された人はいない

を導く推論は限量的に正しいものであるか？　すなわち、④は①～③の限量的帰結であるか？



第3章　公理化 ― 知識の体系化 ―

推論は帰納的なものと演繹的なものに大別できる。このうち**帰納的推論**(inductive inference)は、観察や実験を通して得られた事実の積み重ねから一般的な法則や命題を導くものであり、例えば数学や物理学,生物学,経済学などの分野で一般法則を発見するのに用いられてきた。帰納的推論は全く新しい法則を導き、我々の知識を拡大する。しかしその反面、これによって得られた結論,すなわち法則は必然的に真という訳ではなく、“量子の世界における古典力学の基本法則”の様に新しい事実によって反証されることもある。

これに対し**演繹的推論**(deductive inference)は、帰納とは逆に一般的な法則から特殊な事柄を導き出すものであり、また広義には、1～2章で考えた“論理的に正しい推論”の様に真の前提からは必然的に真の結論を導く形式的推論のことである。当然、この推論は新しい事実を導き出すことはない。しかし、演繹的推論は事実や知識の組織化／体系化に役立ち、我々自身も演繹による知識の体系化を無意識のうちに行っている。すなわち我々人間は、個々の事実を全て知識として記憶しているわけではなく、いくつかの事実を基により一般的な法則を知識として構成／記憶し様々な場面でこれらの法則を適用して必要な結果を導いていると思える。

以下、この章では演繹を用いてどの様に事実や知識を体系化できるか議論しよう。

3.1　公理と演繹による数学の体系化

演繹的推論を用いて事実／知識を体系づけた最初の例としては、古代ギリシア時代のユークリッド(Euclid,紀元前300頃)の幾何学が有名である。古代エジプトでは、ナイル川の氾濫に伴って境界線を引き直すという作業が毎年のように繰り返され、経験的に,従って帰納的推論によって図形に関する様々の有用な原則が発見されていった。しかし、これらの知識を幾何学と名付け理論的に究明していったのはギリシア人たちである。タレス(Thales,紀元前639?-546?),ピタゴラス(Pythagoras,紀元前582?-497?),プラトン(Plato,紀元前427?-347?)を始めとするギリシアの思想家たちは、一般

法則の厳密な演繹的証明を試み、さらに多くの幾何学的原則を発見していった。そして、ユークリッドがこれらの結果を系統的に網羅して“原論(The Elements)”と題する書物全13巻にまとめたのである。それ以来19世紀に至るまで、この“原論”は単に幾何学の教科書としてだけでなく科学的な考え方の規範としても強い影響力を保持し続けてきた。[実際には“原論”は数論も扱っており、例えば計算機科学関係では、2つの整数の最大公約数を求めるための計算法で現在“ユークリッドの互除法”という名で親しまれているものについても（第7巻第2項で）言及している。]

それでは、ユークリッドはどの様な手法で幾何学的知識を体系づけたのだろうか？ まず第一に彼は、特定の線や図形の性質を議論することはなく、常に幾何学的法則を普遍的でしかも厳密な形で定式化している。そして、これらの法則をただ単に並べただけでは満足せず、これらを系統的な順序で並べ演繹的に証明してゆく。これによって、ユークリッドは絶対的論理的必然性を持った厳密さで幾何学の諸法則を確立しようとしたのである。もっと具体的に言うと、“原論”第1巻はまず、幾何学における基本的な23個の用語の定義，幾何学において明らかに正しいと思える5つの原則（公準と呼んだ），幾何学に限らず他の分野にも共通して明らかに正しいと思える9つの原則（共通概念または公理と呼んだ）を列挙している。そして、演繹的推論によってこれらの定義，公準，共通概念から様々な結論を導き、さらにはこの様な厳密な手法によって幾何学的法則の間の論理的なつながりを明確に示そうとしているのである。[“原論”第1巻の定義としては例えば

1．点とは部分を持たないものである，
2．線とは幅のない長さである，
3．線の端は点である

というものがあり、公準としては

1．任意の一点から他の任意の一点へ直線を引くことができる，
2．線分はどこまでも延長できる，
3．任意の点と距離（半径）をもって円を描くことができる，
4．全ての直角は互いに等しい，
5．一つの直線が二つの直線と交わりその一方の側にできる二つの内角を合わせて2直角より小さくなる時は、これら二つの直線はどこまでも延長すれば合わせて2直角より小さくなる内角の側で交わる

というものがあり、共通概念としては例えば

1．同じものに等しいものは互いに等しい（すなわち $a=c, b=c$ ならば $a=b$），
2．等しいものに等しいものを加えれば、全体は等しい（すなわち $a=b, p=q$ ならば $a+p=b+q$），
3．等しいものから等しいものを引けば、残りは等しい

というものがある。また、これらの定義，公準，共通概念から導かれる結果としては第1巻第47項のピタゴラスの定理が有名である。]

現在では、一般に“原論”における公準や共通概念の様に証明なしで仮定される事柄を公理(axiom)と呼び、定義と公理に基づいて様々の結論を演繹する体系を公理系(axiom system)と呼ぶ。ユークリッドの時代には公理は“万人が正しいと認める命題”でなければならなかったが、今では公理は単なる仮定にすぎないのである。実際、この“公理”に対する見方の変化はヒルベルト(D.Hilbert,1862-1943)の**公理主義**(axiomatism)，すなわち

“数学は、定義なしの用語に関する若干の命題を正しいものと仮定し、
それらの仮定に基づいて形式的に推論を進めることによって命題間の形式的な依存関係を追求する学問である”

という考えによってもたらされた。公理主義によって数学は“現象世界の真理性を追求する学問”から厳密で形式的,一般的,抽象的な理論へと変貌することになるが、同時に数学はその抽象性により非常に広範な応用分野を持つことになる。それゆえ、公理主義は１９世紀の終わり頃に芽生えてから次第に浸透し、今や現代数学を支配する基本思想となっているのである。

それでは、公理主義は現代数学の中にどの様に入り込んでいるのであろうか？　まず幾何学においては、公理と演繹によるアプローチがユークリッド以来の習慣になっていて、そのおかげで１９世紀初頭に非ユークリッド幾何学，すなわちユークリッドの５番目の公準の成立を仮定しない幾何学が生まれることになった。そして、１８９９年にヒルベルトにより，またその後ヴェブレン(O.Veblen)やハンティングトン(E.V.Huntington)によっても、ユークリッドの幾何学の厳密な公理系が与えられた。［厳密に言えば、ユークリッドの証明の中には仮定が不十分なために形式論理だけでは結論の導けない箇所がたくさんあった。］ヒルベルトは、公理をただの仮定と考え、これらの仮定の下でユークリッド幾何学を再構成し、同時に副産物として得られた他の幾何学との関係も明らかにしたのであった。

一方、幾何学以外の数学の研究においても、非ユークリッド幾何学の出現に刺激されて公理と演繹によるアプローチがとられる様になった。例えば、自然数の公理系としては１８９１年にイタリアの数学者ペアノ(G.Peano)の与えたものが有名である。彼の公理系を言葉で表すと次の様になる。

１．ゼロは自然数である。
２．任意の自然数に対して、その後者(successor;次の数)も自然数である。
３．ゼロはいかなる自然数の後者でもない。
４．二つの自然数の後者が等しければ、元の二つの自然数はやはり等しい。
５．もしある事柄がゼロについて成り立ち、また“それが１つの自然数について成り立つ時必ずその後者についても成り立つ”ならば、その事柄はあらゆる自然数について成り立つ。

［ここで、自然数nの後者とは通常我々が$n+1$で表す数を指し、５番目の公理は数学的帰納法の原理を保証するものである。］　また代数学においては、１９１０年にシュタイニッツ(E.Steini-

tz)によって体の理論が公理化されたのを始めとして群や環の理論も公理化され、現代の抽象代数学へと発展していった。さらに確率論においては、１９３３年にコルモゴロフ(A.N.Kolmogorov)による“測度論”的な公理系が与えられ、これがそのまま現在の確率論へと発展して行った。

３．２　演繹の公理化 ── 演繹の機械化 ──

公理系を与えるには、通常、そこで許される公理（の形）を指定するだけで、用いる演繹の形態はあまり問題にしない。演繹の各段階で“万人が常に正しいと認める推論”を駆使するだけである。しかし、厳密さを要求する場合には、演繹の形態自体も明確に公理化しなければならない。 単に“一階論理で正しい,すなわち限量的帰結の推論だけを許す”というのでは推論の形が明示されたことにならず、演繹形態が明確に公理化されたとは言い難い。

そこで、真理関数的帰結,限量的帰結の演繹形態をどの様に公理化すれば良いか考えてみよう。⊨に関する演繹定理(定理１．１６,定理２．２６)によれば、Γが有限の場合には $\Gamma \models \alpha$ であるかどうかを判定する問題は恒真性判定の問題に帰着できた。それゆえ“恒真な論理式を演繹する公理系を形式的に記述できれば、真理関数的または限量的帰結の演繹形態を間接的に公理化したことになるのではないか？”と考えるのが自然であり、また、事実この考えは正しいものである。なぜなら、γ というパターンと $\gamma \to \delta$ というパターンから δ を導く推論型式，すなわち

$$\frac{\gamma,\ \gamma \to \delta}{\delta} \quad \begin{array}{l}（前提）\\（結論）\end{array}$$

を形式的な演繹規則として許すことにより、例えば“$\mathcal{B}$が$\mathcal{A}$の真理関数的帰結である”という事実によって正しさの保証される演繹は、恒真式を真理関数的構造とする命題とこの形式的な演繹規則の組み合わせで代用できるからである。実際、命題 $\mathcal{A}, \mathcal{B}$ の真理関数的構造を抽出してそれぞれ α, β という命題論理式が得られるとすると、真理関数的帰結の仮定と⊨に関する演繹定理より $\alpha \to \beta$ が恒真，従って $\mathcal{A} \to \mathcal{B}$ が真理関数的に真であることが分かる。ここで、$\mathcal{A}$が既に演繹されているとすると、これと真理関数的に真な命題 $\mathcal{A} \to \mathcal{B}$ を基に上のパターンを持つ推論

$$\frac{\mathcal{A},\ \mathcal{A} \to \mathcal{B}}{\mathcal{B}}$$

を適用して$\mathcal{B}$という結論を得ることができる。

結局、命題論理や一階論理で正しさの保証された演繹形態を明確に公理化するためには、恒真な論理式を演繹する形式的な公理系を与えればよいのである。そこで、続く３.３～４節ではそれぞれ恒真な命題論理式,恒真な一階論理式を演繹するための公理系を形式的に与えよう。

3．3　命題論理の公理系

前節で述べたように、この節では恒真な命題論理式を演繹するための公理系をいくつか与える。そして、これらの公理系の下での演繹過程を形式化しよう。

恒真式を演繹するための公理系としては、まず１８７９年にフレーゲ(G.Frege)により、そしてその後ラッセル(B.Russel),ヒルベルト,アッケルマン(W.Ackermann),ウカシェヴィッチ(J.Lukasiewicz),ゲンツェン(G.Gentzen),ロサー(J.R.Rosser),ニコド(J.Nicod),メレディス(C.A.Meredith),クリーネ(S.C.Kleene),タルスキー(A.Tarski)等によって様々なものが与えられている。例えば、次の通り。

公理系 R．（ラッセル,1908年）　この公理系では、基本的な論理結合子として $\sim$ と $\vee$ の２つだけを考え、残りについては

$\alpha \wedge \beta$ は $\sim(\sim\alpha \vee \sim\beta)$ の略記,

$\alpha \rightarrow \beta$ は $\sim\alpha \vee \beta$ の略記,

$\alpha \leftrightarrow \beta$ は $\sim(\sim(\sim\alpha \vee \beta) \vee \sim(\sim\beta \vee \alpha))$ の略記,

$\alpha \oplus \beta$ は $\sim(\sim\alpha \vee \beta) \vee \sim(\sim\beta \vee \alpha)$ の略記

と考える。この体系を構成する公理と推論規則は次の通り。

（公理）　次の(R1)〜(R5)の形の命題論理式を公理とする。

(R1)　$\alpha \vee \alpha \rightarrow \alpha$

(R2)　$\beta \rightarrow \alpha \vee \beta$

(R3)　$\alpha \vee \beta \rightarrow \beta \vee \alpha$

(R4)　$\alpha \vee (\beta \vee \gamma) \rightarrow \beta \vee (\alpha \vee \gamma)$

(R5)　$(\beta \rightarrow \gamma) \rightarrow (\alpha \vee \beta \rightarrow \alpha \vee \gamma)$

（推論規則）　次の形の推論，すなわち $\alpha, \alpha \rightarrow \beta$ という形の２つの命題論理式から β を導く推論型式を演繹として許す。［この(MP)を**分離規則**(detachment rule,**モーダスポネンス**, modus ponens)という。］

(MP)　$\dfrac{\alpha,\ \alpha \rightarrow \beta}{\beta}$

公理系 L．（ウカシェヴィッチ,1930年）　この公理系は、１８７９年のフレーゲの体系を整理して得られたものであり、基本的な論理結合子として $\sim$ と $\rightarrow$ の２つだけを持つ。この体系を構成する公理と推論規則は次の通り。

（公理）　次の(L1)〜(L3)の形の命題論理式を公理とする。

(L1)　$\alpha \rightarrow (\beta \rightarrow \alpha)$

(L2) $(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))$

(L3) $(\sim\alpha\to\sim\beta)\to(\beta\to\alpha)$

（推論規則）　次の形の推論を演繹として許す。

(MP) $\dfrac{\alpha,\ \alpha\to\beta}{\beta}$

公理系 H.（ヒルベルト,1934年）　この公理系は、基本的な論理結合子として $\sim,\wedge,\vee,\to,\leftrightarrow$ の５つを持ち、次の公理と推論規則から構成される。

（公理）　次の(H1)～(H12)の形の命題論理式を公理とする。

(H1) $\alpha\to(\beta\to\alpha)$ 　　(→に関する公理)

(H2) $(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))$

(H3) $\alpha\wedge\beta\to\alpha$ 　　(∧に関する公理)

(H4) $\alpha\wedge\beta\to\beta$

(H5) $\alpha\to(\beta\to\alpha\wedge\beta)$

(H6) $\alpha\to\alpha\vee\beta$ 　　(∨に関する公理)

(H7) $\beta\to\alpha\vee\beta$

(H8) $(\alpha\to\gamma)\to((\beta\to\gamma)\to(\alpha\vee\beta\to\gamma))$

(H9) $(\alpha\leftrightarrow\beta)\to(\alpha\to\beta)$ 　　(↔に関する公理)

(H10) $(\alpha\leftrightarrow\beta)\to(\beta\to\alpha)$

(H11) $(\alpha\to\beta)\to((\beta\to\alpha)\to(\alpha\leftrightarrow\beta))$

(H12) $(\sim\alpha\to\sim\beta)\to(\beta\to\alpha)$ 　　(～に関する公理)

（推論規則）　次の形の推論を演繹として許す。

(MP) $\dfrac{\alpha,\ \alpha\to\beta}{\beta}$

公理系 M.（メレディス,1953年）　この公理系は、基本的な論理結合子として ～ と → の２つだけを持ち、次の公理と推論規則から構成される。

（公理）　次の(M1)の形の命題論理式を公理とする。

(M1) $((((\alpha\to\beta)\to(\sim\gamma\to\sim\delta))\to\gamma)\to\varepsilon)\to((\varepsilon\to\alpha)\to(\delta\to\alpha))$

（推論規則）　次の形の推論を演繹として許す。

(MP) $\dfrac{\alpha,\ \alpha\to\beta}{\beta}$

公理系が与えられた時、演繹過程を形式的に扱うために、一般に、その公理系で導かれる論理式（または命題）を“定理”と呼び、定理を導く過程を“証明”と呼ぶ。より形式的には、公理系$\mathcal{L}$

に関して、論理式（または命題）の列 $\alpha_1, ..., \alpha_n$ が条件

どの α_i も

(i) $\mathfrak{L}$の公理であるか

(ii) α_iより前に現れる論理式(または命題)に$\mathfrak{L}$の推論規則を１回だけ適用して得られる

を満たす時、この論理式（または命題）の列を$\mathfrak{L}$の**証明**(proof)という。そして、αを最後の要素とする$\mathfrak{L}$の証明がある時、αを$\mathfrak{L}$の**定理**(theorem)であるといい $\vdash_{\mathfrak{L}} \alpha$ と書く。

この形式的定義に従えば、証明それ自体は単に論理式（または命題）の列にすぎず、定理の導かれる全体像を直観的に分かり易く表したものではない。しかし、図を用いて証明を直観的に分かり易く表すこともできる。例えば、証明の中で用いられる個々の推論を

$$(1)\quad \frac{\square\square\square, \ ... \ , \triangle\triangle\triangle}{\bigcirc\bigcirc\bigcirc}$$

（推論の前提となる論理式や命題の列）
（推論の結論となる論理式，または命題）

という形に表し、これらの中の同一の論理式,同一の命題を繰り返し重ね合わせることにより、**証明図**(proof figure)と呼ばれるものができる。証明図は、(1)の形の推論を２次元平面上に有機的に組み合わせて構成されるものであり、証明の全体像を見渡したい時に有用である。例えば、ある証明の中に２つの推論

$$\frac{p, \ q}{r} \quad と \quad \frac{r, \ s, \ t}{u}$$

が用いられたとすると、これら２つを組み合わせて次の様な証明図が得られる。

$$\frac{\dfrac{p, \ q}{r}, \quad s, \ t}{u}$$

例3.1　任意の命題論理式αに対して、次の命題論理式の列(2)〜(6)は公理系Ｌの証明である。

(2) $(\alpha\to((\alpha\to\alpha)\to\alpha))\to((\alpha\to(\alpha\to\alpha))\to(\alpha\to\alpha))$　(L2)の形の公理

(3) $\alpha\to((\alpha\to\alpha)\to\alpha)$　(L1)の形の公理

(4) $(\alpha\to(\alpha\to\alpha))\to(\alpha\to\alpha)$　(1),(2)に規則(MP)を適用

(5) $\alpha\to(\alpha\to\alpha)$　(L1)の形の公理

(6) $\alpha\to\alpha$　(3),(4)に規則(MP)を適用

従って、$\alpha\to\alpha$ は公理系Ｌの定理，すなわち $\vdash_L \alpha\to\alpha$ である。そして、この(2)〜(6)の証明に対して証明図を構成すると、図３．１の様なものが得られる。

$$\frac{\overset{(L1)}{\alpha\to(\alpha\to\alpha)},\qquad \dfrac{\overset{(L2)}{\alpha\to((\alpha\to\alpha)\to\alpha)},\quad \overset{(L1)}{(\alpha\to((\alpha\to\alpha)\to\alpha))\to((\alpha\to(\alpha\to\alpha))\to(\alpha\to\alpha))}}{(\alpha\to(\alpha\to\alpha))\to(\alpha\to\alpha)}\,(MP)}{\alpha\to\alpha}\,(MP)$$

図３．１　例３．１の証明(2)～(6)に関する証明図

演繹過程の途中で何らかの仮定を行うこともある。一般に、その途中で仮定される論理式（または命題）の集合をΓ,演繹の基になる公理系を$\mathfrak{L}$とし、$\mathfrak{L}$にΓの論理式（または命題）を公理として付け加えて得られる公理系を $\mathfrak{L}[\Gamma]$ で表すことにすれば、この演繹過程は $\mathfrak{L}[\Gamma]$ における証明に他ならない。それゆえ、“論理的帰結”という用語の使い方に倣って、$\vdash_{\mathfrak{L}[\Gamma]}\alpha$ の時に、論理式（または命題）αはΓから**演繹可能**(deducible)であるといい $\Gamma\vdash_{\mathfrak{L}}\alpha$ と書く。特にΓが有限で $\Gamma=\{\beta_1,\ \ldots\ ,\beta_n\}$ の時、$\Gamma\vdash_{\mathfrak{L}}\alpha$ の代わりに $\beta_1,\ \ldots\ ,\beta_n\vdash_{\mathfrak{L}}\alpha$ と略記する。

次の演繹定理は、命題論理式が公理系 R, L, H, M から導かれるかどうかを調べる際に有用である。

定理３．２(演繹定理)　　任意の命題論理式 α,β と命題論理式の集合Γに関して

(i) $\Gamma\cup\{\alpha\}\vdash_{\mathbf{R}}\beta \iff \Gamma\vdash_{\mathbf{R}}\alpha\to\beta$

(ii) $\Gamma\cup\{\alpha\}\vdash_{\mathbf{L}}\beta \iff \Gamma\vdash_{\mathbf{L}}\alpha\to\beta$

(iii) $\Gamma\cup\{\alpha\}\vdash_{\mathbf{H}}\beta \iff \Gamma\vdash_{\mathbf{H}}\alpha\to\beta$

(iv) $\Gamma\cup\{\alpha\}\vdash_{\mathbf{M}}\beta \iff \Gamma\vdash_{\mathbf{M}}\alpha\to\beta$

例３．３　　明らかに $\alpha\vdash_{\mathbf{L}}\alpha$ であり、これに演繹定理を適用すれば、直ちに例３．１の結果 $\vdash_{\mathbf{L}}\alpha\to\alpha$ を導くことができる。

最後に、上で述べた R, L, H, M が恒真式を演繹するための公理系であることは、次の**無矛盾性&完全性定理**(consistency and completeness theorem)によって保証される。

定理３．４(無矛盾性&完全性定理)　　任意の命題論理式 α に関して

αが恒真（すなわち $\models\alpha$ ） $\iff \vdash_{\mathbf{R}}\alpha$

$\iff \vdash_{\mathbf{L}}\alpha$

$\iff \vdash_{\mathbf{H}}\alpha$

$\iff \vdash_{\mathbf{M}}\alpha$

3．4 一階論理の公理系

この節では、恒真な一階論理式を演繹するための公理系をいくつか与え、これらの公理系の下での演繹過程の形式化について考えよう。

恒真な一階論理式を演繹するための公理系としては、ラッセル,アッケルマン,ウカシェヴィッチ,ベルナイス(P.Bernays),プライア(A.N.Prior),ゲンツェン等によって様々なものが与えられている。例えば、次の通り。[これらの公理系は、前節で紹介した“恒真な命題論理式を演繹するための公理系”に、限量子に関する公理と推論規則を付け加えて得られる。限量子としては互いに双対な $\forall$ と $\exists$ しかないから、限量子に関する公理と推論規則の採り方は前節3.3の場合の様に多彩にならない。]

公理系 R＊.（ラッセル,1908年） この公理系は、前節3.3のラッセルの体系Rの拡張であり、基本的な論理結合子,限量子として $\sim$, $\vee$ と $\forall$ の3つだけを持つ。この体系を構成する公理と推論規則は次の通り。[ここでは、もちろん $\alpha \to \beta$ は $\sim\alpha \vee \beta$ の略記, $\exists \boldsymbol{x}\alpha$ は $\sim\forall \boldsymbol{x}\sim\alpha$ の略記と考える。]

（公理） 次の(R1)～(R7)の形の一階論理式を公理とする。

(R1) $\alpha \vee \alpha \to \alpha$

(R2) $\beta \to \alpha \vee \beta$

(R3) $\alpha \vee \beta \to \beta \vee \alpha$

(R4) $\alpha \vee (\beta \vee \gamma) \to \beta \vee (\alpha \vee \gamma)$

(R5) $(\beta \to \gamma) \to (\alpha \vee \beta \to \alpha \vee \gamma)$

(R6) $\forall \boldsymbol{x}(\alpha \to \beta) \to (\alpha \to \forall \boldsymbol{x}\beta)$

但し、α は自由変項として $\boldsymbol{x}$ を含まない。

(R7) $\forall \boldsymbol{x}\alpha(\ \ldots\ , \boldsymbol{x},\ \ldots\ ) \to \alpha(\ \ldots\ , t,\ \ldots\ )$

但し、$\boldsymbol{x}$ に置き換わる t 内の変項はどれも置き換えによって新たに α 内の限量子で修飾されることはない。

（推論規則） 次の２つの形の推論を演繹として許す。[この内(Gen)を**一般化規則**(generalization rule)という。]

(MP) $\dfrac{\alpha,\ \alpha \to \beta}{\beta}$

(Gen) $\dfrac{\alpha}{\forall \boldsymbol{x}\alpha}$

公理系 ＬＲ＊．（ウカシェヴィッチ1930年＋ラッセル1908年） この公理系は、前節３．３のウカシェヴィッチの体系Ｌに限量子に関するラッセルの公理,推論規則を付け加えて得られるものであり、基本的な論理結合子,限量子として ～,→ と ∀ の３つだけを持つ。この体系を構成する公理と推論規則は次の通り。

（公理） 次の (L1)～(L3),(R6),(R7) の形の一階論理式を公理とする。

(L1) $\alpha\to(\beta\to\alpha)$

(L2) $(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))$

(L3) $(\sim\alpha\to\sim\beta)\to(\beta\to\alpha)$

(R6) $\forall \boldsymbol{x}(\alpha\to\beta)\to(\alpha\to\forall \boldsymbol{x}\beta)$

但し、αは自由変項として$\boldsymbol{x}$を含まない。

(R7) $\forall \boldsymbol{x}\alpha(\ \ldots,\boldsymbol{x},\ \ldots\ )\to\alpha(\ \ldots,t,\ \ldots\ )$

但し、$\boldsymbol{x}$に置き換わるt内の変項はどれも置き換えによって新たにα内の限量子で修飾されることはない。

（推論規則） 次の２つの形の推論を演繹として許す。

(MP) $\dfrac{\alpha,\ \alpha\to\beta}{\beta}$

(Gen) $\dfrac{\alpha}{\forall \boldsymbol{x}\alpha}$

公理系 ＬＢ＊．（ウカシェヴィッチ1930年＋ベルナイス1928年以前） この公理系は、前節３．３のウカシェヴィッチの体系Ｌに限量子に関するベルナイスの公理,推論規則を付け加えて得られるものであり、基本的な論理結合子,限量子として ～,→ と ∀ の３つだけを持つ。この体系を構成する公理と推論規則は次の通り。

（公理） 次の (L1)～(L3),(R7) の形の一階論理式を公理とする。

(L1) $\alpha\to(\beta\to\alpha)$

(L2) $(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))$

(L3) $(\sim\alpha\to\sim\beta)\to(\beta\to\alpha)$

(R7) $\forall \boldsymbol{x}\alpha(\ \ldots,\boldsymbol{x},\ \ldots\ )\to\alpha(\ \ldots,t,\ \ldots\ )$

但し、$\boldsymbol{x}$に置き換わるt内の変項はどれも置き換えによって新たにα内の限量子で修飾されることはない。

（推論規則） 次の２つの形の推論を演繹として許す。［この内(∀右)を∀-**導入規則**(∀-introduction rule)という。］

(MP) $\dfrac{\alpha,\ \alpha\to\beta}{\beta}$

$$(\forall 右)\ \frac{\alpha \to \beta}{\alpha \to \forall \boldsymbol{x}\, \beta}$$　但し、αは自由変項として$\boldsymbol{x}$を含まない。

公理系 LP＊. (ウカシェヴィッチ1930年＋プライア1955年)　この公理系は、前節3．3のウカシェヴィッチの体系Lに限量子に関するプライアの（公理,）推論規則を付け加えて得られるものであり、基本的な論理結合子,限量子として ～,→ と ∀ の3つだけを持つ。この体系を構成する公理と推論規則は次の通り。

（公理）　次の (L1)～(L3) の形の一階論理式を公理とする。

(L1) $\alpha \to (\beta \to \alpha)$

(L2) $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$

(L3) $(\sim\alpha \to \sim\beta) \to (\beta \to \alpha)$

（推論規則）　次の3つの形の推論を演繹として許す。［この内(∀左)も∀-導入規則という。］

$$(MP)\ \frac{\alpha,\ \alpha \to \beta}{\beta}$$

$$(\forall 右)\ \frac{\alpha \to \beta}{\alpha \to \forall \boldsymbol{x}\, \beta}$$　但し、αは自由変項として$\boldsymbol{x}$を含まない。

$$(\forall 左)\ \frac{\alpha \to \beta}{\forall \boldsymbol{x}\, \alpha \to \beta}$$

“定理”や“証明”，“証明図”，“演繹可能”といった用語の定義は、もちろん前節3．3に限ったことではなく、一階論理でもやはり有効である。そして、ここで与えた4つの公理系についても、演繹定理や無矛盾性＆完全性定理は成立する。

定理3．5 (演繹定理)　　任意の一階論理式 α, β と一階論理式の集合Γに関して

(i) $\Gamma \cup \{\alpha\} \vdash_{R*} \beta \iff \Gamma \vdash_{R*} \alpha \to \beta$

(ii) $\Gamma \cup \{\alpha\} \vdash_{LR*} \beta \iff \Gamma \vdash_{LR*} \alpha \to \beta$

(iii) $\Gamma \cup \{\alpha\} \vdash_{LB*} \beta \iff \Gamma \vdash_{LB*} \alpha \to \beta$

(iv) $\Gamma \cup \{\alpha\} \vdash_{LP*} \beta \iff \Gamma \vdash_{LP*} \alpha \to \beta$

定理3．6 (無矛盾性＆完全性定理)　　任意の一階論理式 α に関して

αが恒真（すなわち $\vDash \alpha$ ）$\iff \vdash_{R*} \alpha$

$\iff \vdash_{LR*} \alpha$

$\iff \vdash_{LB*} \alpha$

$\iff \vdash_{LP*} \alpha$

3．5 知識ベース — 計算機内での知識の体系化 —

一般に、様々な種類の知識（例えば事実，演繹規則，動作の手順に関する情報など）を計算機の中に体系的に蓄積したものを**知識ベース**(knowledge base)といい、知識ベースの中の知識を用いてある特定の問題を知的に解決（または、それを支援）するシステムを**知識ベースシステム**(knowledge-based system，**知識準拠**システム)という。そして、知識ベースシステムのうち、特に、その知識ベースがインタビューなどを通して専門家から抽出された知識から構成され、専門家とほぼ同レベルの能力を持つものをエキスパートシステム(expert system，**専門家**システム)と呼ぶ。 ［普通は、知識ベースだけを単独に構成することはない。従って、より正確に言うなら、知識ベースシステムがあった時、その中で知識を体系的に蓄積した部分を**知識ベース**と呼び、推論を行う部分を**推論エンジン**(inference engine)と呼ぶ。そして、知識ベースシステムのうち、比較的小規模なものを**知識システム**(knowledge system)と呼んでいる。しかし、最近では“知識ベースシステム”，“エキスパートシステム”，“知識システム”という言葉を同義で用いたり、“知識システム”という言葉を人工知能の応用全般を意味するものとして広義に用いることもあるので、ここでは“知識ベース”という言葉も広義に解釈する。 また、知識ベースに対し、データの定義に従ってデータを組織的に計算機の中に蓄積したものをデータベース(database)と呼ぶ。データベースは、通常、その中の大量のデータを高速に，しかも同時更新を許しながら用いることに重点を置いて構成され、知的水準の高いデータの加工にはあまり利用できない。］

知識を計算機の中で体系的に表現するために、一階または命題論理の表記法を用いることも多い。この知識表現方式では、様々な知識を一階または命題論理式風の表現，すなわち“論理式と同じ構造を持ち言明文字，個体文字，関数文字，述語文字の代わりに意味の固定された文字列を含むもの”で表すことによって、計算機の中に公理系を構成する。例えば、身近な事柄についての知識は一階または命題論理の表記法を用いて次の様に表すことができる。

例3．7(お皿の上の世界) お皿の上にレモンと林檎とトマトが置かれている。これら3つのうち林檎とトマトを果物と思っている人にとっては、このお皿の上の状況は、例えば5つの公理と1つの推論規則からなる公理系

（公理） lemon,
apple,
tomato,
apple→fruit,
tomato→fruit,

（推論規則）

$$(\mathrm{MP})\ \frac{\alpha,\ \alpha\to\beta}{\beta},$$

あるいは3つの公理と2つの推論規則からなる公理系

（公理）　lemon,

apple,

tomato,

（推論規則）

$$\frac{\mathrm{apple}}{\mathrm{fruit}},$$

$$\frac{\mathrm{tomato}}{\mathrm{fruit}}$$

によって表すことができる。但し、ここでは演繹に関する公理と推論規則の大部分を省いている。また、lemon,apple,tomato,fruit は、それぞれ“お皿の上にレモンがある”，“お皿の上に林檎がある”，“お皿の上にトマトがある”，“お皿の上に果物がある”という命題を表す文字列である。

これら2つの公理系のうち最初の公理系を用いることにすれば、例えば fruit という式は次の証明(1)～(3)の様に，すなわち図3.2の証明図の様に導かれる。

(1) apple　　　　２番目の公理

(2) apple→fruit　４番目の公理

(3) fruit　　　　(1),(2)に推論規則(MP)を適用

（公理）　（公理）

$$\frac{\mathrm{apple},\ \mathrm{apple}\to\mathrm{fruit}}{\mathrm{fruit}}\ (\mathrm{MP})$$

図3.2　例3.7の証明(1)～(3)に関する証明図

例3.8（ブレーメンの音楽隊；後藤［25］）　グリム童話に“ブレーメンの音楽隊”というのがある。年を取りすぎて処分されそうになった驢馬と犬と猫，それとスープにされそうになった雄鶏が音楽隊に入るために一緒にブレーメンを目指すという話である。動物達は、ブレーメンへ行く途中で泥棒の隠れ家を見つけ、泥棒達を追い出して結局そこに住み着くことになる。この物語の中で、特に“動物達が泥棒達を驚かして追い出すために、犬が驢馬の背中に飛び乗り，猫が犬の背中によじ登り，雄鶏が猫の頭に止まり、みんなで一斉に鳴き叫ぶ”、という場面に注目してこの時の動物達の位置関係を公理系で表すことにすれば、例えば次の様なものが得られる。［但し、ここでも演繹に関する公理と推論規則の大部分を省いている。また、一階原子論理式風の表現 on(x, y), above(x,

y）は、それぞれ“xはyの上に乗っている”，“xはyの上方にいる”という命題を表す。］

（公理）　on(cock, cat),
　　　　　on(cat, dog),
　　　　　on(dog, donkey),
　　　　　$\mathrm{on}(x, y) \to \mathrm{above}(x, y)$　　但し x, y は任意,
　　　　　$\mathrm{on}(x, z) \wedge \mathrm{above}(z, y) \to \mathrm{above}(x, y)$　　但し x, y, z は任意,

（推論規則）

$$(\mathrm{MP})\ \frac{\alpha,\ \alpha \to \beta}{\beta}\quad,$$

あるいは次の通り。

（公理）　on(cock, cat),
　　　　　on(cat, dog),
　　　　　on(dog, donkey),

（推論規則）　$\dfrac{\mathrm{on}(x, y)}{\mathrm{above}(x, y)}$　　但し x, y は任意,

$$\frac{\mathrm{on}(x, z),\ \mathrm{above}(z, y)}{\mathrm{above}(x, y)}\quad \text{但し } x, y, z \text{ は任意}$$

これら２つの公理系のうち２番目の公理系を用いることにすれば、例えば above(cat, donkey) という式は次の証明(4)～(7)の様に，すなわち図３．３の証明図の様に導かれる。

(4)　on(cat, dog)　　　　　２番目の公理
(5)　on(dog, donkey)　　　３番目の公理
(6)　above(dog, donkey)　　(5)の式に１番目の推論規則を適用
(7)　above(cat, donkey)　　(4), (6)の式に２番目の推論規則を適用

$$\frac{\overset{(\text{公理})}{\mathrm{on(cat, dog)}},\quad \dfrac{\overset{(\text{公理})}{\mathrm{on(dog, donkey)}}}{\mathrm{above(dog, donkey)}}\ (\text{1番目の推論規則})}{\mathrm{above(cat, donkey)}}\ (\text{2番目の推論規則})$$

図３．３　例３．８の証明(4)～(7)に関する証明図

例３．９（ペアノの公理系の下での加算）　３．１節で紹介したペアノの公理系の下では、全ての自然数が“ゼロ”と“後者(successor)”という２つの言葉だけを用いて表される。例えば２は“ゼロの後者の後者”という言い方で表される。今、簡単のため“ゼロ”を 0 という文字で表し“x

の後者”という言い方を s(x) という関数記法で表すことにすれば、この自然数の表記法の下では、加算に関する事実を例えば次の様な公理系で表すことができる。［但し、ここでも演繹に関する公理と推論規則を省いている。また、一階原子論理式風の表現 add(x, y, z) は “ $x+y=z$ ” という命題を表す。］

（公理）　add(０, x, x)　　但し x は任意,

（推論規則）　$\dfrac{\text{add}(x, y, z)}{\text{add}(\text{s}(x), y, \text{s}(z))}$　　但し x, y, z は任意

この公理系を用いることにすれば、例えば、add(s(s(0)),s(0),s(s(s(0)))) という式は次の証明(8)～(10)の様に，すなわち図３．４の証明図の様に導かれる。

(8) add(0, s(0), s(0))　　公理

(9) add(s(0), s(0), s(s(0)))　　(8)の式に推論規則を適用

(10) add(s(s(0)), s(0), s(s(s(0))))　　(9)の式に推論規則を適用

$$\dfrac{\dfrac{\overset{\text{(公理)}}{\text{add}(0,\ \text{s}(0),\ \text{s}(0))}}{\text{add}(\text{s}(0),\ \text{s}(0),\ \text{s}(\text{s}(0)))}}{\text{add}(\text{s}(\text{s}(0)),\ \text{s}(0),\ \text{s}(\text{s}(\text{s}(0))))}$$

図３．４　例３．９の証明(8)～(10)に関する証明図

例３．１０（自然言語の構文解析）　６つの文字列

(11) John dreams,

(12) John eats the cake,

(13) John eats John,

(14) the cake dreams,

(15) the cake eats the cake,

(16) the cake eats John

は、その意味を考えなければ、全て文法的に正しい英文と見ることができる。そして、そのために必要な文法知識は、例えば次の様な公理系で表すことができる。

（公理）　intransitive_verb(dreams),

transitive_verb(eats),

proper_noun(John),

definite_article(the),

noun(cake),

（推論規則） $\dfrac{\text{noun_phrase}(x),\ \text{verb_phrase}(y)}{\text{sentence}(x+y)}$ 但し x, y は任意,

$\dfrac{\text{proper_noun}(x)}{\text{noun_phrase}(x)}$ 但し x は任意,

$\dfrac{\text{definite_article}(x),\ \text{noun}(y)}{\text{noun_phrase}(x+y)}$ 但し x, y は任意,

$\dfrac{\text{intransitive_verb}(x)}{\text{verb_phrase}(x)}$ 但し x は任意,

$\dfrac{\text{transitive_verb}(x),\ \text{noun_phrase}(y)}{\text{verb_phrase}(x+y)}$ 但し x, y は任意

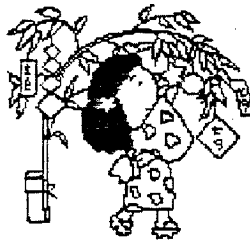
もちろん、ここでも演繹に関する公理と推論規則を省いている。また、この公理系では、関数構造

$x+y$ は“文字列 x と y をこの順につなげて得られる文字列”を

表し、一階原子論理式風の表現

intransitive_verb(x) は“xが自動詞である”という命題を,
transitive_verb(x) は“xが他動詞である”という命題を,
proper_noun(x) は“xが固有名詞である”という命題を,
definite_article(x) は“xが定冠詞である”という命題を,
noun(x) は“xが名詞である”という命題を,
sentence(x) は“xが文である”という命題を,
noun_phrase(x) は“xが名詞句である”という命題を,
verb_phrase(x) は“xが動詞句である”という命題を

表す。



演習問題３

３．１　　ユークリッドの"原論"［10］において、

(1) ピタゴラスの定理，

(2) ２つの整数の最大公約数の計算法

がどの様に記述／証明されているかを調べよ。

３．２　　命題$\mathcal{B}$が命題 $\mathcal{A}_1, \mathcal{A}_2$ の真理関数的帰結である時、$\mathcal{A}_1, \mathcal{A}_2$ から$\mathcal{B}$を導く演繹を

① 真理関数的に真な命題　と

② 分離規則　$\dfrac{\alpha,\ \alpha \to \beta}{\beta}$

の組み合わせで代用できることを示せ。

３．３　　任意の命題論理式αに対して、$\alpha \to \alpha$ が３．３節で与えた４つの公理系 R, L, H, M の定理であることを示せ。

３．４　　定理３．２，定理３．４〜６を証明せよ。

３．５　　例３．８で与えた２番目の公理系によって、above(cock, donkey) という式はどの様に導かれるか？　また、この証明図はどうなるか？

３．６　　例３．８で与えた１番目の公理系によって、above(cat, donkey) という式はどの様に導かれるか？　また、この証明図はどうなるか？

３．７　　例３．９で与えた公理系によって、add(s(s(s(0))), s(s(0)), s(s(s(s(s(0)))))) という式はどの様に導かれるか？　また、この証明図はどうなるか？

３．８　　例３．１０で与えた公理系によって、sentence(John + eats + the + cake) という式はどの様に導かれるか？　また、この証明図はどうなるか？

第4章　Ｐｒｏｌｏｇのプログラムとその実行

Prolog(Programmation en Logique,または Programming in Logic の略)は１９７２年にマルセイユで生まれたプログラミング言語であり、その文法はまだ統一されていない。しかし、１９７７年にエジンバラ大学で DEC-System 10 上に初めての実用的な処理系が開発されて以来、そこで定められた言語仕様（**DEC-10 Prolog** という）が事実上の標準となっている。　　　これから紹介する Prolog は日本電気のオペレーティングシステム ACOS-6/MVX Ⅱ（またはACOS-4/AVP XR,ACOS-4/MVP XE,ACOS-4/XVP）上で起動するものであるが、この言語仕様もほぼ DEC-10 Prolog に従っている。[Prolog の標準化のための国際的な活動は、DEC-10 Prolog を基礎言語として１９８８年に開始され、１９９２年の時点でやっと最終段階に入っているらしい。]

以下、この章では標準的な DEC-10 Prolog の言語仕様に従って、プログラムの例（４.１節）,プログラムの一般形（４.２節）,プログラムの動作原理（４.３節）,プログラムの作成／実行例（４.４節）を示す。

４.１　Ｐｒｏｌｏｇのプログラムと公理系

Prolog のプログラムは、一階または命題論理の表記法に基づく公理系，あるいは、これに（公理系の演繹可能性に関する）質問を加えたものと見なすことができる。　[ここで、プログラム内に“質問”の部分があれば、それがプログラムを起動させることになる。]

例４.１(お皿の上の世界)　　Prolog のプログラム

```
lemon.
apple.
tomato.
fruit :- apple.
fruit :- tomato.
```

は、例３．７で与えた公理系と同等のものと見なせる。［ここで、プログラムの各行がピリオドで終わっていることに注目しよう。ピリオドはプログラムの一部で、１つの公理や推論規則の記述がそこで終わることを表す。それゆえピリオドの省略はできない。］

例４．２(ブレーメンの音楽隊；後藤［25］)　Prolog のプログラム

```
on(cock,cat).
on(cat, dog).
on(dog, donkey).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

は、例３．８で与えた公理系と同等のものと見なせる。［ここで、例３．８の公理系内の x, y, z という文字が、このプログラムではそれぞれ X,Y,Z となっていることに注目しよう。］

例４．３(ペアノの公理系の下での加算)　Prolog のプログラム

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

は、例３．９で与えた公理系と同等のものと見なせる。

例４．４(自然言語の構文解析)　Prolog のプログラム

```
intransitive_verb(dreams).
transitive_verb(eats).
proper_noun('John').
definite_article(the).
noun(cake).
sentence(X+Y) :- noun_phrase(X), verb_phrase(Y).
noun_phrase(X) :- proper_noun(X).
noun_phrase(X+Y) :- definite_article(X), noun(Y).
verb_phrase(X) :- intransitive_verb(X).
verb_phrase(X+Y) :- transitive_verb(X), noun_phrase(Y).
```

は、例３．１０で与えた公理系と同等のものと見なせる。［ここで、例３．１０の公理系内の John という文字列が、このプログラムでは 'John' となっていることに注目しよう。］

4.2 Prologプログラムの一般形

Prolog のプログラムについて語る場合、一般にプログラム内で１つの公理や推論規則に相当する式を“ホーン節”と言い、さらに、ここでは一階論理における原子論理式風の式,命題論理における言明文字風の式を合わせて“**原子式**”と言うことにしよう。すると、例えば例４.１のプログラムは、lemon. と apple. と tomato. と fruit:-apple. と fruit:-tomato. という５つのホーン節を含み、lemon と apple と tomato と fruit という４つの原子式を含むことになる。また例４.３のプログラムは、add(0,X,X). と add(s(X),Y,s(Z)):-add(X,Y,Z). という２つのホーン節を含み、 add(0,X,X) と add(s(X),Y,s(Z)) と add(X,Y,Z) という３つの原子式を含む。

これらの用語を使うと、前節４.１の例からも分かる様に、Prolog のプログラムはホーン節を並べたものであり、さらにホーン節は

① 原子式とピリオドをこの順に並べたもの，または

② １個以上の原子式をコンマで区切りその後にピリオドを,その前に原子式と文字列 :- をこの順に並べたもの

であると言える。以下、原子式,ホーン節,Prologプログラムの一般形がどうなるか見てみよう。

まず原子式は、命題を一階論理における原子論理式風に,あるいは命題論理における言明文字風に表したものであるから、

① 個体文字,関数文字,述語文字,言明文字 の代わりにそれぞれ具体的な 個体,関数,述語(すなわち真理値を値とする関数),言明 を表す文字列が用いられ、また、

② 変項としては一階論理の場合の u, v, w, x, y, z, ... などと異なるものが用いられる

ことを除けば、原子論理式,あるいは言明文字と全く同様に構成される。すなわち、個体,関数,述語,言明の名前として使う文字列,変項を表す文字列を決めてしまえば、これに基づいて、原子式というものが原子論理式や言明文字の場合と全く同一の仕方で決まってしまう。

具体的には、個体,関数,述語,言明の名前としてはアトム(atom)と呼ばれる文字列，すなわち

① 英小文字で始まりそれに０個以上の英数字や下線文字(_)の続く文字列，

② 特殊文字(+-*/\^<>=`~:.?@#$&)の並び，または

③ 任意の文字の並びを１重引用符(')で囲んだもの

を用い、**変項**(variable)としては

英大文字か下線文字(_)で始まりそれに０個以上の英数字や下線文字(_)の続く文字列

を用いる。特に１個の下線文字だけから成る変項を**無名変項**(anonymous variable)という。普通の変項と違って、無名変項は他のどの変項とも異なるものとして扱われる。従って、１つのホーン節

の中に無名変項が複数個現れる場合でも、それらの無名変項に同一の項を割り当てる必要はない。［文字の並びを1重引用符(')で囲んでアトムを構成する場合は、元の文字の並び内の1個の1重引用符に対して連続した2個の1重引用符で代用する。 また、これから紹介する ACOS 上の Prolog では、アトムを構成するためにカレット(^),アクセント符(`),ティルデ(~)という3つの特殊文字を用いることはできず、さらにバックスラッシュ(\)の代わりに円記号(¥)を用いる。しかし、アトムを構成するのに漢字を始めとする2バイト文字も英小文字と同様に使うことができる。感嘆符(!),コンマ(,),セミコロン(;),“空リスト”を表す文字列 [],Prolog言語処理系で予約した特殊な文字列もアトムに分類されるが、このうち [] を述語や言明の名前として用いることはできない。］

定義4.5 間接的に個体を表すための表現，すなわち**項**(term)のクラスを次の(i),(ii)により帰納的に定める。

(i) 変項,(個体の名前である)アトム,整数を表す文字列（すなわち1個以上の数字の列,またはその前に符号を付けたもの）はどれも項である。

(ii) (n引数関数の名前である)アトム α とn個の項 $t_1, \ldots, t_n$ に対して、 $\alpha(t_1, \ldots, t_n)$ も項である。この形の項を特に**複合項**(compound term)という。

［これから紹介する ACOS 上の Prolog では、次の①～⑤のデータも項として定義される。もちろん、これらの項を用いて複合項を構成することもできる。

① 2～9進表示の整数 … 例えば、2進法の 1010 を 2'1010 と表す。

② 実数 … 例えば、数 4500 を浮動小数点表示で 4500.0, 4.5E3 あるいは 0.45e4 と表す。

③ 文字 … 例えば、文字 g を #\g と，空白を #\space と，改行コードを #\newline と表す。

④ 文字列 … 例えば、文字列 abc を "abc" と表す。

⑤ リスト(list;すなわち項の列) … 例えば、アトム 'ABC' と実数 0.5 と文字列 "ABC" から構成されるリストを ['ABC',0.5,"ABC"] と表す。一般には、リストの要素は任意の項である。］

定義4.6 次の (i),(ii) の式を**原子式**(atomic formula,**素式**)という。

(i) (言明の名前である)アトム,

(ii) (n引数述語の名前である)アトム α とn個の項 $t_1, \ldots, t_n$ から構成される式

$\alpha(t_1, \ldots, t_n)$

原子式の一般形が記述できれば、これを用いてホーン節や Prolog のプログラムの一般形を記述するのは容易である。次の通り。

定義4.7 原子式（$\alpha, \beta_1, \beta_2, \ldots, \beta_n$ で表す）をコンマや文字列 :- ,文字列 ?- で区切って並べピリオドで閉じた式のうち、

① α.　　　　　　　　　または

② α :- $\beta_1, \beta_2, \ldots, \beta_n$.　または

③ ?- $\beta_1, \beta_2, \ldots, \beta_n$.　　または

④ :- $\beta_1, \beta_2, \ldots, \beta_n$.

のいずれかの形のものを**ホーン節**(Horn clause)という。そして、①～④の形のホーン節に関して、αの部分を**頭部**(head,ヘッド)，$\beta_1, \beta_2, \ldots, \beta_n$ の部分を**本体**(body,ボディ)といい、各 β_i を**ゴール**(goal)という。ホーン節はその形や役割に応じて次の様に分類される。

- **ホーン節**
  - **確定節**(definite clause)
    - **事実**(fact) -------- ①の形のホーン節
    - **規則**(rule) -------- ②の形のホーン節
  - **ゴール節**(goal clause)
    - **質問**(question) ----- ③の形のホーン節
    - **コマンド**(command) -- ④の形のホーン節

定義4.8 0個以上のホーン節を並べたものを，さらにはその途中に /*と */ で囲まれた文字列を**注釈**(comment)として含むものも（Prolog の）**プログラム**(program)という。［これから紹介する ACOS 上の Prologでは、文字 % から行の終端までの文字列も注釈になる。］

Prolog のプログラムの中で、確定節はプログラムの実行手順を記述した部分であり、ゴール節はプログラムを起動させる部分である。また、コマンドは、質問と違って詳しい実行結果の表示を伴わず、主として、プログラム読み込みの際の環境設定（例えば、新しい“演算子”の定義，使用する文字のモードの切り替え），プログラムの初期設定などに用いられる。従って、通常、確定節とコマンドだけから成るプログラムを作成／保存し、これを実行させる時になって初めて質問を与える。［コマンドの使用例としては、例えば、4.4節の例4.13を参照せよ。］

補足4.9 ここでは論理との類似性を重視して、項と原子式を厳密に区別した。しかし、Prolog プログラムの構文を定める際は、一般に、定義4.5(ii)の形の項と定義4.6(ii)の形の原子式を区別せずに、これらを合わせて**構造体**(structure;構造)と呼んでいる。そして、ここで“原子式”と呼んだものも項に分類する。

4.3 Prologプログラムの動作原理

Prolog のプログラムを動作させるためには、まず、そのプログラムの表す公理系が知識ベースとして **Prolog言語処理系**(Prolog language processor;すなわちPrologプログラムを処理／実行させるためのソフトウェア)内に構築された状態にしておく。この様な状態の下で、Prolog のプログラムは、例えば

① プログラムの表す公理系で ○○ という原子式を演繹できるか？，

② プログラムの表す公理系で、○○ という形の原子式の内どんなものを演繹できるか？，

③ 変項へのどんな割当てを行うと ○○ という列に含まれる原子式を全てプログラムの公理系から演繹できるか？

という意味の質問／ゴール節を与えることによって起動される。当然 Prolog言語処理系はこの様な質問に答えることになるが、このために実際には当該原子式を演繹する証明図を（逆方向に）試行錯誤的に構成しようとする。従って、Prolog のプログラムの実行は証明図の探索に他ならない。

以下、この証明図の探索が実際にどの様に行われるかを4．1節に挙げたプログラムで見てみよう。

例4．10(お皿の上の世界)　例4．1のプログラム

```
lemon.
apple.
tomato.
fruit :- apple.
fruit :- tomato.
```

の表す公理系が知識ベースとしてProlog言語処理系内に構築されている時、“vegetable という原子式が演繹できるか？”という意味の質問

(1) ?- vegetable.

を処理系に与えると、Prolog言語処理系はこの質問に答えるために、vegetable という原子式を演繹する証明図を（与えられたプログラムの表す公理系の下で）試行錯誤的に探し出そうとする。しかし、処理系は vegetable を導く公理や推論規則（，すなわち vegetable を頭部に持つ確定節）がないことを知り、直ちに“no”という回答を返す。

次に、“fruit という原子式が演繹できるか？”という意味の質問

(2) ?- fruit.

を与えると、Prolog言語処理系はこの質問に答えるために、fruit という原子式を演繹する証明図を試行錯誤的に探し出そうとする。具体的には、まず第１段階として、fruit を導く公理と推論規則，すなわち fruit を頭部に持つ確定節をプログラムの中から見つけ出そうとする。その結果、Prolog言語処理系は fruit を導く公理，推論規則が合わせて２個あることを知り、直ちに

fruit を導く証明図があるとすれば、その最後の部分は

$$\text{(3)}\quad \frac{\begin{matrix}\vdots\\ \text{apple}\end{matrix}}{\text{fruit}}$$

または

$$\text{(4)}\quad \frac{\begin{matrix}\vdots\\ \text{tomato}\end{matrix}}{\text{fruit}}$$

となっている筈である

と結論する。[この時点で、(2)の質問は apple または tomato が演繹できるかどうかの問題に帰着する。] そして次の第２段階では、Prolog言語処理系は、例えば(3)の可能性を探るために、apple を導く公理と演繹規則をプログラムの中から見つけ出そうとする。 その結果、処理系は apple を導く公理,推論規則が合わせて１個あることを知り、直ちに(3)を

$$\text{(5)}\quad \frac{\begin{matrix}\text{(公理)}\\ \text{apple}\end{matrix}}{\text{fruit}}$$

と書き換える。これで(2)の質問に対してfruitの演繹過程が実際に構成されたことになり、処理系は “yes” という回答を返す。質問 ?-fruit. に対する探索の様子をまとめると図４.１の様になる。[図では、処理系がその時点までに構築した証明図の候補が箱 ⊂⊃ の中に示され、その時点までに完成した証明図が箱 ⊂⊂⊃⊃ の中に示されている。 箱 ⊂⊃ の中の疑問符(?)はその前の原子式の演繹可能性がまだ調べられていないことを表し、①,②,③等の番号は処理系の思考の順序／流れを表す。]

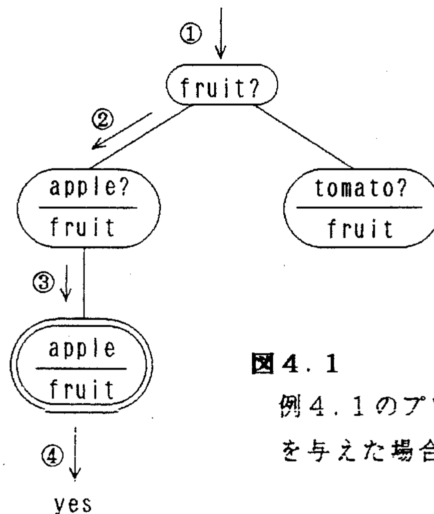


図４.１
例４.１のプログラムに質問 ?-fruit. を与えた場合の、証明図探索の様子

例4.11(ブレーメンの音楽隊;後藤[25]) 例4.2のプログラム

```
on(cock,cat).
on(cat, dog).
on(dog, donkey).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

の表す公理系が知識ベースとしてProlog言語処理系内に構築されている時、“above(dog,○○) という形の原子式の内どんなものを演繹できるか?”という意味の質問

(6) ?- above(dog,Y).

を処理系に与えると、Prolog言語処理系はこの質問に答えるために、above(dog,Y) という形の原子式を演繹する証明図を逆向きに試行錯誤で探し出そうとする。具体的には、まず第1段階として、above(dog,Y) という形の原子式を導く公理と推論規則(,すなわち変項への代入によって、頭部が above(dog,Y) という形になる確定節)をプログラムの中から見つけ出そうとする。 その結果、Prolog言語処理系は、この様な公理と推論規則が合わせて2個あることを知り、直ちに

above(dog,Y) という形の原子式を導く証明図があるとすれば、その最後の部分は

$$(7)\quad \frac{\begin{array}{c}\vdots\\ \text{on(dog,Y)}\end{array}}{\text{above(dog,Y)}}$$

または

$$(8)\quad \frac{\begin{array}{cc}\vdots & \vdots\\ \text{on(dog,Z)}, & \text{above(Z,Y)}\end{array}}{\text{above(dog,Y)}}$$

となる筈である

と結論する。 そして次の第2段階では、Prolog言語処理系は、例えば(7)の可能性を探るために、on(dog,Y) という形の原子式を導く公理や推論規則をプログラムの中から見つけ出そうとする。その結果、処理系はこの様な公理,推論規則が合わせて1個だけあることを知り、直ちに(7)を

$$(9)\quad \frac{\begin{array}{c}\text{(公理)}\\ \text{on(dog,donkey)}\end{array}}{\text{above(dog,donkey)}}$$

と書き換える。これで、(6)の質問に対して above(dog,Y) という形の原子式の証明図が実際に構成されたことになり、処理系は Y=donkey という変項への割当て方法を回答として返す。[変項へのこの割当てを行うことにより above(dog,Y) が演繹可能となることに注目せよ。]

ここで、(6)の質問者がこの回答に満足しこれを処理系に伝えた場合は、Prolog言語処理系は証明図の探索を中止し“yes”という返事を出す。しかし、質問者がこの回答に満足せずに別解を要求した場合は、処理系は証明図の探索を続行して次の第3段階へと進む。 この段階では(7)の可能性を調べ終わった所なので、次に(8)の可能性を探ることになる。式 on(dog,Z) の方を先に調べること

にすれば、処理系は on(dog,Z) という形の原子式を導く公理や推論規則をプログラムの中から見つけ出そうとする。その結果、処理系はこの様な公理,推論規則が合わせて1個だけあることを知り、直ちに(8)を

$$(10)\quad \frac{\overset{\text{(公理)}}{on(dog,donkey)},\ \overset{\vdots}{above(donkey,Y)}}{above(dog,Y)}$$

と書き換える。 第4段階では、(8)から導かれた(10)の可能性を探るために、Prolog言語処理系は above(donkey,Y) という形の原子式を導く公理や推論規則を見つけ出そうとする。その結果、処理系はこの様な公理,推論結果が合わせて2個あることを知り、直ちに

(10),すなわち(8)の形の証明図があるとすれば、それは

$$(11)\quad \frac{\overset{\text{(公理)}}{on(dog,donkey)},\ \dfrac{\overset{\vdots}{on(donkey,Y)}}{above(donkey,Y)}}{above(dog,Y)}$$

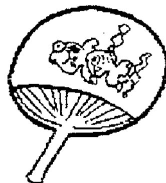
または

$$(12)\quad \frac{\overset{\text{(公理)}}{on(dog,donkey)},\ \dfrac{\overset{\vdots}{on(donkey,Z1)},\ \overset{\vdots}{above(Z1,Y)}}{above(donkey,Y)}}{above(dog,Y)}$$

となる筈である

と結論する。 第5段階では、Prolog言語処理系は、(11)の可能性を探るために on(donkey,Y) という形の原子式を導く公理や推論規則を見つけ出そうとするが、この様な公理も推論規則もないことを知り、(11)の形の証明図が不可能であると結論づける。 最後の第6段階では、Prolog言語処理系は(12)の可能性を探るために on(donkey,Z1) の方から調べる。すなわち on(donkey,Z1) という形の原子式を導く公理や推論規則を見つけ出そうとするが、第5段階と同様にこれが失敗に終わり(12)の形の証明図が不可能であると結論する。この時点で、(10),従って(8)の形の演繹過程が不可能であり、結局、第2段階で得られた変項への割当て Y=donkey を行う以外は原子式 above(dog,Y) が**演繹不能**(undeducible;すなわち、演繹可能でない)ということが分かるから、要求された別解がないという意味で処理系は "no" という回答を返す。

(6)の質問 ?-above(dog,Y) に対する証明図の探索の様子を図4.1に倣ってまとめると図4.2の様なものが得られる。



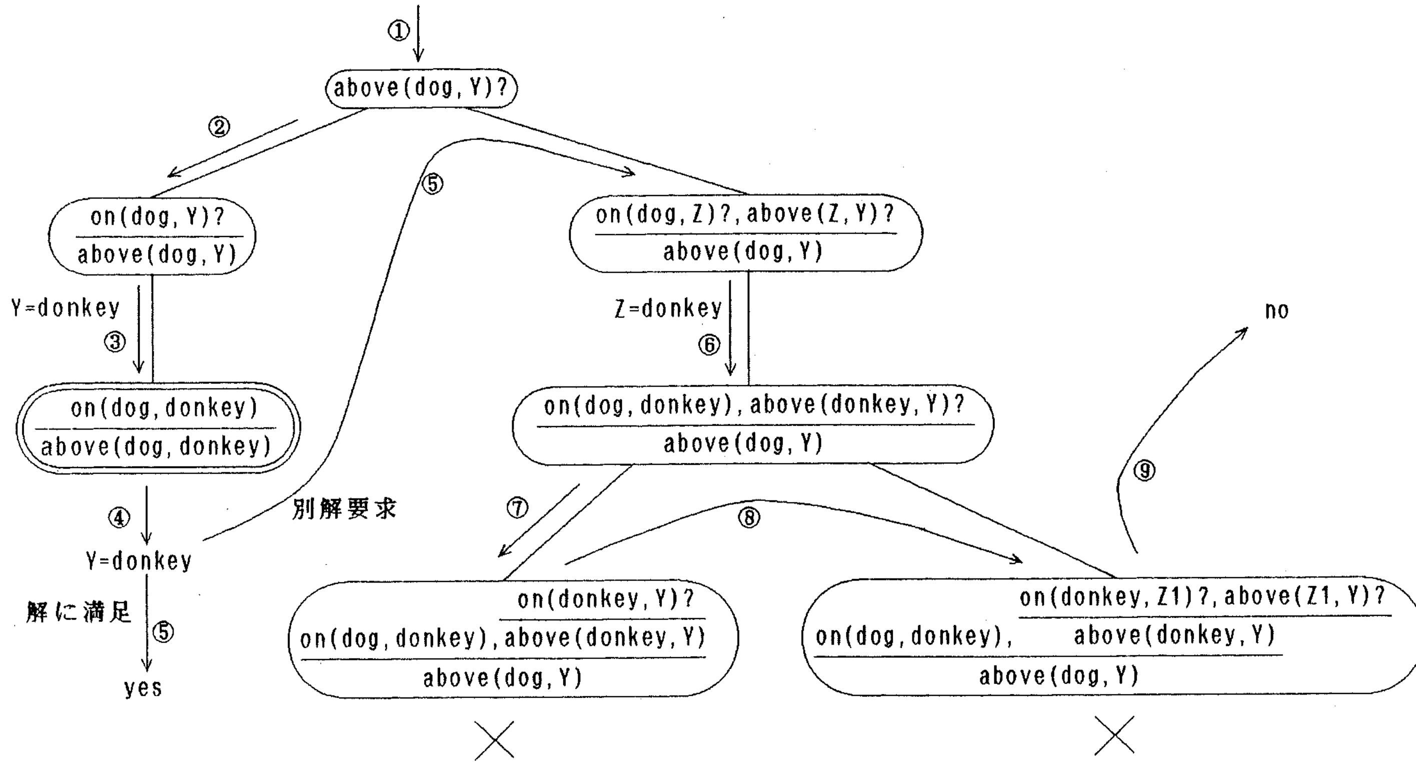


図4.2 例4.2のプログラムに質問 ?-above(dog,Y). を与えた場合の証明図探索の様子

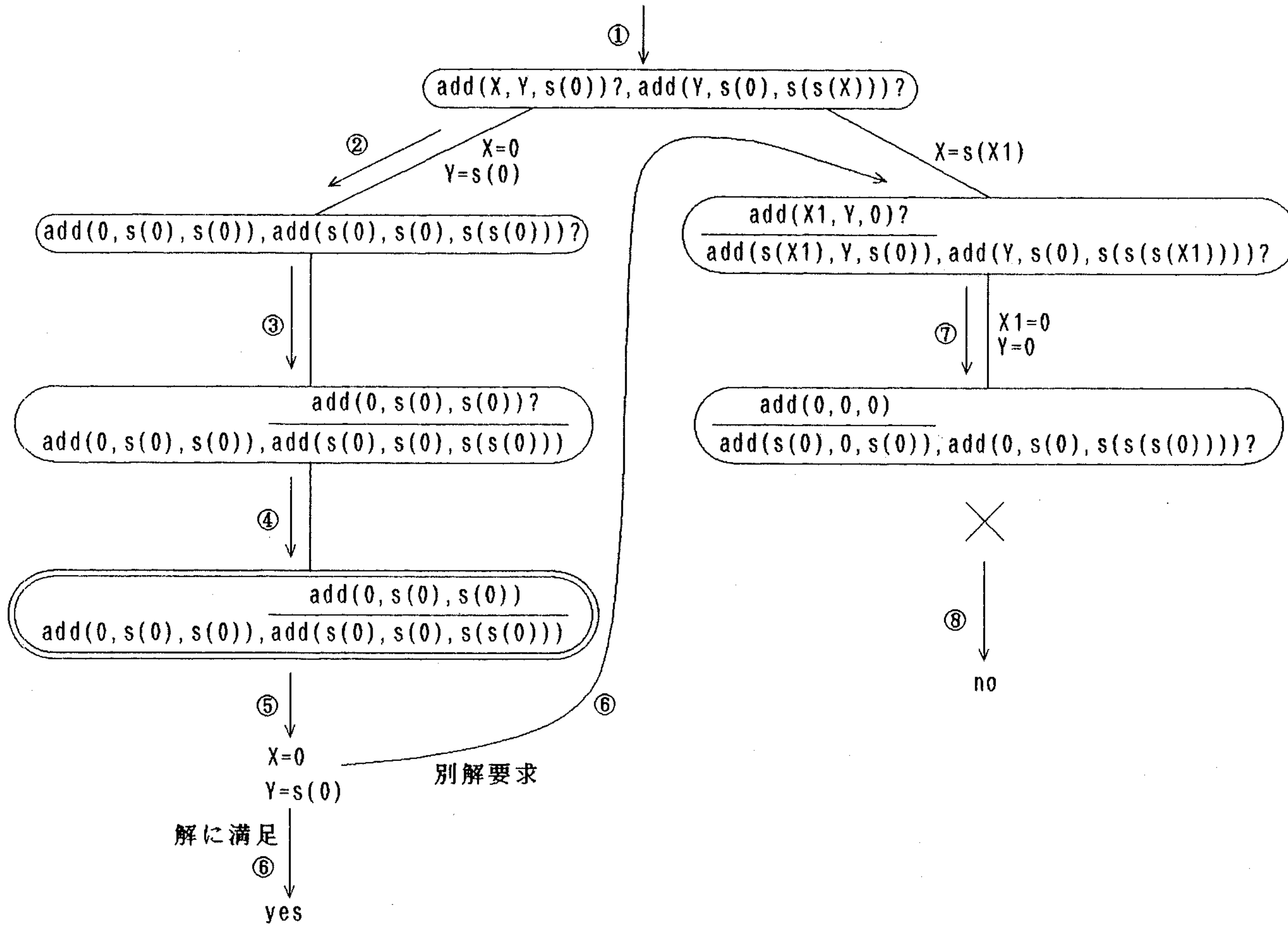


図4.3 例4.3のプログラムに質問 ?-add(X,Y,s(0)),add(Y,s(0),s(s(X))). を与えた場合の証明図探索の様子

例４．１２(ペアノの公理系の下での加算)　　例４．３のプログラム

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

の表す公理系が知識ベースとしてProlog言語処理系内に構築されている時、“どんな X,Y に対して add(X,Y,s(0)) と add(Y,s(0),s(s(X))) が同時に演繹可能となるか？”，すなわち“どんな X,Y に対して X+Y=1 と (X+2)-Y=1 が同時に満たされるか？”という意味の質問

(13)　?- add(X,Y,s(0)),add(Y,s(0),s(s(X))).

を処理系に与えると、Prolog言語処理系はこの質問に答えるために、２つの原子式 add(X,Y,s(0)) と add(Y,s(0),s(s(X))) を演繹する証明図を逆向きに試行錯誤で探し出そうとする。この探索の様子を図４．１～２に倣ってまとめると図４．３の様なものが得られる。

４．４　Prologプログラムの作成／実行

Prolog のプログラムの作成／実行は、大ざっぱに言うと、次の様な手順で行われる。

① Prolog のプログラムをエディタ(editor;すなわち文書データを作成／修正／保存するためのソフトウェア)の下で作成し、これをファイル(file;すなわち意味的なまとまりを持った情報の集まりで一つの単位として扱われるもの)として保存する。

② Prolog言語処理系を呼び出す。

③ Prolog言語処理系の中に①のプログラムを読み込む。［Prolog言語処理系の方では、読み込んだプログラムを公理系と見なして、これを基に処理系内に知識ベースを構築する。］

④ ③で読み込んだPrologプログラム／公理系／知識ベースに関して、様々な質問／ゴール節をProlog言語処理系に与える。［Prolog言語処理系の方では、与えられた質問に答えるために、証明図を逆方向に試行錯誤的に構成しようとする。この証明図探索がプログラムの実行に他ならない。］

⑤ Prolog言語処理系から脱け出す。

このうち、①のプログラム作成／保存については、使用する計算機／ソフトウェアによって様々な方法が考えられるので、ここでは特に説明しない。 以下では、②～⑤の手順が ACOS-6/MVX Ⅱ の下で実際にどの様に行われるかを、４．１節，４．３節に挙げたプログラムで見てみよう。

例４．１３(ブレーメンの音楽隊；後藤［25］)　　例４．２のプログラムに入力環境設定のためのコマンド，注釈を付け加えてファイル（bremen と名付ける）として構成すると、例えば図４．４に示されるものが得られる。［このファイル bremen の１行目のコマンド

```
:- lc.
```

は、プログラム読み込みの際に処理系を英小文字(lower-case letter)使用モードに切り替える働き

```
:- lc.                              % ファイル読み込みの際の環境設定

/*****************************************
* 例4.13　ブレーメンの音楽隊;後藤［25］ *
*****************************************/
on(cock,cat).
on(cat, dog).
on(dog, donkey).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z),above(Z,Y).
```

図4.4　ファイル bremen の中味

をする。 ACOS-6/MVX II の Prolog では英小文字,中括弧({,}),縦線(|)を扱えない端末のためにこれらの文字をそれぞれ英大文字,かぎ括弧(「,」),ドル記号($)で代用するモードが標準になっているが、これから使うパソコン端末では英小文字も中括弧({,})も縦線(|)も全て扱えるので、“英小文字使用モードへの切り替え”をこのコマンドによって行っているのである。 より詳しく言うと、lc は“英小文字使用モードへの切り替え効果”を副作用に持つ**組込み述語**(built-in predicate;すなわちProlog言語処理系の中であらかじめ定義されている述語)である。Prolog言語処理系は、ファイル bremen を読み込む際、1行目を読むと直ちにこのコマンドを,従って組込み述語 lc を実行し、それによって2行目以降に現れる英大文字を英小文字の代用として扱わなくなる。］

そして、Prolog言語処理系を呼び出し、この下でファイル bremen に入ったプログラムを実行している様子は、図4.5に示される。この図についての説明は次の通り。［図4.5では、下線部がユーザの入力を表す。］

（1行目）プロンプト(prompt;すなわちユーザからの入力の催促を表す文字列) * に続いて prolog というTSSコマンドを入力することによって、Prolog言語処理系を呼び出している。

（4行目） プロンプト

<user> ?-

に続いて consult(bremen). と入力することによって、Prolog言語処理系にファイル bremen 内のプログラムを読み込ませる。［より詳しく言うと、consult(bremen). と入力することによって、質問 ?- consult(bremen). の実行をProlog言語処理系に依頼する。 consult は指定ファイル上のプログラムを読み込むための組込み述語であるから、この質問の実行によって（副作用として）ファイル bremen 内のプログラムが処理系に読み込まれることになる。読み込みの際は、1行目を読み終えた時点で直ちにコマンド :- lc. が実行され、英小文字使用モード

```
* prolog                                          --- (1行目)
PROLOG REV 032
COPYRIGHT (C) NEC CORPORATION 1987,1990
<USER> ?-  consult(bremen).                       --- (4行目)
LOWER CASE MODE ON
bremen consulted 182 cells 0.0391200 sec.


yes

<user> ?-  above(dog,Y).                          ┐

Y = donkey  ;                                     │

no                                                │
                                                  ├ (11～22行目)
<user> ?-  above(dog,Y).                          │

Y = donkey                                        │

yes                                               ┘

<user> ?-  listing.                               --- (23行目)

above(_25,_26) :-
   on(_25,_26).
above(_25,_26) :-
   on(_25,_219),
   above(_219,_26).

on(cock,cat).
on(cat,dog).
on(dog,donkey).

yes

<user> ?-  halt.                                  --- (37行目)
```

図４．５　Prolog言語処理系との会話例［下線部がユーザの入力］

に切り替えられる。]

（１１～２２行目）　この部分は、例４．１１(6)の質問

?- above(dog,Y).

に関して実際にどの様な会話が為されるかを示している。まず、プロンプト <user> ?- に続いて above(dog,Y). と入力することによって、Prolog処理系に質問 ?- above(dog,Y). の実行を依頼している。 処理系は Y=donkey という変項への割当てを答として返すことになるが、この回答に続いてセミコロン(;)を入力すると、処理系はこれを<u>別解要求の指示</u>として受け取り直ちに“別解なし”という意味で no と回答する。 また、Y=donkey という回答に続いてセミコロンを入力せず単に送信キー（処理系によってはReturnキー）を押した場合は、処理系は“質問 ?- above(dog,Y). の実行が成功した”という意味で yes と回答する。

（２３行目）　プロンプト <user> ?- に続いて listing. と入力することによって、読み込まれたプログラムの表示をProlog言語処理系に依頼する。 listing は組込み述語であり、その実行によって（副作用として）Prolog言語処理系内に登録されている確定節が全て表示される。［２５～３３行目の表示によって、Prolog言語処理系の内部で変数名の付け替えが行われていることが分かる。］

（３７行目）　　プロンプト <user> ?- に続いて halt. と入力することによって、会話の終了をProlog言語処理系に申し出る。 halt もやはり組込み述語であり、その実行によって（副作用として）Prolog言語処理系との会話が終了する。

演習問題４

4.1 例4.2のプログラム

```
on(cock,cat).
on(cat, dog).
on(dog, donkey).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

に関して、

(1) このプログラムの表す公理系が知識ベースとしてProlog言語処理系内に構築されている時、２つの質問

(a) ?- above(dog,cock).

(b) ?- above(cat,Y).

に対してProlog言語処理系は証明図探索をどの様に行うか？ (a),(b)のそれぞれについて、探索の様子を図4.1～3に倣って図示せよ。

(2) “cat の下方にいて、かつ donkey の上方にいる動物は何か？”という意味の質問はどの様に表されるか？

4.2 例3.8で与えたものと類似した公理系

（公理） on(cock,cat),
on(cat, dog),
on(dog, donkey),

（推論規則） $\dfrac{\text{on}(x,y)}{\text{above}(x,y)}$ 但し x, y は任意,

$\dfrac{\text{on}(z,y),\ \text{above}(x,z)}{\text{above}(x,y)}$ 但し x, y, z は任意

について、

(1) この公理系はPrologのプログラムとしてどの様に表されるか？

(2) この公理系が知識ベースとしてProlog言語処理系内に構築されている時、４つの質問

(a) ?- above(X,cat).

(b) ?- above(dog,Y).

(c) ?- above(X,cat), above(cock,X).

(d) ?- above(cock,X), above(X,cat).

に対してProlog言語処理系は証明図探索をどの様に行うか？ (a)～(d)のそれぞれについ

て、探索の様子を図4．1～3に倣って図示せよ。

4．3　例4．3のプログラム

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

に関して、

(1) このプログラムで用いた自然数の表記法の下では、次の(a),(b)のための質問はそれぞれどの様に表されるか？

(a) 減算　5－2　を行う。

(b) 連立一次方程式 $\begin{cases} x+y \quad =1 \\ \quad y+z=1 \end{cases}$ の解を全て求める。

(2) このプログラムの表す公理系が知識ベースとしてProlog言語処理系内に構築されている時、問(1)の(a),(b)の質問に対してProlog言語処理系は証明図探索をどの様に行うか？　それぞれについて、探索の様子を図4．1～3に倣って図示せよ。

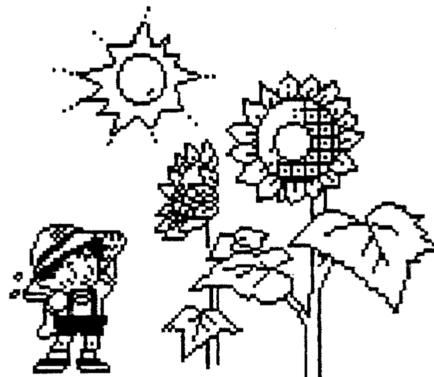
4．4　例4．4のプログラム

```
intransitive_verb(dreams).
transitive_verb(eats).
proper_noun('John').
definite_article(the).
noun(cake).
sentence(X+Y) :- noun_phrase(X), verb_phrase(Y).
noun_phrase(X) :- proper_noun(X).
noun_phrase(X+Y) :- definite_article(X), noun(Y).
verb_phrase(X) :- intransitive_verb(X).
verb_phrase(X+Y) :- transitive_verb(X), noun_phrase(Y).
```

の表す公理系が知識ベースとしてProlog言語処理系内に構築されている時、質問

```
(a) ?- sentence('John' + dreams).
(b) ?- sentence('John' + (eats + (the + cake))).
(c) ?- sentence('John' + eats).
(d) ?- sentence('John' + (eats + 'John')).
(e) ?- sentence('John' + Y).
```

を処理系に与えると、それぞれの場合にProlog言語処理系は証明図探索をどの様に行うか？
例えば(e)の質問について、探索の様子を図4．1～3に倣って図示せよ。



第5章　Ｐｒｏｌｏｇ初級プログラミング

前節４．４で挙げた lc,consult,listing,halt の他にも、Prolog言語処理系内にはあらかじめ組み込まれた述語や関数が多数存在する。そこでこの章では、**ACOS** 上の **Prolog** に備わった組込み述語,組込み関数の中でも、特に基本的なものに目を向けよう。

まず、５．１節では Prologプログラムの実行の際に基本となる手続きとその実行を引き起こす組込み述語(単一化)について述べ、続く５．２節では算術計算のための組込み関数(四則演算など),組込み述語(is,大小比較)について、５．５節では入出力のための基本的な組込み述語(read,write,nl,tab)について説明する。そして、これらの節のまとめとして、５．６節では“簡単なパズル”を解くPrologプログラムを作成する。　　また５．３～４節では、Prologプログラムの**実行追跡**(trace)／**デバッグ**(debug,虫取り;すなわち誤りを取り除くこと)に関する組込み述語(debug,nodebug,trace,notrace,spy,nospy,leash)を紹介する。

５．１　単一化

Prologプログラムの動作原理は４．３節に述べた通りであるが、プログラムを実際に実行させる際に重要になるのは“単一化”という手続きである。例えば、例４．３のPrologプログラム

(1)　add(0,X,X).

(2)　add(s(X),Y,s(Z)) :- add(X,Y,Z).

に関して質問 ?- add(X,Y,s(0)). をProlog言語処理系に行った場合、処理系は、ゴール add(X,Y,s(0)) に適用できるホーン節を探すために、各ホーン節に対して

(3) 変項へのどんな代入を行えば

　　ホーン節の頭部とゴール add(X,Y,s(0)) が同一になるかの調査

を行うが、この(3)の様に２つの項／原子式を重ね合わせて１つに合成する手続きを“単一化”と呼んでいる。［但し、同じ名前の変項でも、それを含むホーン節が異なれば、別の変項として扱う。］
ホーン節(1)の頭部 add(0,X,X) とゴール add(X,Y,s(0)) との単一化の結果、処理系は

ホーン節のX ＝ s(0)

ゴールのX　 ＝ 0

ゴールのY　 ＝ s(0)

という変項への代入の下で公理(1)がゴール add(X,Y,s(0)) に適用できると判断する。また、ホーン節(2)の頭部 add(s(X),Y,s(Z)) とゴール add(X,Y,s(0)) との単一化を行うと、処理系は

ホーン節のY ＝ ゴールのY

ホーン節のZ ＝ 0

ゴールのX　 ＝ s(ホーン節のX)

という変項への代入のもとで規則(2)もゴール add(X,Y,s(0)) に適用できると判断する。

一般に、"変項へのどんな代入を行えば２つの項／原子式が同一になるか"を調べる手続き，すなわち２つの項／原子式を重ね合わせて１つに合成する手続きを**単一化**(unification,ユニフィケーション)と言う。単一化は、Prologプログラムを実際に実行させる際に暗黙に,かつ頻繁に行われる手続きであり、Prolog言語処理系に不可欠の機能である。組込み述語の中にも単一化を引き起こすものが用意されている。次の通り。

単一化のための組込み述語 =(項または原子式,項または原子式).　原子式 =(τ_1, τ_2) は、

τ_1 と τ_2 が項／原子式として全く同一のものである

という命題を表し、通常

τ_1=τ_2

と略記される。Prologプログラムの実行中にゴール =(τ_1, τ_2) が起動されると、その演繹可能性を調べるためにProlog言語処理系は、

=(τ_1, τ_2) を満たす（，すなわち τ_1 と τ_2 が項／原子式として同一となる）様な変項への代入

を見つけ出そうとする。その結果、

① 変項への代入の捜索が成功すれば、その様な代入の下でゴール =(τ_1, τ_2) が演繹可能であることが分かる。そして、その様な代入の内で最も一般性のあるものを考え、その下で（もしあれば）新たなゴールが起動される。

② 変項への代入の捜索が失敗に終われば、ゴール =(τ_1, τ_2) は演繹不能ということになる。

例5.1(単一化)　単一化のための組込み述語 =(項または原子式,項または原子式) の使用例は図5.1に示される。[この図の中の最後の質問 ?- X=f(X). に対して処理系は"STACK OVERFLOW"というエラーメッセージを表示して処理を中止しているが、これに関して説明を行っておこう。ゴール X=f(X) に対する解は無限の長さの項を用いて X=f(f(f(f(f(...))))) と表されるが、通常のProlog言語処理系では無限長の項を扱うことはできない。単一化によって無限長の項が発生する

```
<user> ?-  parent(X,father(susie)) = parent(tom,father(X)).

no

<user> ?-  parent(X,father(susie)) = parent(tom,Y).

X = tom
Y = father(susie)

yes

<user> ?-  parent(X,Y) = parent(tom,Z).

X = tom
Y = _22
Z = _22

yes

<user> ?-  f(X,Y,Z) = f(Y,Z,tom).

X = tom
Y = tom
Z = tom

yes

<user> ?- X = f(X).

  X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
                    ・・・ （途中５０行省略） ・・・
f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(

 STACK OVERFLOW
  TERM022 E FO-MEMORY ADDRESS FAULT
      AT  335267           BA=   000000  120000    I=1040     00
                           SW=   000000  000000      000000  400000
                          SLV=   000000  120000  USL=000000  112000
```

図５.１　単一化述語の使用例

かどうかをチェック(出現チェック,occur check,という)することもできるが、この様なチェックは処理効率の低下を招くため、実用規模の処理系ではこのチェックを省くのが普通である。従って、ユーザは単一化による無限長の項の発生に十分注意しなければならない。]

補足５.２　単一化述語 = とよく似たものに組込み述語 == がある。原子式 ==(τ_1, τ_2) は、
それまでの実行過程で行われた変項への代入の下で
τ_1 と τ_2 が（その中の変項名も含めて）項／原子式として全く同一である
という命題を表し、通常 τ_1==τ_2 と略記される。従って、単一化述語 = の場合と違って、ゴール ==(τ_1, τ_2) を成功させるために変項への新たな代入は許されない。　[また、== の否定を表す組込み述語もある。すなわち、原子式 ¥==(τ_1, τ_2) は ==(τ_1, τ_2) の否定命題を表し、通常 τ_1 ¥==τ_2 と略記される。]

単一化述語の場合に限らず、一般にゴール α が起動されると、その演繹可能性を調べるために Prolog言語処理系は、ホーン節の探索を行いながら、
α が演繹されるための変項への代入
を見つけ出そうとする。その結果、

① 変項への代入の捜索が成功すれば、その様な代入の下でゴール α が演繹可能であることが分かり、ゴール α（の実行）は成功(succeed)すると言う。もちろん、新たなゴールがあれば、それは α を満たす代入の内で最も一般的なものの下で起動される。

② 変項への代入の捜索が失敗に終われば、ゴール α は演繹不能ということになり、ゴール α（の実行）は失敗(fail)すると言う。

５．２　算術計算

ACOS 上の Prolog では、変項やアトムの他に２～１０進整数,実数なども基本的な項として扱われる（定義４.５）。そして、数値を表すこれらの項に関して、次の様な組込み関数,組込み述語が処理系内に備えられている。

算術演算関数.　[DEC-10 Prolog では、実数を扱えないので、次の演算関数のうち / が整数間の除算を意味し、// は定義されていない。また、abs も組込み関数ではない。]

単項演算
- \+ ・・・ 恒等関数。 +(X) は +X と略記できる。
- \- ・・・ 符号反転。 -(X) は -X と略記できる。
- abs ・・・ 絶対値。

２項演算
- + ・・・ 加算。 +(X,Y) は X+Y と略記できる。
- - ・・・ 減算。 -(X,Y) は X-Y と略記できる。
- * ・・・ 乗算。 *(X,Y) は X*Y と略記できる。
- / ・・・ 実数間の除算（結果は実数）。 /(X,Y) は X/Y と略記できる。
- // ・・・ 整数間の除算（結果は整数）。 //(X,Y) は X//Y と略記できる。
- mod ・・・ mod(X,Y) は X を Y で割ったときの余りを表し、その符号は X と同じ。 mod(X,Y) は X mod Y と略記できる。

論理演算関数.　［DEC-10 Prolog では、次の演算関数のうち @ は定義されておらず、円記号(¥)の代わりにバックスラッシュ( )が用いられる。］

単項演算 ¥ ・・・ ビット毎の否定（反転）。 ¥(X) は ¥X と略記できる。

２項演算
- /¥ ・・・ ビット毎の論理積。 /¥(X,Y) は X/¥Y と略記できる。
- ¥/ ・・・ ビット毎の論理和。 ¥/(X,Y) は X¥/Y と略記できる。
- @ ・・・ ビット毎の排他的論理和。 @(X,Y) は X@Y と略記できる。
- << ・・・ <<(X,Y) は、X の内容を Y ビットだけ左にシフトし（すなわち、ずらし）て得られる結果を表し、X<<Y と略記できる。
- >> ・・・ >>(X,Y) は、X の内容を Y ビットだけ右にシフトし（すなわち、ずらし）て得られる結果を表し、X>>Y と略記できる。

型変換関数.　［DEC-10 Prolog では、実数を扱えないので、次の演算関数はどれも定義されていない。］

単項関数
- float ・・・ 実数化。
- floor ・・・ 整数化（切り捨て）。すなわち floor(X)＝X以下の最大整数 である。
- ceiling ・・・ 整数化（切り上げ）。すなわち ceiling(X)＝X以上の最小整数 である。
- truncate ・・・ 整数化（零に向けての切り捨て）。すなわち X≧0 の時 truncate(X)＝floor(X), X＜0 の時 truncate(X)＝ceiling(X) である。
- round ・・・ 整数化（四捨五入）。すなわち round(X)＝floor(X+0.5) である。

略記法.　算術式においては、式の曖昧さを避けるために括弧を用い、通常通り括弧の省略も行う。括弧が省略された場合は、結合の優先順位は次の順に高いものとする。［但し、同じグループ内では、式の左側に現れる方が結合が強いものとする。］

① mod

② *, /, //, <<, >>

③ +（単項）, -（単項）, ¥

④ +（２項）, -（２項）, /¥, ¥/, @

算術式の評価のための組込み述語 is(項,項).　原子式 is(τ, ε) は、

　　算術式εを評価して得られる数値がτと等しい

という命題を表し、通常

　　τ is ε

と略記される。Prologプログラムの実行中にゴール is(τ, ε) が起動されると、その演繹可能性を調べるためにProlog言語処理系は、まず項 ε を算術式として評価し、続いてその評価値と項 τ との単一化を試みる。その結果、単一化が成功すればゴール is(τ, ε) も成功し、単一化が失敗に終わればゴール is(τ, ε) も失敗する。より詳しく言うと、

① 項 ε を算術式として評価できない場合，例えば ε が値の割り当てられていない変項を含む場合には、エラー(error)となる。

② 項 ε を算術式として評価でき,その評価値が τ と単一化できる場合，例えば τ が変項で ε が算術式として評価できる場合には、ゴール is(τ, ε) は成功し演繹可能と結論づけられる。

③ 項 ε を算術式として評価できるが,その評価値が τ と単一化できない場合，例えば τ が構造体で ε が算術式として評価できる場合には、ゴール is(τ, ε) は失敗し演繹不能と結論づけられる。

算術比較のための組込み述語.　算術式の値の大小比較のためには次の６つの組込み述語が用意されている。

- =:=$(\varepsilon_1, \varepsilon_2)$ ・・・ “ε_1 の評価値が ε_2 の評価値と等しい”という命題を表す。
- =¥=$(\varepsilon_1, \varepsilon_2)$ ・・・ “ε_1 の評価値が ε_2 の評価値と等しくない”という命題を表す。
- $<(\varepsilon_1, \varepsilon_2)$ ・・・ “ε_1 の評価値が ε_2 の評価値より小さい”という命題を表す。
- $=<(\varepsilon_1, \varepsilon_2)$ ・・・ “ε_1 の評価値が ε_2 の評価値以下である”という命題を表す。
- $>(\varepsilon_1, \varepsilon_2)$ ・・・ “ε_1 の評価値が ε_2 の評価値より大きい”という命題を表す。
- $>=(\varepsilon_1, \varepsilon_2)$ ・・・ “ε_1 の評価値が ε_2 の評価値以上である”という命題を表す。

これらの原子式は、それぞれ、通常 ε_1=:=ε_2, ε_1=¥=ε_2, $\varepsilon_1<\varepsilon_2$, $\varepsilon_1=<\varepsilon_2$, $\varepsilon_1>\varepsilon_2$, $\varepsilon_1>=\varepsilon_2$ と略記される。 Prologプログラムの実行中に例えばゴール =:=$(\varepsilon_1, \varepsilon_2)$ が起動されると、その演繹可能性を調べるためにProlog言語処理系は、まず項 ε_1, ε_2 を算術式として評価する。そして、それらの値が等しいかどうかによって、ゴール =:=$(\varepsilon_1, \varepsilon_2)$ の成功／失敗を判定する。もちろん、項 ε_1, ε_2 を算術式として評価できない場合はエラーとなる。

算術計算のための組込み関数,組込み述語を用いれば、様々の処理／計算手順をPrologプログラムで表すことができる。例えば次の通り。

例５．３(人口密度の計算)　Prologは広義のデータベースの機能，すなわちデータを組織的に蓄積／処理する機能を備えているので、これと算術演算の機能を組み合わせれば、

任意に与えられた国の人口密度を計算するプログラム

をPrologで記述するのは容易である。例えば、図５．２の様なPrologプログラムを構成して、これを図５．３の様に実行することができる。［図５．３の実行例からも分かる様に、このプログラムは、データベースが不完全なために、限られた国々に対してしか人口密度を計算できない。］

```
:- lc.                                   % ファイル読み込みの際の環境設定

/*************************
* 例５．３　人口密度の計算 *
*************************/
population('China',      113906).    % 単位は 万人，１９９０年，
population('India',       82706).    % 上位８か国，
population('U.S.A.',      24998).    % データは文献［57］による
population('Indonesia',   17930).
population('Brazil',      15037).
population('Russia',      14804).
population('Japan',       12354).
population('Bangladesh',  11559).

area('Russia',    17075).            % 単位は 1000平方キロ，１９８９年，
area('Canada',     9976).            % 上位７か国と日本，
area('China',      9597).            % データは文献［57］による
area('U.S.A.',     9373).
area('Brazil',     8512).
area('Australia',  7713).
area('India',      3288).
area('Japan',       378).

density(X, Y) :-
     population(X, P),
     area(X, A),
     Y is float(P)/float(A).
```

図５．２　人口密度計算のプログラム

```
<user> ?-  density('Japan',X).

X = 32.6825

yes

<user> ?-  density('Canada',X).

no

<user> ?-  density(X,Y).

X = China
Y = 11.8689  ;

X = India
Y = 25.1539  ;

X = U.S.A.
Y = 2.66702  ;

X = Brazil
Y = 1.76656  ;

X = Russia
Y = 0.866999  ;

X = Japan
Y = 32.6825  ;

no
```

図5．3　“人口密度計算のプログラム”の実行例

例5．4(階乗の計算)　　Prologでは**再帰法**(recursion;すなわちある事柄を定義する時に、今定義しようとしている“事柄”を用いて定義する方法)によって述語を定義できるので、これと算術演算の機能を組み合わせれば、

任意に与えられた非負整数の階乗を計算するプログラム

をPrologで記述するのは容易である。例えば、漸化式(recurrence formula)

$$f(x) = \begin{cases} 1 & (x=0 \text{ の場合}) \\ x \times f(x-1) & (x>0 \text{ の場合}) \end{cases}$$

によって階乗関数 $f(x)=x!$ が定義されることに着目すれば、階乗についての公理系

(公理)　$1=f(0)$,

(規則)　$\dfrac{x>0,\ y_1=f(x-1),\ y=x\times y_1}{y=f(x)}$

が得られる。これをそのままPrologプログラムに書き直せば図5.4の様になり、このプログラムを実行させると例えば図5.5の様になる。

図5.4のプログラムは分かり易いが、その実行時は再帰の進行に伴って"Y is X*Y1"という形の未処理の（すなわち演繹可能性をまだ調べていない）ゴールの個数が増えるので、大きな記憶領域を必要とすることがある。これとは逆に、多少分かりにくくなるが記憶領域のあまり要らないプログラムを作ることもできる。例えば、漸化式

$$g(z,x)=\begin{cases} z & (x=0\ \text{の場合}) \\ g(z\times x,\ x-1) & (x>0\ \text{の場合}) \end{cases}$$

によって関数 $g(z,x)=z\times x!$ が定義されることに着目して公理系

```
:- lc.                                  % ファイル読み込みの際の環境設定

/*****************************************
* 例5.4(その1)　再帰法による階乗の計算 *
*****************************************/
factorial1(0, 1).

factorial1(X, Y) :-
     X > 0,
     X1 is X-1,
     factorial1(X1, Y1),
     Y is X*Y1.
```

図5.4　再帰法で階乗を計算するプログラム

```
<user> ?-  factorial1(4,X).

X = 24

yes
```

図5.5　"再帰法による階乗計算のプログラム"の実行例

（公理）　$z=g(0,z)$,

（規則）　$$\frac{x>0,\ y=g(x-1,z\times x)}{y=g(x,z)},\qquad \frac{y=g(x,1)}{y=x!}$$

を導き、これをそのままPrologプログラムとして書き直せば図5.6の様になる。実行例は図5.7の通りである。 このプログラムは、図5.4のものと異なり、実行時は再帰の進行に伴って未処理のゴールを次々に発生させることがなく、それゆえ記憶領域を効率的に使用する。この様な再帰法を**末端再帰法**(tail recursion,**終端再帰法**,**限嗣再帰法**)という。

```
:- lc.                              % ファイル読み込みの際の環境設定

/**********************************************
* 例5.4(その2)　末端再帰法による階乗の計算 *
**********************************************/
factorial2(X, Y) :-
     fact(X, 1, Y).

fact(0, Z, Z).
fact(X, Z, Y) :-
     X > 0,
     X1 is X-1,
     Z1 is Z*X,
     fact(X1, Z1, Y).
```

図5.6　末端再帰法で階乗を計算するプログラム

```
<user> ?-  factorial2(4,X).

X = 24

yes
```

図5.7　“末端再帰法による階乗計算のプログラム”の実行例

例5.5(ユークリッドの互除法)　Prologでは再帰法によって述語を定義できるので、これと算術演算の機能を組み合わせれば、

任意に与えられた2つの非負整数の最大公約数(greatest common divisor)
を計算するプログラム

を容易に作成することができる。例えば、図5.8のPrologプログラムを構成して、これを図5.9の様に実行することができる。[このプログラムは、基本的には、

x, y が正整数で $x > y$ なら

xとyの最大公約数 ＝ ($x-y$)とyの最大公約数,

従って、これを連続して floor(x/y) 回適用して

xとyの最大公約数 ＝ ($x-y$) とyの最大公約数
＝ ($x-2y$)とyの最大公約数
＝ ($x-3y$)とyの最大公約数　　floor(x/y)行
⋮
＝ mod(x, y)とyの最大公約数
＝ yとmod(x, y)の最大公約数

という事実に基づく**ユークリッドのアルゴリズム**(algorithm,計算手順)をPrologプログラムとして表したものである。]

```
:- lc.                              % ファイル読み込みの際の環境設定

/****************************************************
* 例5.5　ユークリッドの互除法による最大公約数の計算 *
****************************************************/
gcd(0, Y, Y).
gcd(X, 0, X).

gcd(X, Y, Z) :-
     X > 0,
     Y > 0,
     Remainder is X mod Y,
     gcd(Y, Remainder, Z).
```

図5.8　ユークリッドの互除法で最大公約数を計算するプログラム

```
<user> ?- gcd(144,270,Z).

Z = 18

yes
```

図５．９ “最大公約数計算のプログラム”の実行例

５．３ プログラムの手続き的解釈と箱モデル

Prologプログラムは、その中のホーン節の並ぶ順序や各規則の本体内(i.e.仮定部)のゴールの並ぶ順序に依存して、異なる実行結果をもたらし得る。例えば、１つのホーン節だけから成るプログラム

```
above(X,Y) :- on(X,Z), above(Z,Y).
```

に質問 ?- above(cat,Y). を与えると即座に no と返ってくるが、本体部のゴールを並べ替えて得られるプログラム

```
above(X,Y) :- above(Z,Y), on(X,Z).
```

に質問 ?- above(cat,Y). を与えても図５．１０に示される無限探索を行うだけで何の返答もない。

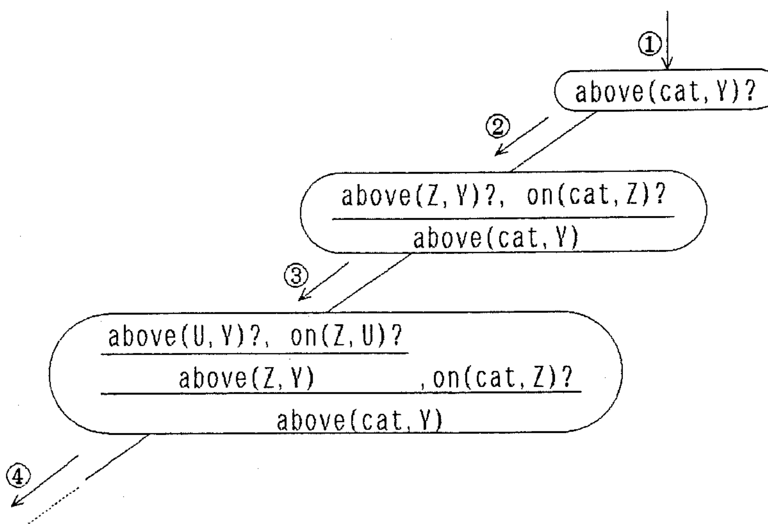


図５．１０ 無限探索の様子

従って、プログラムの実行過程に注目する際には、4章の様にPrologプログラムを単に公理系と見るだけでは不十分であり、プログラムの実行の様子を反映した計算モデルが必要である。この様なモデルとしては、4.3節の図4.1〜3に例示された探索図を挙げることもできるが、より一般的には、Prolog言語処理系の立場に立って、

Prologプログラムを処理手続きの集まりと見る考え方,

いわゆる**手続き的解釈**(procedural interpretation)が利用されている。また、手続き的解釈に基づいてプログラムの実行過程の**追跡**(trace)をするためには、バード(L.Byrd)の“箱モデル”が有名である。

Prologプログラムを手続き的に解釈する立場では、同一の述語名を頭部に持つホーン節の並び／集まりを1つの処理手続きと見る。

例5.6(手続き的解釈)　例4.2,例4.11で考えた(ブレーメンの音楽隊の)プログラム

```
on(cock,cat).
on(cat,dog).
on(dog,donkey).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

の中には2つの処理手続きが含まれる。この内の1つは、述語名 on を頭部に持つホーン節の並び

```
on(cock,cat).
on(cat,dog).
on(dog,donkey).
```

で表され、

on(△,□) の実行(i.e.演繹可能性の判定)には

① on(△,□) と on(cock,cat) の照合／単一化,

② on(△,□) と on(cat,dog) の照合／単一化,

③ on(△,□) と on(dog,donkey) の照合／単一化

を順に行い、単一化可能と判定された時点で即座に on(△,□) の実行を成功で終える。また、単一化が全て失敗すれば、on(△,□) の実行を失敗で終える、

という処理手続きを意味する。また、もう1つの処理手続きは、述語名 above を頭部に持つホーン節の並び

```
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

で表され、

above(△,□) の実行（i.e.演繹可能性の判定）には、まず

　① on(△,□) を実行し、それが成功に終われば above(△,□) の実行も成功で終える。

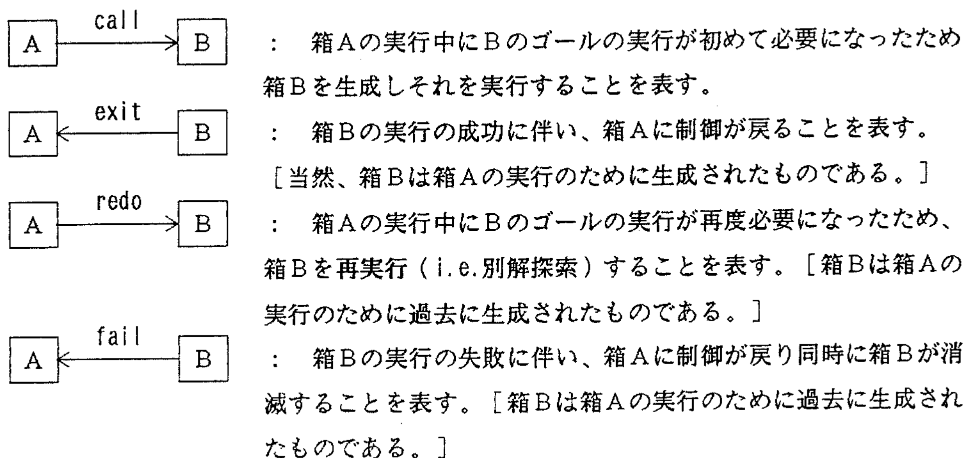
そして、これで成功しなければ、次に

　② on(△,Z) を実行し、それが成功する様な Z について above(Z,□) を実行する。ここでも成功すれば above(△,□) の実行を成功で終える。

また、①でも②でも成功しなければ、above(△,□) の実行は失敗で終える、

という意味を持つ。

箱モデル(box model)は、Prologプログラムの実行の流れを手続き的解釈に基づいて視覚化するための計算モデルであり、プログラムの実行過程を見易い形で実際に図示するために変形して用いられることもある。例えば新田&佐藤［28］の変形版では、ゴールの実行（i.e.演繹可能性の判定）のための処理をそれぞれ１つの箱で表し、制御の流れを箱の間の矢印で表す。そして、各々の矢印をその意味によって次の様にラベル付けする。



： 箱Ａの実行中にＢのゴールの実行が初めて必要になったため箱Ｂを生成しそれを実行することを表す。

： 箱Ｂの実行の成功に伴い、箱Ａに制御が戻ることを表す。［当然、箱Ｂは箱Ａの実行のために生成されたものである。］

： 箱Ａの実行中にＢのゴールの実行が再度必要になったため、箱Ｂを**再実行**（i.e.別解探索）することを表す。［箱Ｂは箱Ａの実行のために過去に生成されたものである。］

： 箱Ｂの実行の失敗に伴い、箱Ａに制御が戻り同時に箱Ｂが消滅することを表す。［箱Ｂは箱Ａの実行のために過去に生成されたものである。］

以下、この変形箱モデルを用いてプログラムの実行過程を図示してみよう。

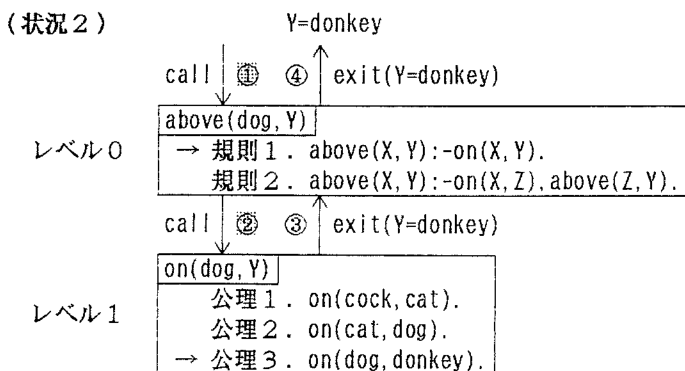
例５．７(変形箱モデルを用いた､プログラム実行過程の追跡)　　例４．２,例４．１１,例５．６で考えた（ブレーメンの音楽隊の）プログラム

```
on(cock,cat).
on(cat,dog).
on(dog,donkey).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

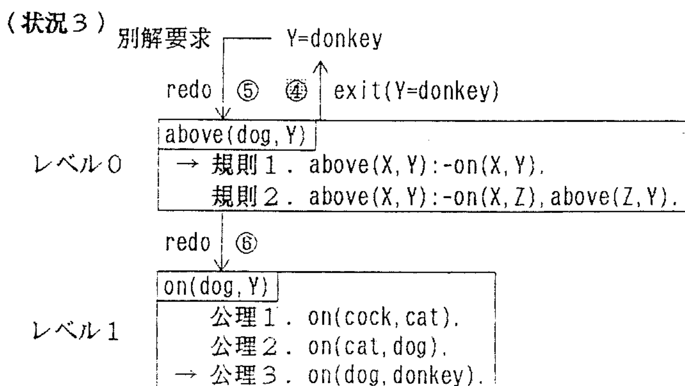
に質問 ?- above(dog,Y). を与えた時の実行の様子は、4.3節の図4.2にも示されているが、変形箱モデルを用いて図示すると次の様になる。[ここで、①,②,③, ... は制御の流れの順序を表す。また、分かり易さのために、それぞれの箱の中には (i) そこで実行されるゴール, (ii) そのゴールの実行のための処理手続き,および (iii) 処理手続き内で現在実行中のホーン節を指す矢印を入れておく。]

(状況1)

(状況2)

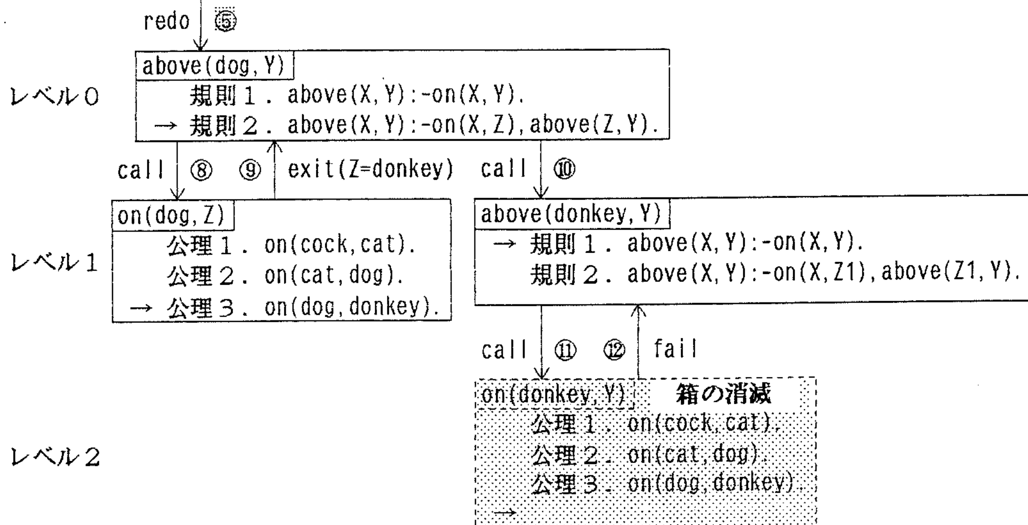


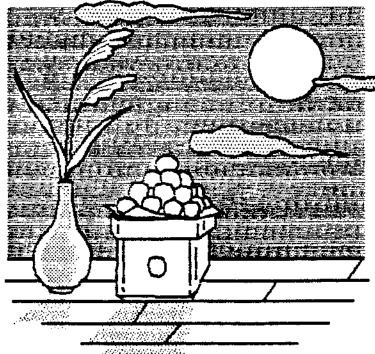
(状況3)



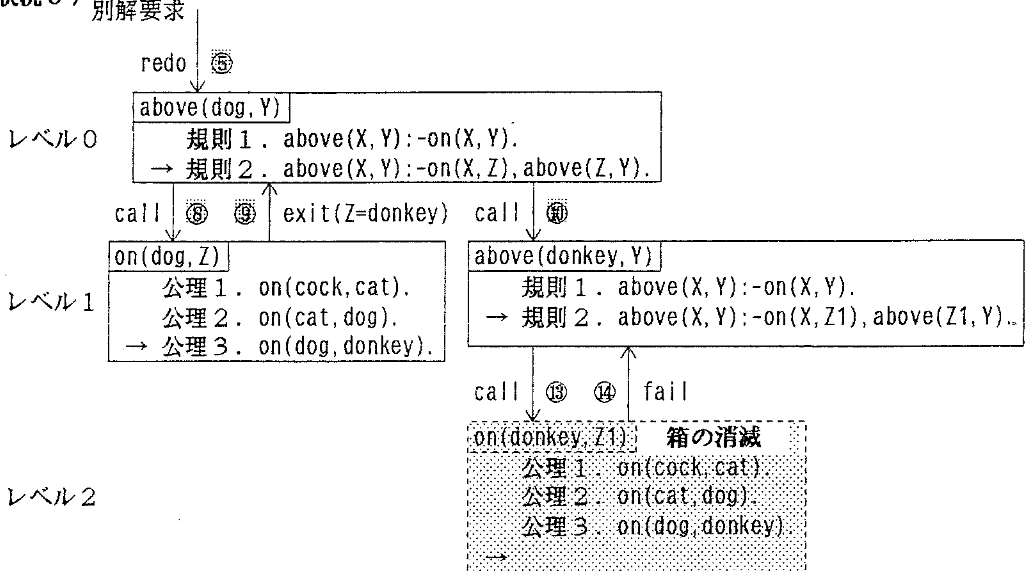
（状況４）

（状況５）

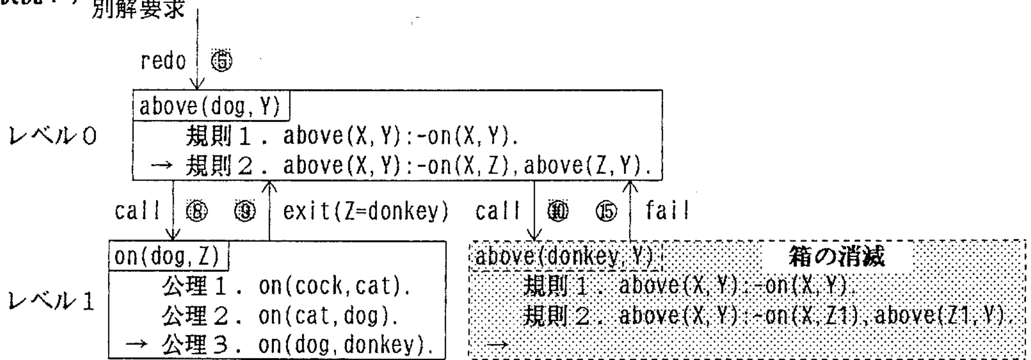




（状況6）

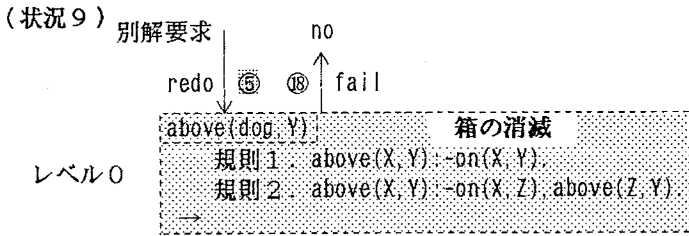


（状況7）



（状況8）

（状況９）



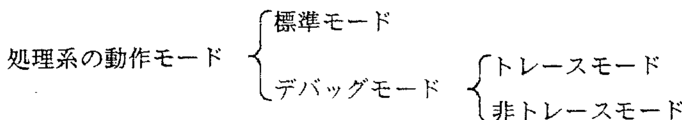
５．４　デバッガ

ACOS上のProlog言語処理系では、プログラムの実行過程を箱モデルに基づいて追跡して途中の出来事を表示するための様々な道具が備わっていて、プログラマはこれらを**デバッガ**(debugger;すなわちプログラムの誤り,bug,を発見／除去するための道具)として利用できる様になっている。実行過程の追跡･表示は、具体的には、箱モデルにおいて

① どの箱が生成され呼び出(call)されたか？，
② どの箱が再実行(redo)されたか？，
③ どの箱の実行がどういう風に成功して制御が箱の外に出(exit)たか？，
④ どの箱が失敗(fail)によって消滅したか？，

を起こった順に表示させることにより行うが、効果的な追跡･表示のために一部の箱にだけ注目してそれらについての表示だけを行わせることもできる。例えば、ある述語名の処理手続きに注目してこれらの箱（**スパイ点**,spy point,という）に関する出来事だけを表示させたり、デバッガとの会話によって上の①～④の出来事の表示を中断／再開させたりもできる。

より詳しく言うと、ACOS-Prolog言語処理系の動作モードは、プログラムの実行過程の追跡（･表示）を行うかどうかによって、**デバッグモード**(debug mode),**標準モード**(normal mode) の２つに分類できる。この内デバッグモードでは、プログラムの実行速度やメモリの使用効率が低下するが、スパイ点への制御の流れ(call,redo),スパイ点からの制御の流れ(exit,fail)についての表示は必ず行われる。スパイ点以外の箱に関する出来事を表示するかどうかはデバッガとの会話などにより決まるが、初期設定として最も下位レベルの箱（すなわち質問の本体中のゴールに相当する箱で、例５.１３の図では“レベル０”という表示の右の箱）についての表示をするかどうかでデバッグモードはさらに **トレースモード**(trace mode),**非トレースモード**(notrace mode) という２つに分かれる。従って、ACOS-Prolog言語処理系の動作モードは、結局、次の様に分類される。

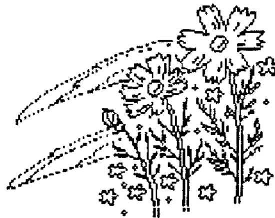


トレースモードではスパイ点に加えて“最も下位レベルの箱”についての出来事も表示されるが、非トレースモードでは“最も下位レベルの箱”についての出来事は表示されない。また、スパイ点でも“最も下位レベルの箱”でもない箱については、トレースモードにおいても非トレースモードにおいてもデバッガとの会話によって決まるが、特に指示がなければすぐ下のレベルと同じ扱いになる。従って、スパイ点を指定せずにトレースモードに入り以後何の指示も与えなければ、全ての箱についての出来事が表示されることになる。

例5.8(トレースモードの下での徹底した追跡・表示) 例5.6～7で考えた（ブレーメンの音楽隊の）プログラム

```
on(cock,cat).
on(cat,dog).
on(dog,donkey).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

に質問 ?- above(dog,Y). を与えた時の実行の様子は、変形箱モデルを用いて例5.7に図示されているが、ACOS-Prologのデバッガを用いて出来事の徹底した(exhaustive)表示をさせると図5.11の様になる。［ここで、組込み述語の trace は処理系の動作モードをトレースモードに切り替える働きをする。箱についての出来事(call,redo,exit,fail)の表示の右にある文字 ? はユーザからの指示を催促するプロンプトであり、その右の文字 c は“すぐ上のレベルの箱についても表示を続ける(continue)”ことを処理系に指示している。また、図の左側に付け加えられた番号 ①,②,③, ... は例5.7で用いた番号に対応しており、その横の [0],[1] または [2] はレベル数を表す。図5.11を見て理解できない場合は、例5.7で示した図と見比べると良い。］



```
    <user> ?-  trace.
    Enter debug mode.
    Enter trace mode.

    yes

    <user> ?-  above(dog,Y).
①  [0]   Call:  above(dog,_21)  ?   c
②  [1]   Call:  on(dog,_21)  ?   c
③  [1]   Exit:  on(dog,donkey)
④  [0]   Exit:  above(dog,donkey)

    Y = donkey  ;
⑤  [0]   Redo:  above(dog,donkey)  ?   c
⑥  [1]   Redo:  on(dog,donkey)  ?   c
⑦  [1]   Fail:  on(dog,_21)
⑧  [1]   Call:  on(dog,_238)  ?   c
⑨  [1]   Exit:  on(dog,donkey)
⑩  [1]   Call:  above(donkey,_21)  ?   c
⑪  [2]   Call:  on(donkey,_21)  ?   c
⑫  [2]   Fail:  on(donkey,_21)
⑬  [2]   Call:  on(donkey,_299)  ?   c
⑭  [2]   Fail:  on(donkey,_299)
⑮  [1]   Fail:  above(donkey,_21)
⑯  [1]   Redo:  on(dog,donkey)  ?   c
⑰  [1]   Fail:  on(dog,_238)
⑱  [0]   Fail:  above(dog,_21)

    no
```

図５．１１　プログラム実行の全面追跡・表示の例［下線部がユーザの入力］

例5.9(スパイ点を利用した追跡・表示)　例5.6～8で考えた（ブレーメンの音楽隊の）プログラム

```
on(cock,cat).
on(cat,dog).
on(dog,donkey).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

に質問 ?- above(dog,Y). を与えたときの実行の様子は、例えば above という述語の処理の箱をスパイ点に設定してこれらの表示だけを行えば、図5.12の様になる。　［ここで、組込み述語の spy は、引数部に指定された述語の処理の箱をスパイ点とし、同時に、処理系の動作モードが標準モードの時これを非トレースのデバッグモードに切り替える働きをする。この組込み述語の使い方が、spy(述語名) ではなく、spy 述語名 または spy [述語名, ... ,述語名] であることに注意しよう。また、レベル数の表示の右の文字 * は、この行がスパイ点に関する情報表示であることを表す。プロンプト ? の右の文字 l は“次のスパイ点まで一挙に跳ん(leap)で、それまで制御の流れの情報表示を一時中断する”ということを処理系に指示している。］

```
<user> ?-  spy above.
Enter debug mode.
Spy point is set on above/2

yes

<user> ?-  above(dog,Y).
[0] * Call:  above(dog,_21)  ?  l
[0] * Exit:  above(dog,donkey)

Y = donkey  ;
[0] * Redo:  above(dog,donkey)  ?  l
[1] * Call:  above(donkey,_21)  ?  l
[1] * Fail:  above(donkey,_21)
[0] * Fail:  above(dog,_21)

no
```

図5.12　スパイ点を利用して追跡・表示する例［下線部がユーザの入力］

最後に、デバッガに関する組込み述語,プロンプト ? の次に入力する文字（すなわち制御の流れの情報表示の中断／再開を制御するコマンド）についてまとめておこう。まず、組込み述語については次の通り。

debug. 最初の１度だけ成功する０引数述語であり、処理系が標準の動作モードの時にこれを非トレースのデバッグモードに切り替える働きをする。

nodebug. 最初の１度だけ成功する０引数述語であり、処理系の動作モードを標準モードに切り替える働きをする。

trace. 最初の１度だけ成功する０引数述語であり、処理系の動作モードをトレースモードに切り替える働きをする。

notrace. 最初の１度だけ成功する０引数述語であり、処理系の動作モードがトレースモードの時にこれを非トレースのデバッグモードに切り替える働きをする。

spy 述語名 または spy [述語名, ... ,述語名]. 最初の１度だけ成功する１引数述語であり、引数部に指定した述語の処理の箱をスパイ点とする働きをする。同時に、処理系が標準の動作モードの時にこれを非トレースのデバッグモードに切り替える。

nospy 述語名 または nospy [述語名, ... ,述語名]. 最初の１度だけ成功する１引数述語であり、引数部に指定した述語の処理の箱をスパイ点から外す働きをする。

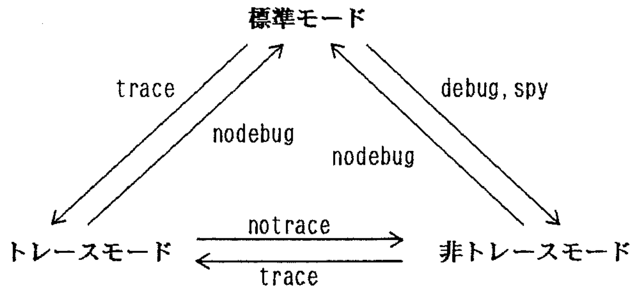
debugging. 最初の１度だけ成功する０引数述語であり、その時のデバッグ環境を表示する働きをする。

leash( ). 最初の１度だけ成功する１引数述語であり、引数で指定した出来事の直後にプロンプト ? を出す様に処理系に指示する働きを持つ。 引数としては full,tight, half,loose,off,0,1,2, ... ,15 のいずれかが可能であり、その意味は次の通りである。

full	……	call,redo,fail,exitの流れが表示された直後。	
tight	……	call,redo,fail	の流れが表示された直後。
half(標準)	…	call,redo	の流れが表示された直後。
loose	……	call	の流れが表示された直後。
off	……	（プロンプト ? を出さない。）	
整数	……	引数を２進数として表して 2'cerf となったとすると、	

ｃが１ならcallの表示の直後に，ｅが１ならexitの表示の直後に，
ｒが１ならredoの表示の直後に，ｆが１ならfailの表示の直後に
プロンプト ? を出すことを意味する。

従って、処理系の動作モードの切り替えは次の図の様に行う。



また、プロンプト ? の次に入力する文字（すなわち制御の流れの情報表示の中断／再開を制御するコマンド）としては、次のものが有用である。［他に、e,[,+ または f を入力することができるが、これらの説明はここでは省略する。］

c または 入力コマンド省略 … “すぐ上のレベルの箱についても表示を続け(continue)ながら、１ステップずつ実行を進める(creep)”ということを処理系に指示する。

s … “より上位の箱については表示を飛ばす(skip,抜かす)”ということを処理系に指示する。但し、より上位の箱にスパイ点がありそこでプロンプト ? が表示される場合は、そこから先のレベルではそのプロンプトに続く制御コマンドの指示が優先される。

l … “次のスパイ点まで一挙に跳ん(leap)で、それまで制御の流れの情報表示を一時中断する”ということを処理系に指示する。

n … “組込み述語 nodebug を実行して処理系を標準の動作モードに切り替えることによって、プログラム実行の追跡･表示を中止する”ということを処理系に指示する。

a … “プログラムの実行を中止(abort)させる”ことを処理系に指示する。

h または ? … 処理系に助け(help)を求める。すなわち、入力コマンドとして可能なものとそれらの簡単な説明を処理系に要求する。

5．5　入出力のための基本述語

他の多くのプログラミング言語と同様に、Prologは端末（キーボード）やファイルからデータを入力したり、端末（ディスプレイ）やファイルにデータを出力したりする機能を備えている。複数の入力源,出力先を同時に扱うこともできるが、各時点で“暗黙の入力源”，“暗黙の出力先”をそれぞれ1個ずつ固定して、その時点の“暗黙の入力源”（から流れ出るデータの流れ）を**カレント入力ストリーム**(current input stream)，その時点の“暗黙の出力先”（へ流れ込むデータの流れ）を**カレント出力ストリーム**(current output stream)と呼ぶ。［1個のファイルを入力と出力のために同時に用いることはできない。念のため。］

特にProlog言語処理系との会話の開始直後はカレント入力ストリームは端末のキーボード,カレント出力ストリームは端末のディスプレイに設定されており、それゆえ、新たに再設定／指定しなければ入出力は端末を介して行われることになる。

それでは、入出力のための組込み述語のうち基本的なものを紹介しよう。次の通り。

項の入力のための組込み述語 read(項).　原子式 read(τ) は、

　　　カレント入力ストリーム上の次の項がτと等しい

という命題を表す。従って、Prologプログラムの実行中にゴール read(τ) が起動されると、その演繹可能性を調べるために、Prolog言語処理系はカレント入力ストリームから次の項を読み込み、これと τ との単一化を試みる。その結果、単一化が成功すれば read(τ) も成功し、単一化が失敗に終わればゴール read(τ) も失敗する。［ここで、項の入力の際は“カレント入力ストリーム上の項は、ピリオド（と改行コードの列）によって区切られている”と仮定される。それゆえ、入力する項の後ろにはピリオド（と改行コード）を置かなければならない。］

項の出力のための組込み述語 write(項).　原子式 write(τ) は、

　　　カレント出力ストリーム上の次の項がτと等しい

という命題を表す。　従って、Prologプログラムの実行中にゴール write(τ) が起動されると、Prolog言語処理系は、単にカレント出力ストリームに τ を書き出すだけである。

改行のための組込み述語 nl.　原子式 nl は、

　　　カレント出力ストリーム上の次の箇所に改行コードが現れる

という命題を表す。従って、Prologプログラムの実行中にゴール nl が起動されると、Prolog言語処理系は、単にカレント出力ストリームに改行コードを書き出すだけである。

字下げのための組込み述語 tab(整数または変項).　原子式 tab(τ) は、

　　カレント出力ストリーム上の次の箇所に τ 個の空白が現れる

という命題を表す。　従って、Prologプログラムの実行中にゴール tab(τ) が起動されると、Prolog言語処理系は、単にカレント出力ストリームに空白を τ 個だけ書き出す。もちろん、τが整数を表さない場合はエラーとなる。

これらの組込み述語の使用例を次に挙げよう。

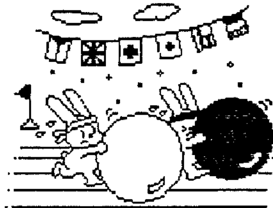
例５．１０(オーム返し)　組込み述語 read, write を用いれば、

　　項をカレント入力ストリームから読み込んで
　　これをそのままカレント出力ストリームに書き出すプログラム

をPrologで記述するのは容易である。例えば、図５．１３の様なプログラムを構成して、これを図５．１４の様に実行することができる。

```
:- lc.                          % ファイル読み込みの際の環境設定

/************************
* 例5．１０　オーム返し *
************************/
echo :-
     read(Term),
     write(Term).
```

図５．１３　オーム返しをするプログラム



```
<user> ?-  echo.
>  1+2+3.
1+2+3
yes

<user> ?-  echo.
>  +(1,2).
1+2
yes

<user> ?-  echo.
>  Variable.
_22
yes

<user> ?-  echo.
>  'This is an atom.'.
This is an atom.
yes

<user> ?-  echo.
>  "This is a string.".
This is a string.
yes
```

図5.14 "オーム返しのプログラム"の実行例［下線部がユーザの入力］

例5.11(英-和の間の単語翻訳)　組込み述語 read, write を用いれば、

英語または日本語の単語をカレント入力ストリームから読み込み
これをそれぞれ日本語または英語に翻訳してカレント出力ストリームに書き出す
プログラム

をPrologで記述するのは容易である。例えば、組込み述語 nl, tab を使って出力の仕方に工夫すれば、図5.15の様なプログラムを構成して、これを図5.16の様に実行することができる。[ここで、図5.15のプログラム内の """" は1個の二重引用符(")だけで構成される文字列型の項を表す。一般に、文字列を二重引用符で囲むと文字列型の項ができるが、元の文字列が二重引用符を含む場合は、それらのそれぞれに対して2個の連続した二重引用符で代用する。]

```
:- lc.                              % ファイル読み込みの際の環境設定

/*********************************
* 例5.11　英-和の間の単語翻訳 *
*********************************/
english_japanese(mouse,    鼠).        % 辞書
english_japanese(cow,      牛).
english_japanese(tiger,    虎).
english_japanese(rabbit,   兎).
english_japanese(dragon,   竜).
english_japanese(snake,    蛇).
english_japanese(horse,    馬).
english_japanese(sheep,    羊).
english_japanese(monkey,   猿).
english_japanese(bird,     鳥).
english_japanese(dog,      犬).
english_japanese(wild_boar,猪).

look_up(Word, Equivalent) :-
    english_japanese(Word, Equivalent).
look_up(Word, Equivalent) :-
    english_japanese(Equivalent, Word).

translate :-
    read(Word),
    look_up(Word, Equivalent),
    write(""""), write(Word), write("""という単語の訳語は"), nl,
    tab(8), write(""""), write(Equivalent), write("""    です。").
```

図5.15　英-和の間で単語の訳語を見つけるプログラム

```
<user> ?-  translate.
 >  dragon.
"dragon"という単語の訳語は
        "竜"    です。
yes

<user> ?-  translate.
>  猪.
"猪"という単語の訳語は
        "wild_boar"    です。
yes

<user> ?-  translate.
>  cat.

no
```

図5.16　“英-和の間の単語翻訳プログラム”の実行例

5．6　Ｐｒｏｌｏｇによるパズルの解法

4．3節で見た様に、Prolog言語処理系には試行錯誤で解を探索する機能が備わっている。従って、単純な試行錯誤だけで解けるパズルは、Prologプログラムを用いて容易に解くことができる。例えば次の通り。

例5．12(虫食い算)　虫食い算

```
    □□
   ×8□
  ─────
   □□□
   □□
  ─────
  □□□□
```

は、単純な試行錯誤を繰り返すだけで解けるので、Prologの処理系の下で容易に解くことができる。[**虫食い算**(alphametics)は、与えられた計算パターンを持つ様に、虫食いを表す箱□のそれぞれを数字で置き換える問題である。但し、置き換えに際して、最上位の位を表す箱には０以外の数字を入れる。]

```
:- lc.                              % ファイル読み込みの際の環境設定

/**********************
* 例5.12　虫食い算 *
**********************/
digit(0).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
digit(8).
digit(9).

nonzero_digit(X) :-
     digit(X),
     X =¥= 0.

worm(X1, X0, Y0) :-
     digit(X0),                     % Xの行
     nonzero_digit(X1),
     X is 10*X1+X0,
     W is X*8,                      % Wの行
     W < 100,                          % W>9 は X1>0 から導かれる
    %
     digit(Y0),
     V is X*Y0,                     % Vの行
     V >= 100,
    %
     Z is V+W*10,                   % Zの行
     Z >= 1000,
    %
     nl, tab(3), write(X),          % ***** 解答を出力 *****
     nl, tab(2), write("*8"), write(Y0),
     nl, tab(1), write("-----"),
     nl, tab(2), write(V),
     nl, tab(2), write(W),
     nl,         write("------"),
     nl, tab(1), write(Z).
```

図5.17　虫食い算を解くプログラム

```
<user> ?-  worm(X1,X0,Y0).

  12
 *89
-----
 108
 96
------
1068
X1 = 1
X0 = 2
Y0 = 9 ;

no
```

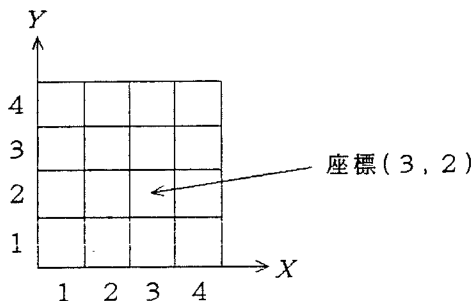
図５．１８ “虫食い算のプログラム”の実行例

例えば、それぞれの箱□を変数で置き換えて上の計算パターンを

$$
\begin{array}{rcl}
X_1\ X_0 & \cdots & X = X_1 * 10 + X_0 \\
\times\ 8\ \ Y_0 & & \\
\hline
V_2\ V_1\ V_0 & \cdots & V = V_2 * 100 + V_1 * 10 + V_0 \\
W_1\ W_0\ \ \ & \cdots & W = W_1 * 10 + W_0 \\
\hline
Z_3\ Z_2\ Z_1\ Z_0 & \cdots & Z = Z_3 * 1000 + Z_2 * 100 + Z_1 * 10 + Z_0
\end{array}
$$

と表して X_0, X_1, Y_0 に関して試行錯誤を行うことにすれば、図５．１７の様なプログラムが得られる。このプログラムの実行例は図５．１８の通りである。

例５．１３（４-Ｑｕｅｅｎｓ問題）　**４-Ｑｕｅｅｎｓ問題**(the 4 queens problem)は４×４の升目の盤にチェスのクイーンを４個、お互いに攻撃しないように配置する問題である。［クイーンは、縦横斜めにその勢力を及ぼすので、将棋の飛車と角行を合わせた様な駒である。］　この問題を解くために４×４の升目の盤を



```
:- lc.                                  % ファイル読み込みの際の環境設定

/********************************
* 例5.13   4-Queens問題 *
********************************/
point(1, _).
point(2, _).
point(3, _).
point(4, _).

queen4(X1, X2, X3, X4) :-
    point(X1, 1),
    point(X2, 2),
        not_attack(point(X2,2), point(X1,1)),
    point(X3, 3),
        not_attack(point(X3,3), point(X1,1)),
        not_attack(point(X3,3), point(X2,2)),
    point(X4, 4),
        not_attack(point(X4,4), point(X1,1)),
        not_attack(point(X4,4), point(X2,2)),
        not_attack(point(X4,4), point(X3,3)),
                nl,
    write_row(X4), nl,
    write_row(X3), nl,
    write_row(X2), nl,
    write_row(X1), nl.

not_attack(point(X1,Y1), point(X2,Y2)) :-
    X1 =¥= X2,
            % 条件 Y1=¥=Y2 の成立する場合だけこの述語が呼び出される
    X1+Y1 =¥= X2+Y2,
    X1-Y1 =¥= X2-Y2.

write_row(1) :- write("Q+++").
write_row(2) :- write("+Q++").
write_row(3) :- write("++Q+").
write_row(4) :- write("+++Q").
```

図5.19　4-Queens問題を解くプログラム

```
<user> ?-  queen4(X1,X2,X3,X4).

++Q+
Q+++
+++Q
+Q++

X1 = 2
X2 = 4
X3 = 1
X4 = 3  ;

+Q++
+++Q
Q+++
++Q+

X1 = 3
X2 = 1
X3 = 4
X4 = 2  ;

no
```

図５．２０　“４-Ｑｕｅｅｎｓ問題を解くプログラム”の実行例

という座標系で表すことにすれば、この問題は、条件

$\forall i \forall j$ [$i \neq j$ → 座標 $(X_I, Y_I), (X_J, Y_J)$ のクイーンが互いに攻撃し合わない]，

すなわち

$$\forall i \forall j [\, i \neq j \rightarrow X_I \neq X_J \wedge Y_I \neq Y_J \wedge X_I + Y_I \neq X_J + Y_J \wedge X_I - Y_I \neq X_J - Y_J \,]$$

を満たすような座標の集合

$$\{(X_1, Y_1), (X_2, Y_2), (X_3, Y_3), (X_4, Y_4)\} \subseteq \{1, 2, 3, 4\}^2$$

を見つけ出すことに他ならない。 ここで、４個の座標に関して $Y_1 = 1$，$Y_2 = 2$，$Y_3 = 3$，$Y_4 = 4$ と仮定しても一般性を失わないから、この仮定の下で所要の X_1，X_2，X_3，X_4 を試行錯誤で見つけ出すことにすれば、４-Ｑｕｅｅｎｓ問題を解くPrologプログラムとして例えば図５．１９のものが得られる。このプログラムの実行例は図５．２０に示される。　［このプログラムは、一般の N-Ｑｕｅｅｎｓ問題に対して一般化が難しく、この点であまり好ましいものとは思えない。しかし、これは、試行錯誤の手続きをそのままプログラムに反映した点で、分かり易い。］

```
:- lc.                                  % ファイル読み込みの際の環境設定

/*************************************************
* 例5.14  4-Queens問題（全解の自動生成） *
*************************************************/
point(1, _).
point(2, _).
point(3, _).
point(4, _).

queen4 :-
    point(X1, 1),
    point(X2, 2),
        not_attack(point(X2,2), point(X1,1)),
    point(X3, 3),
        not_attack(point(X3,3), point(X1,1)),
        not_attack(point(X3,3), point(X2,2)),
    point(X4, 4),
        not_attack(point(X4,4), point(X1,1)),
        not_attack(point(X4,4), point(X2,2)),
        not_attack(point(X4,4), point(X3,3)),
                nl,
    write_row(X4), nl,
    write_row(X3), nl,
    write_row(X2), nl,
    write_row(X1), nl,
    fail.

not_attack(point(X1,Y1), point(X2,Y2)) :-
    X1 =¥= X2,
              % 条件 Y1=¥=Y2 の成立する場合だけこの述語が呼び出される
    X1+Y1 =¥= X2+Y2,
    X1-Y1 =¥= X2-Y2.

write_row(1) :- write("Q+++").
write_row(2) :- write("+Q++").
write_row(3) :- write("++Q+").
write_row(4) :- write("+++Q").
```

図5.21　4-Queens問題の全ての解を自動的に生成するプログラム

［下線部は図5.19と違う部分］

```
<user> ?-  queen4.

++Q+
Q+++
+++Q
+Q++

+Q++
+++Q
Q+++
++Q+

no
```

図５．２２　“４-Ｑｕｅｅｎｓ問題の全解自動生成プログラム”の実行例

例５．１４(全解の自動生成)　常に失敗する組込み述語 **fail** を用いればPrologで解ける問題の解の全てを自動的に生成することも可能である。例えば、先の例５．１３で考えた４-Ｑｕｅｅｎｓ問題の解を全て自動的に生成するには、図５．１９のプログラムを図５．２１の様にほんの少し書き換えるだけでよい。この図５．２１のプログラムの実行例は図５．２２の通りである。

例５．１５(ハノイの塔問題;Prologの試行錯誤機能に頼りすぎてはいけない例)　３本の棒と大きさの異なる n 枚の円盤があり、円盤が全て下から大きい順に１つの棒に差し込まれている。この時、

　　積み重ねられた円盤の大きさは常に下の方が大きい

という状態を保ったまま円盤を１枚ずつ１本の棒から別の棒に移すという操作を繰り返し、最終的に全ての円盤を指定した棒に積み上げるための手順を見つけ出したい。　これを**ハノイの塔問題** (the "towers of Hanoi" problem)という。［図５．２３を参照。］

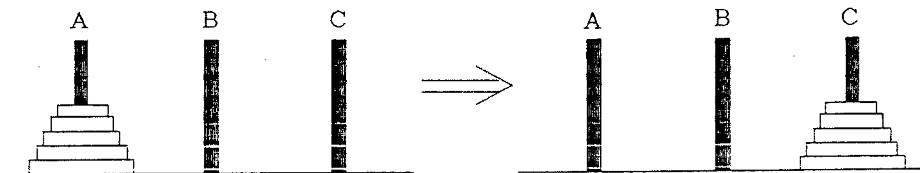


図５．２３　ハノイの塔問題（ n＝５の場合）

この問題に対して“虫食い算や4-Queens問題の様に試行錯誤だけで解けないか？”ということを最初に思い付くであろうが、この方針では何について試行錯誤するかでまず考えてしまう。［虫食い算や4-Queens問題では解（の候補）を固定した個数の変項で表しこれらについて試行錯誤すれば良かった訳であるが、ハノイの塔の問題では求める解は円盤の移動作業の列であってこれを固定した個数の変項で表せないのである。］ 実際には、Prologには不特定多数のデータの列を表すために“リスト”（6.1節）と呼ばれる道具が備わっているから、これを用いて円盤の移動動作の列を表し解を試行錯誤で求めることも可能である。ところが、この解法は3^n個の状態から成る状態空間の探索に他ならず、無限探索の可能性も残す。無限探索が起きないように工夫したとしても、nが少し大きいと解の探索に時間がかかりすぎて事実上計算不能になる。

```
:- lc.                                % ファイル読み込みの際の環境設定

/****************************
* 例5.15  ハノイの塔問題 *
****************************/
hanoi(N, From, To, Via) :-
     move(N, From, To, Via, 0).

move(1, From, To, Via, Tab) :-
     print_a_move(1, From, To, Tab).
move(N, From, To, Via, Tab) :-         % 円盤1～円盤Nを
     N > 1,                            % 棒 From から棒 To へ移す。
     N1 is N-1,                        % ここで、円盤1は最小の円盤を表す。
     Tab3 is Tab+3,
     move(N1, From, Via, To, Tab3),
     print_a_move(N, From, To, Tab),
     move(N1, Via, To, From, Tab3).

print_a_move(N, From, To, Tab) :-
     tab(Tab),
     write("円盤("),
     write(N),
     write(")を棒("),
     write(From),
     write(")から棒("),
     write(To),
     write(")へ移す。"),
     nl.
```

図5.24 ハノイの塔問題を解くプログラム

```
<user> ?-  hanoi(4,'A','C','B').
            円盤(1)を棒(A)から棒(B)へ移す。 ┐
        円盤(2)を棒(A)から棒(C)へ移す。     │
            円盤(1)を棒(B)から棒(C)へ移す。 │ move(3,'A','B','C',3)
    円盤(3)を棒(A)から棒(B)へ移す。         ├ によって出力された
            円盤(1)を棒(C)から棒(A)へ移す。 │ 移動手順
        円盤(2)を棒(C)から棒(B)へ移す。     │
            円盤(1)を棒(A)から棒(B)へ移す。 ┘
円盤(4)を棒(A)から棒(C)へ移す。
            円盤(1)を棒(B)から棒(C)へ移す。 ┐
        円盤(2)を棒(B)から棒(A)へ移す。     │
            円盤(1)を棒(C)から棒(A)へ移す。 │ move(3,'B','C','A',3)
    円盤(3)を棒(B)から棒(C)へ移す。         ├ によって出力された
            円盤(1)を棒(A)から棒(B)へ移す。 │ 移動手順
        円盤(2)を棒(A)から棒(C)へ移す。     │
            円盤(1)を棒(B)から棒(C)へ移す。 ┘

yes
```

図5.25　“ハノイの塔問題のプログラム”の実行例

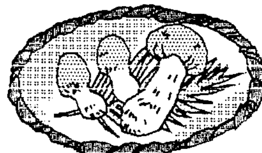
しかし、よく考えると

① 棒Ａに差し込まれているｎ枚の円盤の内、上の($n-1$)枚を棒Ｂに移し、

② 棒Ａの残りの１枚の円盤を棒Ｃに移し、最後に

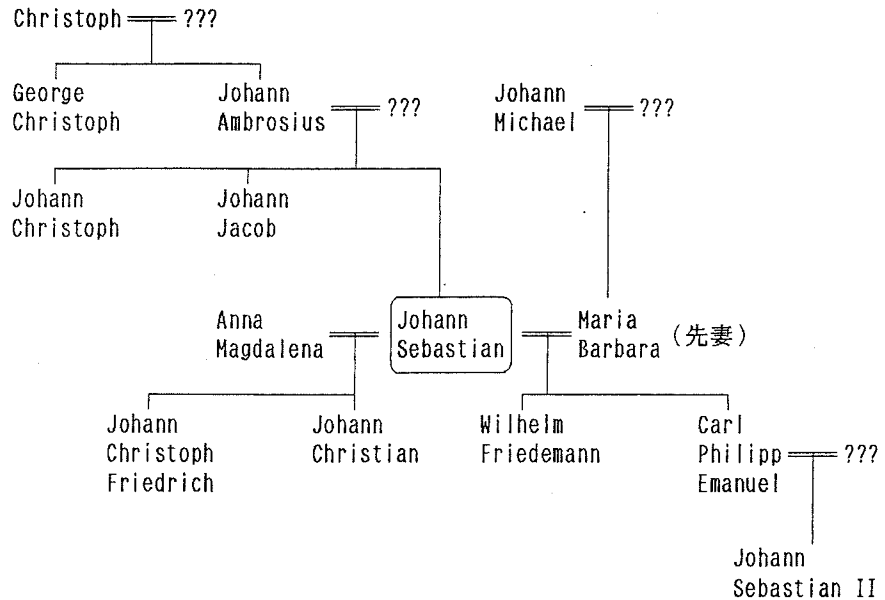
③ 棒Ｂに移した($n-1$)枚の円盤を棒Ｃに移す

という決まりきった手順で、棒Ａのn枚の円盤をそっくり棒Ｃに移せることに気づく。この事実は、小さい方から選んだ($n-1$)枚をそっくり移すための手順が分かれば、それを使ってn枚をそっくり移すための手順も機械的に記述できることを示している。最も小さな円盤１枚の移動手順は明かであるから、結局、これで一般的にn枚の円盤の移動手順を再帰的に記述できることになる。この考えをPrologプログラムとして表せば図５．２１の様になり、このプログラムを実行させると例えば図５．２２の様になる。［容易に分かるように、円盤がn枚ある場合、このプログラムの与える解は $1+2+4+\cdots+2^{n-1}=2^n-1$ 回の円盤の移動から成る。これは最適解、すなわち円盤移動の回数の最も少ない解である。］



演習問題５

５.１　J. S. Bach 一族の家系図（の一部）



について

任意に与えられた人の子孫を見つけるためのPrologプログラム

を作成せよ。その際、例えば２つの述語

father(x, y)　⇔　xはyの父親である，

mother(x, y)　⇔　xはyの母親である

を用いて家系図のデータを表し、２つの述語

parent(x, y)　⇔　xはyの親である，

ancestor(x, y)　⇔　xはyの祖先（すなわちyはxの子孫）である

を用いてそれぞれ親子関係，祖先-子孫関係を表せ。

５.２　power(x, n, p) ⇔ $p = x^n$ という意味の述語 power を再帰的に定義することによって

任意に与えられた整数（または実数）xと非負整数nを基に

べき乗 x^n を計算するPrologプログラム

を作成せよ。その際、次の２つの方針に従うと、それぞれどの様なプログラムが得られるか？

(1)　漸化式

$$f(x, n) = \begin{cases} 1 & (n=0 \text{ の場合}) \\ x \times f(x, n-1) & (n \geqq 1 \text{ の場合}) \end{cases}$$

によってべき乗関数 $f(x, n) = x^n$ が定義されることに着目し、この漸化式に忠実に従った形で再帰的に述語 power を定義する。

(2) 漸化式

$$g(x, n) = \begin{cases} 1 & (n=0 \text{ の場合}) \\ g(x^2, n/2\text{の整数部}) & (n \geqq 1,\ n:\text{偶数 の場合}) \\ x \times g(x^2, n/2\text{の整数部}) & (n \geqq 1,\ n:\text{奇数 の場合}) \end{cases}$$

によってべき乗関数 $g(x, n) = x^n$ が定義されることに着目し、この漸化式を基に末端再帰法で述語 power を定義する。

5.3 漸化式

$$h(n) = \begin{cases} 1 & (n=1 \text{ または } 2 \text{ の場合}) \\ h(n-1) + h(n-2) & (n \geqq 3 \text{ の場合}) \end{cases}$$

によって決まる関数 h の計算をするPrologプログラム，すなわち

任意に与えられた正整数 n を基に $h(n)$ を計算するPrologプログラム

を作成せよ。その際、次の２つの方針に従うと、それぞれどの様なプログラムが得られるか？

[数列 $h(1)$, $h(2)$, $h(3)$, ... をフィボナッチ数列という。]

(1) 上の漸化式を忠実に反映した再帰法を用いる。

(2) 上の漸化式を基にして末端再帰法を用いる。

5.4 再帰法を用いて

任意に与えられた非負整数の最小公倍数(largest common multiplier)を計算するPrologプログラム

を作成せよ。

5.5 再帰法を用いて

任意に与えられた正整数が素数(prime number)かどうかを判定するPrologプログラム

を作成せよ。

5.6 問題５．１で構成したPrologプログラムについて、

(1) 次の３つの質問を行え。

① Maria Barbara (Bach) の子孫３人を全て見つける。

② Johann Ambrosius (Bach) の子孫８人を全て見つける。

③ Johann Sebastian (Bach) の祖先２人を全て見つける。

(2)　問(1)①の質問を行った際の実行の様子を、デバッガを用いて徹底的に追跡･表示させてみよ。

(3)　問(1)①の質問を行った際の実行の様子を、スパイ点を利用して追跡･表示させてみよ。但し、スパイ点には述語 ancestor の処理を選び、スパイ点の出来事だけを表示させるものとする。

５．７　再帰法を用いて

任意に与えられた数以下の正整数を全て出力するPrologプログラム

を作成せよ。

５．８　再帰法を用いて

任意に与えられた正整数以下の素数を全て出力するPrologプログラム

を作成せよ。　［ヒント．例えば n以下の素数を生成する手続きを generate_primes(n) と表す。この手続きによって生成される素数の集合 primes(n) に関して漸化式

$$\mathrm{primes}(n)=\begin{cases}\emptyset & n<2\\ \{n\}\cup \mathrm{primes}(n-1) & n\geqq 2,\ n\text{は素数}\\ \mathrm{primes}(n-1) & n\geqq 2,\ n\text{は素数でない}\end{cases}$$

が成り立つことに注目すればよい。］

５．９　虫食い算

```
   □□
  ×7□
 ─────
  □□□
  □□
 ─────
  □□□
```

を解くPrologプログラムを作成せよ。

５．１０　虫食い算

```
      6□□
    ×□6□
   ───────
     □□□6
    □□□□
   6□□□
  ─────────
   □6□□6□
```

を解くPrologプログラムを作成せよ。

第6章　Ｐｒｏｌｏｇプログラミング技法

先の第５章では、Prolog言語処理系に備わっている単一化の機能，算術計算の機能，（広義の）データベースを扱う機能，再帰法を許す機能，入出力の機能，試行錯誤で解を探索する機能を用いて様々なプログラムを作成／実行した。もちろん、これらの機能だけを用いて他の様々な問題に対処することもできるが、場合によっては、プログラミングの際に不自由感を覚えることもある。例えば、例５．９の４-Ｑｕｅｅｎｓ問題を一般化して得られる“*N*-Ｑｕｅｅｎｓ問題”を解こうとしても、これらの機能／道具だけでは綺麗にプログラミングができない。

そこでこの章では、Prolog言語処理系に備わっている他の有用な機能について説明しよう。まず６．１節では、不定個数の要素の列を表すための道具(リスト)について述べる。続く６．２節では、実行制御のための述語(カット述語,fail,true,call,否定述語,条件分岐述語,repeat)について述べ、“試行錯誤による解探索”の効率化,プログラムの構造化に目を向ける。　そして最後の６．３節では、（広義の）データベース／知識ベースを操作するための述語(assert,asserta,assertz,clause,abolish,retract)について述べる。

６．１　リストの扱い

Prologでは、不定個数の要素の列を表すためにリスト(list)と呼ばれるデータ構造を用いる。具体的には、要素の列 $a_1, a_2, a_3, \ldots, a_n$ は、処理系内にあらかじめ組み込まれた２引数関数文字・(ドット,dot,と読む)と特殊アトム [] (空リスト,null list,と呼ぶ)を用いて、

$$\cdot(a_1, \cdot(a_2, \cdot(a_3, \ldots, \cdot(a_n, []) \ldots)))$$

あるいは、これを略記して、

$$[a_1, a_2, a_3, \ldots, a_n]$$

と表される。 また、$a_1, a_2, a_3, \ldots, a_n$ で始まる要素列は、それ以降の要素列を $Rest$ で表して、

$$\cdot(a_1, \cdot(a_2, \cdot(a_3, \ldots, \cdot(a_n, Rest) \ldots)))$$

あるいは、これを略記して、

$[a_1, a_2, a_3, \ldots, a_n \mid Rest]$

と表される。［ここで、通常の使い方では $Rest$ はリスト構造のデータを表すが、一般にはその様な制限はなく、$Rest$ がアトムを表してもよい。］

リストの扱いに慣れるために、まず、リストを操作するプログラムの例をいくつか与えよう。

例6.1(リストの結合・分離)　Prologでは再帰法によって述語を定義できるので、

任意に与えられた２つのリスト $[a_1, a_2, \ldots, a_m]$ と $[b_1, b_2, \ldots, b_n]$ を結合(append)して新しいリスト $[a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n]$ を構成するプログラム

をPrologで記述するのは容易である。例えば、図6.1の様なプログラムを構成して、これを図6.2の様に実行することができる。Prologプログラムの公理的性格により、このプログラムは、図6.3に示す様に、

任意に与えられたリスト $[a_1, a_2, \ldots, a_n]$ を２つのリスト $[a_1, a_2, \ldots, a_i]$ と $[a_{i+1}, a_{i+2}, \ldots, a_n]$ に分離／分割するプログラム

として用いることもできる。

```
:- lc.                                  % ファイル読み込みの際の環境設定

/******************************
* 例6.1  リストの結合・分離 *
******************************/
append([], List, List).
append([Head | Tail1], List2, [Head | Tail3]) :-
    append(Tail1, List2, Tail3).

                          % この規則は次の規則と等価である。
                          %   append(List1, List2, List3) :-
                          %       List1 = [Head | Tail1],
                          %       List3 = [Head | Tail3],
                          %       append(Tail1, List2, Tail3).
```

図6.1　リストを結合・分離するためのプログラム

```
<user> ?-  append([john, eats, the, cake], [with, evident, satisfaction], W).

W = [john,eats,the,cake,with,evident,satisfaction]

yes
```

図６．２　append述語（図６．１）をリストの結合に用いる例

```
<user> ?-  append(X, Y, [a, b, c]).

X = []
Y = [a,b,c]  ;

X = [a]
Y = [b,c]  ;

X = [a,b]
Y = [c]  ;

X = [a,b,c]
Y = []  ;

no
```

図６．３　append述語（図６．１）をリストの分離に用いる例

例６．２(リストの反転)　Prologでは再帰法によって述語を定義できるので、

任意に与えられたリスト［$a_1, a_2, \ldots, a_n$］の要素の順序を反転(reverse)して新しいリスト［$a_n, \ldots, a_2, a_1$］を作り出すプログラム

をPrologで記述するのは容易である。例えば、append述語を利用して図６．４の様なプログラムを構成して、これを図６．５の様に実行することができる。また、図６．６に示す様に、末端再帰法を用いてより計算効率の良いプログラムを組むこともできる。

```
:- lc.                                % ファイル読み込みの際の環境設定

/*******************************
* 例6.2(その1) リストの反転 *
*******************************/
reverse1([], []).
reverse1([X | L], Rev_XL) :-
     reverse1(L, Rev_L),
     append(Rev_L, [X], Rev_XL).

append([], L, L).
append([X | T1], L2, [X | T3]) :-
     append(T1, L2, T3).
```

図6.4 append述語を使ってリストを反転するプログラム

```
<user> ?-  reverse1([a, b, c, d], Rev_L).

Rev_L = [d,c,b,a]

yes
```

図6.5 リスト反転プログラム(図6.4)の実行例

```
:- lc.                                % ファイル読み込みの際の環境設定

/**********************************************
* 例6.2(その2) 末端再帰法によるリストの反転 *
**********************************************/
reverse2(L, Rev_L) :-
     rev(L, [], Rev_L).

rev([], L, L).
rev([X | T1], L2, Rev_L) :-
     rev(T1, [X | L2], Rev_L).
```

図6.6 末端再帰法でリストを反転するプログラム

リストを用いると、様々な問題に対してプログラミングが容易になる。例えば次の通り。

例6.3(一筆書き)　線を組み合わせて構成される図は、線の端点間の接続関係だけに注目する場合、リストを用いて単純な形で表すことができる。例えば、各線の端点に名前を付け一般に2つの端点 *u* と *v* の間の線を *u*-*v* または *v*-*u* と表すことにすれば、図6.7はリストを用いて [a-b,a-c,a-d,b-c,b-d,c-d,c-e,d-e] と表せる。ここで、この表記法を採用しPrologの試行錯誤の機能を利用することにすれば、

線を組み合わせて構成される図が引数として任意に与えられた時、この図が一筆書きで書けるかどうかを判定し、もし一筆書き可能ならその手順を見つけるプログラム

としては例えば図6.8の様なものが得られる。このプログラムの実行例は図6.9の通りである。[一筆書きが可能かどうかは、実際には、各端点の**次数**(degree;すなわち端点につながっている線の本数)を調べるだけで分かる。なぜなら、“一筆書き可能⇔次数が奇数の端点が2個以下”であることが知られているからである。また、一筆書きが可能な場合にその手順を見つけ出す機械的な方法もよく知られている。]

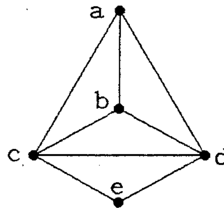


図6.7　線を組み合わせて構成される図の例

```
:- lc.                                  % ファイル読み込みの際の環境設定

/********************
* 例6.3　一筆書き *
********************/
draw([], [Point]).
draw(Picture, [X, Y | Path]) :-
     choose(Picture, X-Y, Picture1),
     draw(Picture1, [Y | Path]).

choose([X-Y | Picture], X-Y, Picture).
choose([X-Y | Picture], Y-X, Picture).
choose([Line | Picture], Choosed_Line, [Line | Picture1]) :-
     choose(Picture, Choosed_Line, Picture1).
```

図6.8　一筆書きのプログラム

```
<user> ?-  draw([a-b, a-c, a-d, b-c, b-d, c-d, c-e, d-e], Path).

Path = [a,b,c,a,d,c,e,d,b]  ;

Path = [a,b,c,a,d,e,c,d,b]  ;

Path = [a,b,c,d,a,c,e,d,b]  ;

Path = [a,b,c,d,e,c,a,d,b]

yes
```

図6．9　“一筆書きのプログラム”の実行例

例6．4(覆面算)　　覆面算

```
  SEND
 +MORE
 ─────
 MONEY
```

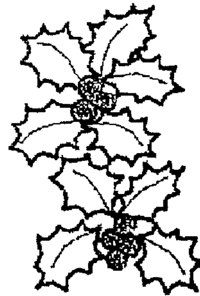
は、単純な試行錯誤を繰り返すだけで解くことができる。その際、リストを用いて実行効率の上がる様に工夫すると、例えば図6．10の様なプログラムが得られる。このプログラムの実行例は図6．11の通りである。［**覆面算**(alphametics)は、数字の代わりに英字,仮名,漢字などの覆面文字を用いた計算パターンが与えられた時、この計算パターンを持つ様に、覆面文字のそれぞれを数字で置き換える問題である。但し、置き換えに際して、異なる覆面文字は異なる数字で置き換え、最上位の数字を表す覆面文字は０以外の数字で置き換える。］

```
:- lc.                                    % ファイル読み込みの際の環境設定

/************************************************
* 例6.4 覆面算, SEND+MORE=MONEY *
************************************************/
send_more_money :-
    delete(D, [0,1,2,3,4,5,6,7,8,9], Digits1),  % 一の位
    delete(E, Digits1, Digits2),
    Y is (D+E) mod 10,
    delete(Y, Digits2, Digits3),
    Carry1 is (D+E)//10,
   %
    delete(N, Digits3, Digits4),                % 十の位
    R is (E-Carry1-N+20) mod 10,
    delete(R, Digits4, Digits5),
    Carry2 is (N+R+Carry1)//10,
   %
    O is (N-Carry2-E+20) mod 10,                % 百の位
    delete(O, Digits5, Digits6),
    Carry3 is (E+O+Carry2)//10,
   %
    delete(S, Digits6, Digits7),                % 千, 万の位
    S > 0,
    M is (O-Carry3-S+20) mod 10,
    M > 0,
    delete(M, Digits7, Digits8),
    M is (S+M+Carry3)//10,
   %
    Send is 1000*S+100*E+10*N+D,                % ***** 解の出力 *****
    More is 1000*M+100*O+10*R+E,
    Money is 10000*M+1000*O+100*N+10*E+Y,
    tab(5), write(Send), nl,
    tab(4), write("+"), write(More), nl,
    tab(3), write("-------"), nl,
    tab(4), write(Money), nl,
    nl,
    fail.

/***** delete述語 *****/
delete(X, [X | Digits], Digits).
delete(X, [Y | Digits2], [Y | Digits3]) :-
    delete(X, Digits2, Digits3).
```

図6.10 覆面算 SEND+MORE=MONEY を解くプログラム

```
<user> ?-  send_more_money.
     9567
    +1085
   -------
    10652


no
```

図６．１１ “覆面算 SEND+MORE=MONEY を解くプログラム”の実行例



例６．５(素数の生成;エラトステネスの篩(ふるい)い) 与えられた正整数以下の素数を全て生成するための方法としては、大きく分けて２種類の計算手順がよく知られている。その内の１つは、与えられた数以下の正整数のそれぞれについて“それが素数であるかどうか”を判定する方法で、再帰法と算術計算の機能を組み合わせて容易にPrologで記述できる。もう１つの方法は、**エラトステネスの篩い**(Eratosthenes' sieve)と呼ばれ、

２以上,与えられた数以下の整数を最初に篩いに入れ、

他の数の倍数になっている数を次々と篩いから落としてゆく

というものである。 この方法をプログラムで記述する際には“篩いの中の整数の集合”を表わすデータ構造が必要になるが、Prologの場合はこのデータ構造としてリストを用いることができる。リスト構造を用いて“エラトステネスの篩い”を行うPrologプログラムとしては、例えば図６．１２の様なものが考えられる。このプログラムの実行例は図６．１３の通りである。

```
:- lc.                                  % ファイル読み込みの際の環境設定

/***********************************************
* 例6.5  ”エラトステネスの篩”による素数の生成 *
***********************************************/
generate_primes(N, List_of_primes) :-
     generate_integers(2, N, List_of_integers),  % List_of_integers =
     sift(List_of_integers, List_of_primes).     %   [2,3,4, ... ,N]
                                                 % となる。
generate_integers(M, N, []) :-
     M > N.
generate_integers(M, N, [M | Integers]) :-      % Integers =
     M =< N,                                    %   [M+1,M+2, ... ,N]
     M1 is M+1,                                 % となることが期待される。
     generate_integers(M1, N, Integers).

sift([], []).
sift([P | Integers], [P | Primes]) :-           % この時点では P は素数。
     filter(P, Integers, New_integers),         % New_integers =
     sift(New_integers, Primes).                %   Integersから
                                                %   P の倍数を除去したもの
                                                % となる。
filter(P, [], []).
filter(P, [K | Integers], [K | New_integers]) :-
     K mod P =¥= 0,
     filter(P, Integers, New_integers).
filter(P, [K | Integers], New_integers) :-
     K mod P =:= 0,
     filter(P, Integers, New_integers).
```

図6.12　“エラトステネスの篩い”で素数を生成するプログラム

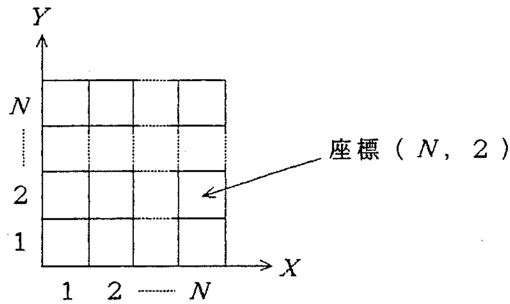
```
<user> ?-  generate_primes(100, List_of_primes).

List_of_primes = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,7
9,83,89,97]

yes
```

図6.13　“エラトステネスの篩いのプログラム”で１００以下の素数を生成する実行例

例6.6(*N*-Ｑｕｅｅｎｓ問題)　例5.１３～１４の4-Ｑｕｅｅｎｓ問題を一般化して得られる***N*-Ｑｕｅｅｎｓ問題**(the *N* queens problem)，すなわち

　　正整数 *N*が任意に与えられた時、*N*× *N*の升目の盤にチェスのクィーンを *N*個，
　　お互いに攻撃しないように配置する問題

を考えよう。この問題を解くために、例5.１３～１４の場合と同様に *N*× *N*の升目の盤を



という座標系で表し、この中の座標に試行錯誤でクィーンを置いてゆくプログラムを組むことはできる。 しかし、*N*-Ｑｕｅｅｎｓ問題に対しては、例5.１３のプログラムをそのまま一般化することは難しく、普通、リストを用いて図6.１４の様に一般化する。 このプログラムの実行例は図6.１5の通りである。

```
:- lc.                                       % ファイル読み込みの際の環境設定

/******************************
* 例6.6   N-Ｑｕｅｅｎｓ問題 *
******************************/
n_queens(N, Solution) :-
    generate_integers(1, N, Integers),          % Integers = [1,2, ... ,N]。
    queens(Integers, Integers, [], Solution),   % queensの第一引数はクィー
    write_a_solution(Solution, N).              % ンの置かれていないX座標
                                                % のリスト，第二引数はクィ
                                                % ーンの置かれていないY座
                                                % 標のリスト，第三引数はク
                                                % ィーンの置かれている座標
                                                % のリスト，第四引数は最終
                                                % 的にN個のクィーンの置か
                                                % れる座標のリスト，を表す。
```

図6.１４　*N*-Ｑｕｅｅｎｓ問題を解くプログラム（次ページに続く）

```
generate_integers(M, N, []) :-
     M > N.
generate_integers(M, N, [M | Integers]) :-
     M =< N,
     M1 is M+1,
     generate_integers(M1, N, Integers).

queens([], [], Solution, Solution).
queens(Integers1, [Y | Integers2], Points, Solution) :-
     delete(X, Integers1, Rest1),
     not_attack(point(X,Y), Points),
     queens(Rest1, Integers2, [point(X,Y) | Points], Solution).

delete(X, [X | Integers], Integers).
delete(X, [X1 | Integers], [X1 | Rest]) :-
     delete(X, Integers, Rest).

not_attack(point(X,Y), []).
not_attack(point(X,Y), [point(X1,Y1) | Points]) :-
     X+Y =¥= X1+Y1,
     X-Y =¥= X1-Y1,
     not_attack(point(X,Y), Points).

/***** 解の出力 *****/
write_a_solution([], N).
write_a_solution([point(X,Y) | Points], N) :-
     write_a_row(1, X, N),
     nl,
     write_a_solution(Points, N).

write_a_row(M, X, N) :-
     M > N.
write_a_row(M, X, N) :-
     M =< N,
     M =:= X,
     write("Q"),
     M1 is M+1,
     write_a_row(M1, X, N).
write_a_row(M, X, N) :-
     M =< N,
     M =¥= X,
     write("+"),
     M1 is M+1,
     write_a_row(M1, X, N).
```

図6.14 *N*-Ｑｕｅｅｎｓ問題を解くプログラム（前ページからの続き）

```
<user> ?-  n_queens(8, Solution).
+++Q++++
+Q++++++
++++++Q+
++Q+++++
+++++Q++
+++++++Q
++++Q+++
Q+++++++

Solution = [point(4,8),point(2,7),point(7,6),point(3,5),point(6,4),point(8,3),
point(5,2),point(1,1)]  ;
++++Q+++
+Q++++++
+++Q++++
++++++Q+
++Q+++++
+++++++Q
+++++Q++
Q+++++++

Solution = [point(5,8),point(2,7),point(4,6),point(7,5),point(3,4),point(8,3),
point(6,2),point(1,1)]

yes
```

図６．１５ “*N*-Ｑｕｅｅｎｓ問題を解くプログラム” の実行例（*N*＝８とする）

例６．７（自然言語の構文解析） ６つの文字列

① John dreams,

② John eats the cake,

③ John eats John,

④ the cake dreams,

⑤ the cake eats the cake,

⑥ the cake eats John

は、その意味を考えなければ、全て文法的に正しい英文と見ることができる。そして、例３．１０ではこのために必要な文法知識を（一階述語論理に基づく）公理系として表し、例４．４ではこの公理系を基に“与えられた文字列が英文と見なせるかどうかを判断したり、英文と見なせる文字列を生成したりする” Prologプログラムを構成した。このプログラムとその実行例はそれぞれ図６．１６，

図6.17の通りである。図6.17に示される様にこのプログラムに英文の候補を与える際にはその構造も同時に与えなければならないので、実際には図6.16のプログラムは**構文解析**(parsing;すなわち文の構造の解析)を全然行っていないことになる。

そこで、構造を明示しない形で英文の候補を受け取りその構造を解析／認識するプログラムを次に作ってみよう。まず、構造を表面に出さない様な英文の表現法としては、単に英文を構成する単語のリストを用いるのが自然である。また、プログラムの内部では通常、構文解析の容易さのために、**d-リスト**(difference list,**差分リスト**;double list,**重リスト**)と呼ばれるデータ構造を用いて単語列を表す。d-リストは単にリストを2個並べたものであり、例えば“the cake”という単語

```
:- lc.                                  % ファイル読み込みの際の環境設定

/****************************************************************
* 例6.7(その1)  構造を明示した6つの文字列を正しい英文と見る *
*                  ための文法知識                               *
****************************************************************/
sentence(X+Y) :-
     noun_phrase(X),
     verb_phrase(Y).

noun_phrase(X) :-
     proper_noun(X).
noun_phrase(X+Y) :-
     definite_article(X),
     noun(Y).

verb_phrase(X) :-
     intransitive_verb(X).
verb_phrase(X+Y) :-
     transitive_verb(X),
     noun_phrase(Y).

/***** 辞書 *****/
intransitive_verb(dreams).
transitive_verb(eats).
proper_noun('John').
definite_article(the).
noun(cake).
```

図6.16 例4.4で与えたプログラム

```
<user> ?-  sentence('John'+eats+the+cake).
no
<user> ?-  sentence('John'+(eats+(the+cake))).
yes
<user> ?-  sentence(S).
S = John+dreams  ;
S = John+(eats+John)  ;
S = John+(eats+(the+cake))  ;
S = the+cake+dreams  ;
S = the+cake+(eats+John)  ;
S = the+cake+(eats+(the+cake))  ;
no
```

図６．１７　例４．４で与えたプログラムの実行例

の列を表す場合は

([the, cake | *List*], *List*)　　但し *List* は任意のリストを表す

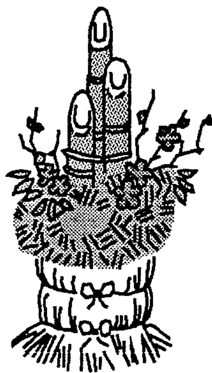
という格好のデータ構造になる。構文解析を行うためにｄ-リストを利用することにすれば、図６．１６のプログラムを例えば図６．１８の様に修正することができる。このプログラムの実行例は図６．１９の通りである。［ｄ-リストを用いずに、単にリストとリストを結合するための述語 append（例６．１）を用いて図６．２０の様なプログラムを作ることもできる。しかし、このプログラムは、結局は文法的に許される文を次々に生成して与えられた単語のリストと照合するものであるから、構文解析をしているとは言い難い。そこで、“プログラム内の規則の本体部にある３つのappend ゴール（すなわち図６．２０の下線部）を本体内の先頭に移す”という操作を図６．２０のプログラムに施すことにすれば、得られたプログラムは、“全ての文を生成して与えられた単語のリストと照合”という無駄な計算をしなくなる。しかし、今度は質問 ?- sentence(S). を行った場合に、この修正版プログラムはゴール append(Np,Vp,S) の実行を，従って無限計算を引き起こす。］

```
:- lc.                           % ファイル読み込みの際の環境設定

/*********************************************************
* 例6.7(その2)　d-リストを用いた構文解析                    *
*        --- 構造を明示しない6つの文字列を正しい英文と見る *
*            ための文法知識をd-リストを用いて記述する。    *
*********************************************************/
sentence(S) :-                   % S は英文の候補となる単語のリストを表す。
    sentence(S, []).             % リストの組 (S, []) でd-リストを表す。

sentence(S1, S2) :-              % 2引数の述語 sentence は
    noun_phrase(S1, L),          % 1引数の sentence と別の述語と見なされる
    verb_phrase(L, S2).

noun_phrase(Np1, Np2) :-
    proper_noun(Np1, Np2).
noun_phrase(Np1, Np2) :-
    definite_article(Np1, L),
    noun(L, Np2).

verb_phrase(Vp1, Vp2):-
    intransitive_verb(Vp1, Vp2).
verb_phrase(Vp1, Vp2) :-
    transitive_verb(Vp1, L),
    noun_phrase(L, Vp2).

/***** 辞書 *****/
intransitive_verb([dreams | L], L).
transitive_verb([eats | L], L).
proper_noun(['John' | L], L).
definite_article([the | L], L).
noun([cake | L], L).
```

図6.18　構文解析をするプログラム

```
<user> ?-  sentence(['John', eats, the, cake]).

yes

<user> ?-  sentence(['John', eats, the, apple]).

no

<user> ?-  sentence(S).

S = [John,dreams]  ;

S = [John,eats,John]  ;

S = [John,eats,the,cake]  ;

S = [the,cake,dreams]  ;

S = [the,cake,eats,John]  ;

S = [the,cake,eats,the,cake]  ;

no
```

図6.19　“構文解析をするプログラム”の実行例



```
:- lc.                              % ファイル読み込みの際の環境設定

/*********************************************************
* 例6.7(補足) append述語を用いた構文解析(?)           *
*       --- 構造を明示しない6つの文字列を正しい英文と見る *
*           ための文法知識をappend述語を用いて記述する。   *
*********************************************************/
sentence(S) :-                      % S は英文の候補となる単語のリストを表す。
     noun_phrase(Np),
     verb_phrase(Vp),
     append(Np, Vp, S).

noun_phrase(Np) :-
     proper_noun(Np).
noun_phrase(Np) :-
     definite_article(Def_art),
     noun(Noun),
     append(Def_art, Noun, Np).

verb_phrase(Vp) :-
     intransitive_verb(Vp).
verb_phrase(Vp) :-
     transitive_verb(Vt),
     noun_phrase(Np),
     append(Vt, Np, Vp).

/***** 辞書 *****/
intransitive_verb([dreams]).
transitive_verb([eats]).
proper_noun(['John']).
definite_article([the]).
noun([cake]).

/***** append述語 *****/
append([], List, List).
append([Head | Tail1], List2, [Head | Tail3]) :-
     append(Tail1, List2, Tail3).
```

図6.20 append述語を用いて構文解析らしきことをするプログラム

6.2 実行制御のための述語

4.3節や5.4節で見た様に、Prologのプログラムは言語処理系に備わった試行錯誤の機能を基に実行される。プログラム実行途中の時点では、Prolog言語処理系は未処理の（すなわち演繹可能性をまだ調べていない）ゴールを1個以上持ち、以後の実行段階でこれらのゴールを系統的に試行錯誤的に調べる。 より詳しく言うと、各時点で、Prolog言語処理系は未処理のゴールのうち最も左の（すなわち最も古い）ゴールに着目し、このゴールに適用可能な（すなわちこのゴールと単一化可能な頭部を持つ）ホーン節を探す。そして、これらの中から適用可能なホーン節を1個選んで“このホーン節の頭部とゴールの単一化に伴う代入”をホーン節と未処理のゴール列に施し、得られたホーン節の本体部で着目している未処理ゴールを置き換える。 結局、Prolog言語処理系は、適用可能なホーン節の中から実際にどれを選んで適用するかで次の動作に関して様々な可能性を持ち、これらの可能性に関して系統的に試行錯誤することになる。

場合によっては、複数の適用可能なホーン節のうち有効な（すなわち以後の実行が成功に至る）ものが1個だけに決まることもある。しかし、これまでの考え方に基づくと、Prolog言語処理系は、既に有効なホーン節を見つけてそれに基づいて解を探索したとしても、別のホーン節を使って解を探索する可能性を残したままである。そこで、プログラム実行に伴うこの様な非効率性を除くために、Prologには**カット**(cut)と呼ばれる特殊な組込み述語 ! が用意されている。カット述語 ! はその実行により不要な試行錯誤を処理系に知らせる働きを持ち、この情報に基づいてProlog言語処理系は不要な試行錯誤の実行を抑制することになる。

また、Prologのプログラムを作成する時に、Pascal,FORTRAN,C を始めとする“通常の手続き型プログラミング言語”の場合の様に、処理の繰り返しや条件分岐をはっきりと記述したくなることもある。このため、DEC-10 Prolog を始めとする多くのProlog言語処理系には、処理手順／制御構造をはっきりと表すための述語がいくつか用意されている。

それでは、カット述語 ! ,処理の繰り返しのための述語 repeat ,条件分岐のための述語 α->β ;γ と α->β を始め、実行制御のための述語としてどんなものがProlog言語処理系に組み込まれているかを紹介しよう。次の通り。

fail. 常に失敗する0引数述語であり、例5.10の様に“試行錯誤による別解の探索”を強制的に引き起こす際によく用いられる。

true.　最初の１度だけ成功する０引数述語であり、

```
true.
```

というホーン節によっても定義できる。

カット述語 !.　最初の１度だけ成功する０引数述語であり、副作用として

このカット述語を含むホーン節の中で、カット述語の左側にあるゴール（すなわちホーン節の頭部と単一化したゴール，および本体部のゴールのうち！の左側に現れるもの）についての試行錯誤が全て不要になった

ことをProlog言語処理系に知らせる働きがある。従って、！の右側の実行が失敗に終われば、即座に、このカット述語を含むホーン節を呼び出したゴールの実行が失敗となる。

call(ゴール列).　原子式 call(α) は、αをゴール,またはゴールの並びと見なしてProlog言語処理系に実行させ、その結果を元のゴール call(α) の実行結果とさせる働きを持つ。［もちろん、αをゴールの並びと見なせない場合はエラーとなる。また、引数として２個以上のゴールの並びを指定したい場合は、ゴール列を丸括弧“(”と“)”で囲まなければならない。特にACOS上のPrologでは、α内にカット述語が含まれていても、このカット述語は call(α) の左側のゴールの試行錯誤を抑制しない。］

否定述語 not(ゴール列).　原子式 not(α) はαの否定を表す。ゴール not(α) を実行するためには、Prolog言語処理系はαをゴール,またはゴールの並びと見なして実行し、その結果を反転して元のゴール not(α) の実行結果とする。すなわち、αの実行が成功すれば not(α) の実行は失敗し、αの実行が失敗すれば not(α) の実行は成功することになる。従って、カット述語等を用いて、この否定述語 not を

```
not(P) :- call(P), !, fail.
not(P).
```

と定義することもできる。［もちろん、αをゴールの並びと見なせない場合はエラーとなる。また、引数として２個以上のゴールの並びを指定したい場合は、ゴール列を丸括弧“(”と“)”で囲まなければならない。］

条件分岐述語 ゴール列 -> ゴール列 ; ゴール列.　原子式 α->β;γ は手続き型プログラミング言語の

if α then β else γ

という制御構造に相当する。ゴール α->β;γ を実行するためには、Prolog言語処理系はαをゴール（の並び）と見なして実行し、これが成功すればβを,失敗すればγを

次に実行し、その結果を元のゴール α->β;γ の実行結果とする。ゴール α->β;γ を再実行する際は、処理系はβまたはγを再実行するだけでαを再実行することはない。従って、カット述語等を用いて、この条件分岐述語を

```
(P -> Q ; R) :- call(P), !, call(Q).
(P -> Q ; R) :- !, call(R).
```

と定義することもできる。［この条件分岐述語は -> と ; という2個のアトムを組み合わせて構成され、正確には α->β;γ は ;(->(α,β),γ) という構造体の略記である。また、ゴール α->β;γ と他のゴールをコンマで区切って並べたい場合は、丸括弧で囲って（α->β;γ）と書かないと文法的に別に解釈されてしまう。］

条件分岐述語 *ゴール列 -> ゴール列*. 原子式 α->β は手続き型プログラミング言語の

if α then β

という制御構造に相当し、α->β;fail と等価である。ゴール α->β を実行するためには、Prolog言語処理系はαをゴール（の並び）と見なして実行し、これが成功すればβを次に実行し、その結果を元のゴール α->β の実行結果とする。また、αの実行が失敗に終われば α->β の実行も失敗とする。ゴール α->β を再実行する際は、処理系はβを再実行するだけでαを再実行することはない。従って、カット述語等を用いて、この条件分岐述語を

```
(P -> Q) :- call(P), !, call(Q).
```

と定義することもできる。［正確には α->β は ->(α,β) という構造体の略記である。また、ゴール α->β と他のゴールをコンマで区切って並べたい場合は、丸括弧で囲って（α->β）と書かないと文法的に別に解釈されてしまう。］

繰り返しのための述語 repeat. 実行,再実行の度に無条件に成功する0引数述語であり、同一の処理を様々の入力データに対して繰り返し行う際によく用いられる。処理の繰り返しのための述語 repeat を

```
repeat.
repeat :- repeat.
```

と定義することもできる。

次に、実行制御のためのこれらの述語の使用例を挙げてゆこう。まず手始めに、カット述語,否定述語,repeat述語の実行効果をより明確／正確に理解するために、これらの述語の実行例をそれぞれ考えてみよう。

例6．8(カット述語の効果)　カット述語の効果を示すために２つの類似したプログラム

プログラム (a)	プログラム (b)
a :- b, c.	a :- b, !, c.
a :- d.	a :- d.
b.	b.
b :- d.	b :- d.
d.	d.

に質問 ?- a. を与えて実行過程をトレースモードの下で追跡･表示させると、それぞれ図6．21（ａ），（ｂ）の様な結果が得られる。これらの結果を比較して２つの実行過程の違いをより明確に示すために、4．3節の図4．1～3に倣ってこれらの“証明図探索”の様子を図示すると、それぞれ図6．22（ａ），（ｂ）の様になる。　［２つのプログラムは、最初の規則中にカット述語が含まれるかどうかの違いを除けば、全く同一である。しかし、この違いのために、プログラム（ｂ）では、２つの規則 a:-d. と b:-d. が質問 ?-a. に対して何の働きもしなくなる。］

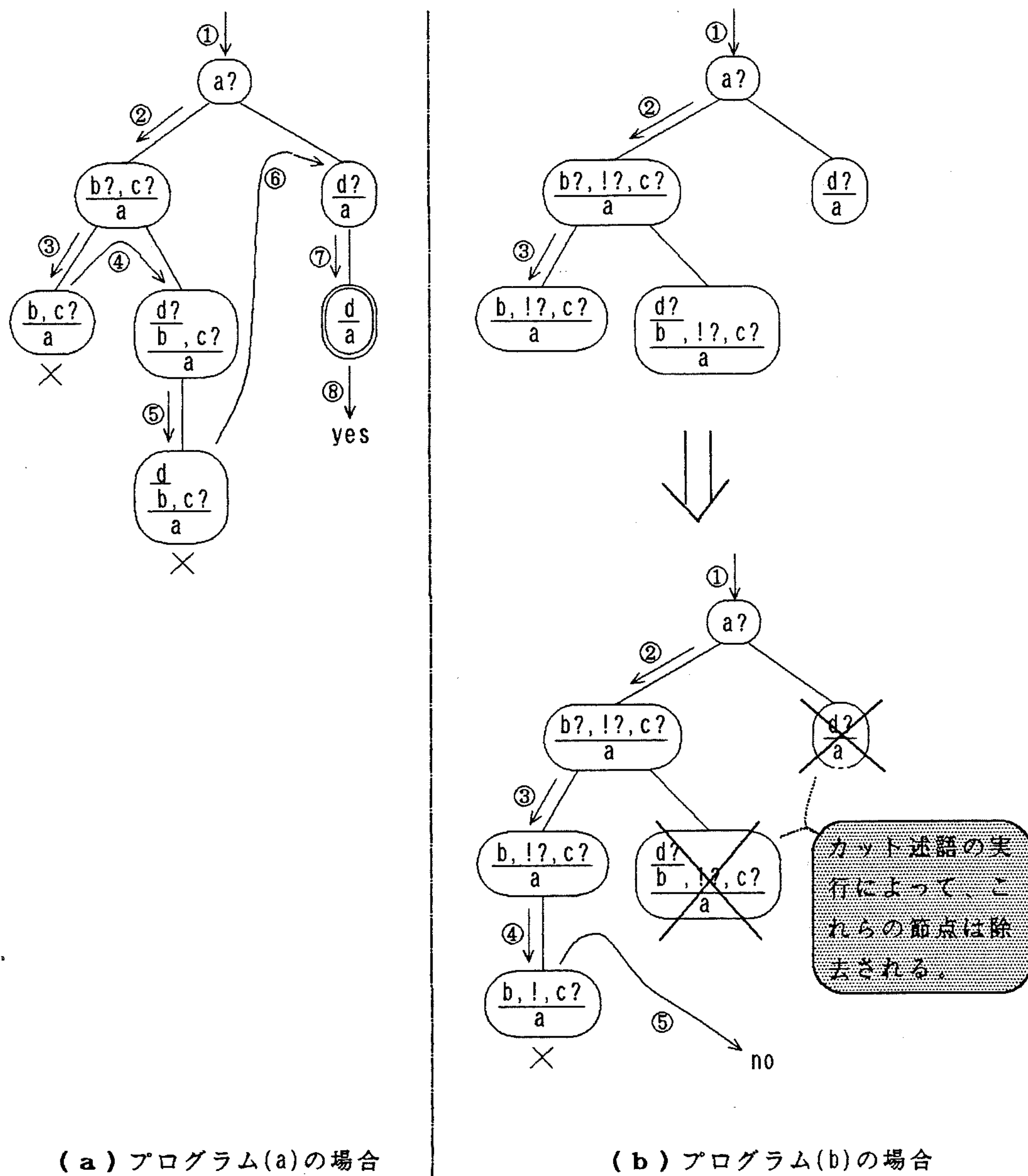
```
<user> ?-  a.
[0]   Call:  a  ?   c
[1]   Call:  b  ?   c
[1]   Exit:  b
[1]   Call:  c  ?   c
c is undefined predicate.
[1]   Fail:  c
[1]   Redo:  b  ?   c
[2]   Call:  d  ?   c
[2]   Exit:  d
[1]   Exit:  b
[1]   Call:  c  ?   c
c is undefined predicate.
[1]   Fail:  c
[1]   Redo:  b  ?   c
[2]   Redo:  d  ?   c
[2]   Fail:  d
[1]   Fail:  b
[1]   Call:  d  ?   c
[1]   Exit:  d
[0]   Exit:  a

yes
```

（ａ）プログラム(a)の場合

```
<user> ?-  a.
[0]   Call:  a  ?   c
[1]   Call:  b  ?   c
[1]   Exit:  b
[1]   Call:  !  ?   c
[1]   Exit:  !
[1]   Call:  c  ?   c
c is undefined predicate.
[1]   Fail:  c
[1]   Redo:  !  ?   c
[0]   Fail:  a

no
```

（ｂ）プログラム(b)の場合

図6．21　質問 ?-a. に対する実行過程の追跡・表示



(a) プログラム(a)の場合

(b) プログラム(b)の場合

図6.22　質問 ?-a. に対する探索の様子

例6.9(否定述語の正しい使い方)　大ざっぱな言い方をすれば、組込み述語 not は論理結合子の否定に相当する働きをする。しかし、not述語の引数に変項が含まれる場合は注意が必要で、例えば ?-not(on(dog,X)). という質問は“原子式 on(dog,X) を満たさない X の探索”を引き起こす訳ではない。なぜなら、先に記述した様にゴール not(on(dog,X)) を実行するためには、Prolog言語処理系はまずゴール on(dog,X) を実行し、これが成功すれば,すなわち on(dog,X) を満たす X が見つかれば元のゴール not(on(dog,X)) を失敗で終える。そして、on(dog,X) を満たす X が見つからずにゴール on(dog,X) の実行が失敗すれば、元のゴール not(on(dog,X)) を失敗で終える。結局、質問 ?-not(on(dog,X)). は“原子式 on(dog,X) を満たす X は存在しないか？”という意味を持つのである。従って、プログラム

```
on(cock, cat).
on(cat,  dog).
on(dog,  donkey).
```

に対して、質問 ?-not(on(dog,X)). を行うと no という結果が表示され、2つの類似した質問

```
?- on(X,cat), not(on(dog,X)).
?- not(on(dog,X)), on(X,cat).
```

を行うと全く異なる結果が得られる。すなわち、図6.23に示される様な実行例が得られる。

```
<user> ?-  not(on(dog,donkey)).
no
<user> ?-  not(on(dog,cock)).
yes
<user> ?-  not(on(dog,X)).
no
<user> ?-  on(X,cat), not(on(dog,X)).
X = cock  ;
no
<user> ?-  not(on(dog,X)), on(X,cat).
no
```

図6.23　否定述語の使用例

例６．１０(repeat述語による繰り返しの原理) 組込み述語 repeat は、同一の処理を様々の入力に対して繰り返し実行する際に有効である。例えば、例５．１０で考えたオーム返しの処理を様々の入力に対して繰り返し実行させるには、例５．１０のオーム返しプログラムを図６．２４の様に改造すればよい。このプログラムの実行例は図６．２５の通りであり、プログラムは halt というアトムを入力として受け取ると繰り返しを終了する。repeat述語の動作をより明確に理解するために、この図６．２５の中の２番目の実行例の“証明図探索”の様子を４．３節の図４．１～３に倣って図示すると、図６．２６の様になる。

```
:- lc.                              % ファイル読み込みの際の環境設定

/*****************************************************
* 例6.10  repeat述語によるオーム返し処理の繰り返し *
*****************************************************/
echo :-
     repeat,
     read(T),
     write(T), nl,
     T = halt.
```

図６．２４ オーム返しを繰り返すプログラム

```
<user> ?-  echo.
 >  one.
one
>  two.
two
>  three.
three
>  halt.
halt

yes

<user> ?-  echo.
>  a.
a
>  halt.
halt

yes
```

図６．２５ “オーム返しを繰り返すプログラム”の実行例

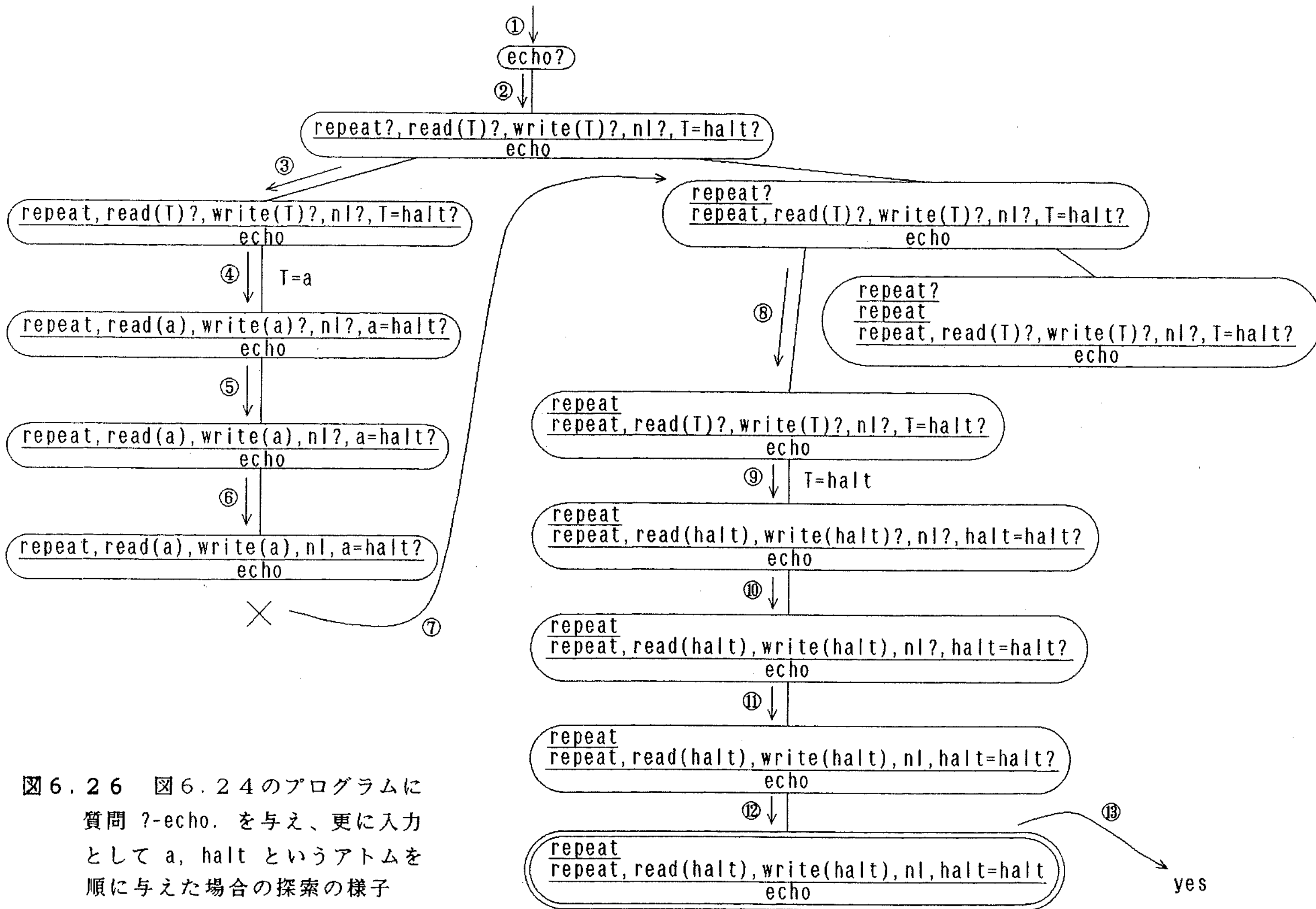


図6.26 図6.24のプログラムに質問 ?-echo. を与え、更に入力として a, halt というアトムを順に与えた場合の探索の様子

```
<user> ?-  sentence(S), write(S), nl, fail.
[John,dreams]
[John,eats,John]
[John,eats,the,cake]
[the,cake,dreams]
[the,cake,eats,John]
[the,cake,eats,the,cake]

no
```

図６．２７　図６．１８のプログラムを用いて全解自動生成する質問例

実行制御のための述語を用いれば、全解の自動生成を行ったり、実行効率の良いPrologプログラムを綺麗に作り上げたりするのが容易になる。例えば次の通り。

例６．１１(全解の自動生成)　組込み述語 fail を用いて全ての解を自動的に生成する方法は、４-Ｑｕｅｅｎｓ問題を用いて例５．１４で例示した通りである。例５．１４ではプログラムの中に全解自動生成の機構を組み込んだが、この機構を質問の中に組み込むこともできる。例えば、前節の例６．７のプログラム（図６．１８）をそのまま用いて全解自動生成するには、図６．２７の様な質問をすればよい。

例６．１２(カット述語を用いた場合分け;ＪＲ普通運賃の計算)　カット述語を用いると場合分けをすっきりとした形で行えることがある。例えば、１９９２年２月現在、ＪＲ幹線の（税抜きの）普通運賃は次の６つの規則に従って決まる。

① 運賃計算をする時、利用区間の営業キロ数の小数部を切り上げてから計算する。

② 税抜きの普通運賃は、利用区間の営業キロ数が３ｋｍ以下なら１４０円，４～６ｋｍなら１７０円，７～１０ｋｍなら１８０円である。

③ 営業キロ数は１１～５０ｋｍの間は５ｋｍ刻みで，５１～１００ｋｍの間は１０ｋｍ刻みで，１０１～６００ｋｍの間は２０ｋｍ刻みで，６０１ｋｍ以上の間は４０ｋｍ刻みで区分し、それぞれの区分内で運賃を同じにする。　従って、営業キロ数は１１～１５ｋｍ，１６～２０ｋｍ，５１～６０ｋｍ，...，１０１～１２０ｋｍ，... と分類され、それぞれの区分に対して税抜きの普通運賃が決められる。

④ 利用区間の営業キロ数が１１ｋｍ以上の場合、大ざっぱには、利用区間の最初の３００ｋｍまでは１ｋｍ当たり１６円２０銭，３００～６００ｋｍの区間は１ｋｍ当たり１２円８５銭，６００ｋｍを越えた区間は１ｋｍ当たり７円５銭の運賃率で、税抜きの普通運賃を

計算する。

⑤ ③のそれぞれの区分に対して税抜きの普通運賃を計算するためには、まず、それぞれの区分の中間点の営業キロ数に対して④の方法で“大ざっぱな運賃”を計算する。［但し、例えば１１～１５ｋｍの区分の中間点は１３ｋｍ，５１～６０ｋｍの区分の中間点は５５ｋｍとする。他も同様。］　そして、営業キロ数が１００ｋｍ以下の場合は“大ざっぱな運賃”を切り上げで１０円単位の金額に修正し、営業キロ数が１０１ｋｍ以上の場合は“大ざっぱな運賃”を四捨五入で１００円単位の金額に修正することによって税抜きの運賃を決定する。

これらの規則に基づけば、実際にはＪＲ幹線の（税抜きの）普通運賃は、利用する区間の営業キロ数を x，小数点以下の切り上げ関数を ceiling，切り捨て関数を floor で表して、

$x \leq 3$ なら 140 円，

$3 < x \leq 6$ なら 170 円，

$6 < x \leq 10$ なら 180 円，

$10 < x \leq 50$ なら ceiling({floor((ceiling(x)－1)／5)×5＋3}×1.62)×10 円，

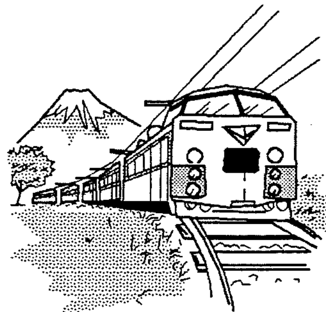
$50 < x \leq 100$ なら ceiling({floor((ceiling(x)－1)／10)×10＋5}×1.62)×10 円，

$100 < x \leq 300$ なら ceiling({floor((ceiling(x)－1)／20)×20＋10}×0.162＋0.5)×100 円，

$300 < x \leq 600$ なら ceiling(300×0.162＋
{floor((ceiling(x)－301)／20)×20＋10}×0.1285＋0.5)×100 円，

$600 < x$ なら ceiling(300×0.162＋300×0.1285＋
{floor((ceiling(x)－601)／40)×40＋20}×0.0705＋0.5)×100 円

と計算できる。また、消費税込みの普通運賃は、税抜きの普通運賃に１．０３を乗じ四捨五入で１０円単位の金額に修正することによって計算できる。　従って、これらの事実を基にすれば、利用する区間の営業キロ数から普通運賃を計算するPrologプログラムとして例えば図６．２８のものが得られる。このプログラムの実行例は図６．２９の通りである。



```
:- lc.                              % ファイル読み込みの際の環境設定

/*******************************
* 例６．１２ ＪＲ普通運賃の計算 *
*******************************/
fare_table(Distance, Fare) :- not((number(Distance), Distance > 0.0)), !,
    write("述語 fare_table の第一引数は正数でなければならない。"),
    fail.
fare_table(Distance, Fare) :-
    fare_table0(Distance, Fare_before_tax),
    Fare is (Fare_before_tax*103+500)//1000*10.

fare_table0(Distance, Fare_before_tax) :- Distance =< 3.0, !,
    Fare_before_tax is 140.
fare_table0(Distance, Fare_before_tax) :- Distance =< 6.0, !,
    Fare_before_tax is 170.
fare_table0(Distance, Fare_before_tax) :- Distance =< 10.0, !,
    Fare_before_tax is 180.
fare_table0(Distance, Fare_before_tax) :- Distance =< 50.0, !,
    Fare_before_tax is ((ceiling(Distance)-1)//5*810+585)//100*10.
fare_table0(Distance, Fare_before_tax) :- Distance =< 100.0, !,
    Fare_before_tax is ((ceiling(Distance)-1)//10*162+90)//10*10.
fare_table0(Distance, Fare_before_tax) :- Distance =< 300.0, !,
    Fare_before_tax is ((ceiling(Distance)-1)//20*324+212)//100*100.
fare_table0(Distance, Fare_before_tax) :- Distance =< 600.0, !,
    Fare_before_tax is ((ceiling(Distance)-301)//20*2570+50385)//1000*100.
fare_table0(Distance, Fare_before_tax) :-
    Fare_before_tax is ((ceiling(Distance)-601)//40*282+8906)//100*100.
```

図６．２８ ＪＲ普通運賃を計算するプログラム

```
<user> ?-  fare_table(14.9,Uchino_Niigata), fare_table(538.4,Niigata_Kyoto),
>        X is 14.9+538.4, fare_table(X,Uchino_Kyoto).

Uchino_Niigata = 230
Niigata_Kyoto = 8030
X = 553.300
Uchino_Kyoto = 8340  ;

no
```

図６．２９ “ＪＲ普通運賃を計算するプログラム”の実行例

例6.13(カット述語,条件分岐述語を用いたプログラムの簡素化;素数の生成)　カット述語や条件分岐述語を用いると、プログラム，特に**決定的**(deterministic;すなわち実行時にゴールに適用可能なホーン節が常に1個以下になるよう)なプログラムを、実行効率がより良くしかもより簡潔な形のものに書き換えれることがある。　例えば、例6.5の素数生成プログラムは、カット述語と α->β;γ という形の条件分岐述語を用いて図6.30の様に簡素化できる。

```
:- lc.                                % ファイル読み込みの際の環境設定

/*********************************************************************
 * 例6.13　カット述語,条件分岐述語による素数生成プログラムの簡素化 *
 *********************************************************************/
generate_primes(N, List_of_primes) :-
     generate_integers(2, N, List_of_integers),  % List_of_integers =
     sift(List_of_integers, List_of_primes).     %   [2,3,4, ... ,N]
                                                 % となる。

generate_integers(M, N, [M | Integers]) :- M =< N, !, % Integers =
     M1 is M+1,                                        %  [M+1, ... ,N]
     generate_integers(M1, N, Integers).               % となる見込み。
generate_integers(M, N, []).

sift([P | Integers], [P | Primes]) :-           % この時点では P は素数。
     filter(P, Integers, New_integers),         % New_integers =
     sift(New_integers, Primes).                %   Integersから
sift([], []).                                   %   P の倍数を除去したもの
                                                % となることが期待される。

filter(P, [K | Integers], New_integers) :-
     K mod P =:= 0 -> filter(P, Integers, New_integers)
                  ;(New_integers = [K | Rest],
                    filter(P, Integers, Rest)).
filter(P, [], []).
```

図6.30　カット述語,条件分岐述語を使って図6.12のプログラムを簡素化した例

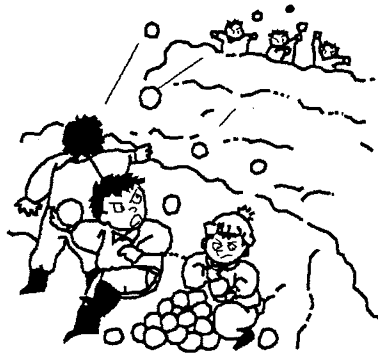
例6．14（repeat述語，条件分岐述語の組み合わせによる処理の繰り返し;英-和の間の単語翻訳）
例6．10で述べた様に、repeat述語は同一の処理を（様々な入力に対して）繰り返し実行する際に有効である。そして、このrepeat述語と α->β;γ という形の条件分岐述語を組み合わせると、手続き型プログラミング言語の **while-do** に類似した制御構造を作り出すことができる。 例えば、例5．11で考えた英-和の間の単語翻訳処理を様々な入力に対して繰り返す場合は、repeat述語で繰り返し制御を，条件分岐述語で繰り返しの終了制御を行うことにして、図5．15のプログラムを図6．31の様に改造すればよい。図6．32の実行例に見られる様に、このプログラムは halt というアトムを入力として受け取ると繰り返しを終了する。

```
:- lc.                                   % ファイル読み込みの際の環境設定

/**************************************************************************
* 例6．14　repeat述語,条件分岐述語による英-和間の単語変換処理の繰り返し*
**************************************************************************/
english_japanese(mouse,     鼠).         % 辞書
english_japanese(cow,       牛).
english_japanese(tiger,     虎).
english_japanese(rabbit,    兎).
english_japanese(dragon,    竜).
english_japanese(snake,     蛇).
english_japanese(horse,     馬).
english_japanese(sheep,     羊).
english_japanese(monkey,    猿).
english_japanese(bird,      鳥).
english_japanese(dog,       犬).
english_japanese(wild_boar,猪).

look_up(Word, Equivalent) :-
     english_japanese(Word, Equivalent).
look_up(Word, Equivalent) :-
     english_japanese(Equivalent, Word).

translate :-
     repeat,
     read(Word),
     (Word = halt -> true
                    ; (look_up(Word, Equivalent),
                       write(""""), write(Word),
                       write("""という単語の訳語は"),
                       write(""""), write(Equivalent),
                       write("""です。"), nl,
                       fail                          )).
```

図6．31　英-和間の単語翻訳を繰り返すプログラム

```
<user> ?-  translate.
 >  dog.
"dog"という単語の訳語は"犬"です。
>  兎.
"兎"という単語の訳語は"rabbit"です。
>  cat.
>  halt.

yes
```

図6.32　"英-和間の単語翻訳を繰り返すプログラム"の実行例



6．3 知識ベース操作のための述語

4．3〜4節で見た様に、Prologプログラムを実行させるには、まずProlog言語処理系を呼び出してその中にプログラムを読み込ませ、次に処理系に質問を与える。これらの操作を受けて、Prolog言語処理系の方では、読み込んだプログラムを公理系と見なしてこれを基に知識ベースを構築し、与えられた質問項目が公理系から導けるかどうかを調べることになる。

ここで、Prolog言語処理系の中に構築された知識ベースは、普通はプログラムの実行によって書き換えられることはない。しかし、Prolog言語処理系には、指定されたホーン節を知識ベースに追加したり知識ベースから削除したりするための述語も組み込まれており、これらの組込み述語を用いればプログラムの実行によって知識ベースを書き換えることも可能である。プログラマの目から見れば、

自分自身を修正しながら実行するプログラムを書ける

という、他のプログラミング言語には不可能で奇妙なことができるわけであり、それゆえ、知識ベース操作のためのこれらの組込み述語はプログラマに様々な可能性を与えてくれる。［但し、知識ベース操作の述語は使い方を誤るとデバッグにも困ることになるので注意が必要である。特に、プログラム実行時の規則の追加／削除はプログラムの複雑さの元凶となり得るので、これらの操作は必要最小限に止めなければならない。］

それでは、知識ベース操作のための述語としてどんなものがProlog言語処理系に組み込まれているかを紹介しよう。基本的なものは次の通り。

assert(ホーン節). 最初の１度だけ成功する１引数述語であり、副作用として

引数をホーン節と見なし処理系内の知識ベースに追加登録する

働きがある。 ［ここで、引数として規則を指定する場合は規則を丸括弧“(”と“)”で囲まなければ文法エラーとなる。また、特に ACOS 上の Prolog の場合には、assert述語は引数のホーン節を知識ベースの最後に追加登録する。］

asserta(ホーン節). assert述語とほぼ同じ。但し、引数として指定したホーン節は知識ベースの最初に追加登録される。

assertz(ホーン節). assert述語とほぼ同じ。但し、引数として指定したホーン節は知識ベースの最後に追加登録される。

clause(頭部,本体).　Prolog言語処理系内では、一般に α. という形の事実は α:- true. という規則の略記と見なされる。この様な取扱いの下で、原子式 clause(α,β) は、

α:-β. という形の確定節が知識ベースに登録されている

という命題を表す。[ここで、clauseの第一引数が原子式でない場合はエラーとなる。また、第二引数としてゴールの並びを指定する場合はそれらを丸括弧 “(” と “)” で囲まなければならない。]　従って、プログラムの実行中にゴール clause(α,β) が起動されると、その演繹可能性を調べるためにProlog言語処理系は、α:-β. というパターンと単一化可能な確定節を知識ベースの中から探し出そうとする。その結果、

① もしその様な確定節が見つかればゴール clause(α,β) は成功する。そして、新たなゴールがあれば、それらは単一化に伴う代入の下で起動される。

② もしその様な確定節が見つからなければ clause(α,β) は失敗する。

もちろん、再実行の際には、Prolog言語処理系は α:-β. というパターンと単一化可能な確定節を新たに知識ベースの中から探し出そうとする。

abolish(アトム,整数).　最初の１度だけ成功する２引数述語であり、副作用として

第一引数で指定された述語名，第二引数で指定されたパラメータ数を頭部に持つホーン節を全て処理系内の知識ベースから削除する

働きがある。

retract(ホーン節).　原子式 retract(α) は

α. という形の確定節が知識ベースに登録されている

という命題を表し、（実行が成功に終われば）副作用として

α. という形の（最初の）確定節を１個だけ知識ベースから削除する

働きがある。[ここで、引数として規則を指定する場合は規則を丸括弧 “(” と “)” で囲まなければ文法エラーとなる。]　従って、プログラムの実行中にゴール retract(α)が起動されると、その演繹可能性を調べるためにProlog言語処理系は、α. というパターンと単一化可能な確定節を知識ベースの中から探し出そうとする。その結果、

① もしその様な確定節が見つかれば、ゴール retract(α) は成功し副作用としてその確定節が知識ベースから削除される。そして、新たなゴールがあれば、それらは単一化に伴う代入の下で起動される。

② もしその様な確定節が見つからなければ retract(α) は失敗する。

もちろん、再実行の際には、Prolog言語処理系は α. というパターンと単一化可能な確定節を新たに知識ベースの中から探し出して削除しようとする。

補足6.15(assert(p:-q) という記述が文法エラーとなる原因) Prolog言語処理系内では、ホーン節の頭部と本体を区切る文字列“:-”，ゴールとゴールを区切ってそれらの連言を表す文字“,”（，ゴールとゴールを区切ってそれらの選言を表す文字“;”），なども**演算子**(operator,**オペレータ**)と見なされ、ホーン節から（最後尾の）ピリオドを除去して得られる文字列も構造体（従って項）として扱われる。算術式内の括弧を省略するために算術演算子(+,-,*,/,//,mod)間の結合の優先順位は5.2節の通りに決められているが、Prolog言語処理系内では算術演算子だけでなく論理演算子(¥,/¥,¥/,@,<<,>>)や“:-”，“?-”，“,”，“;”なども含めて演算子間の優先順位が決められている。 そして、この優先順位に関連して、引数と引数を区切るためのコンマ“,”と演算子のコンマ“,”の間の混同を避けるために、

構造体を構成する引数の結合の強さを演算子“,”による結合よりも強くする、

という文法規則が設けられ、これに違反しないために

引数の最も外側の演算子が“:-”，“?-”，“;”，“->”または“,”である場合にはその引数を丸括弧“(”と“)”で囲まなければならなくなる

のである。

補足6.16 assert,clause,retract や前節6.2の call,not などの述語は、引数として原子式,原子式の並び,またはホーン節を取るので、**高階**(higher-order)の述語と呼ばれる。これに対して、is や read などの様に引数として（定義4.5で定めた狭義の）項しか取らない述語は、**一階**(first-order)の述語と呼ばれる。

知識ベース操作の述語を用いると、（通常のプログラミング言語の場合の様に）データの記憶／更新をはっきりとした形で行ったり、また、（場合によっては）プログラムの基本構造を変えずにプログラムの実行効率化を図ったりできる様になる。例えば次の通り。

例６．１７(疑似乱数)　計算機を用いて（疑似）乱数を次々に発生する方法としては、**線形合同式法**(linear congruential method,**線形合同法**)，すなわち線形の漸化式

$$x_n=(a\times x_{n-1}+c) \bmod m \qquad (n\geqq 1)$$

但し、a, mは正の整定数，cは非負の整定数

によって定まる数列｛x_n｝を（区間［０，$m-1$］から整数を無作為に選んで並べた）乱数列と見なす方法がよく知られている。例えば、漸化式

$$(1)\ \begin{cases} x_0=0 \\ x_n=(3x_{n-1}+1) \bmod 10 \qquad (n\geqq 1) \end{cases}$$

により周期４の疑似乱数列　０→１→４→３→０→１→４→３→……　が定義され、漸化式

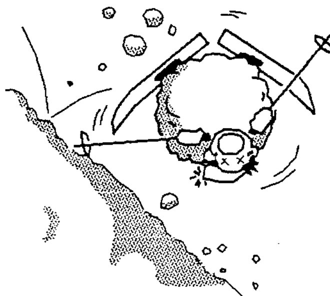
$$(2)\ \begin{cases} x_0=293 \\ x_n=(30725x_{n-1}+1) \bmod 2^{20} \qquad (n\geqq 1) \end{cases}$$

により周期　2^{20}　の疑似乱数列が定義される。［但し、この(2)の疑似乱数列では偶数と奇数が交互に現れてしまう。］

それでは、線形合同式法により疑似乱数列を発生する機能をPrologでどう実現するかを考えよう。例えば、(1)で定まる疑似乱数列の項を実行の度に１個ずつ生成／出力する述語 random が欲しいというのであれば、それを図６．３３の様に定義して図６．３４の様に用いることもできる。　また、(2)で定まる数列を基にして、

区間［1,R］から整数を（一見したところ）無作為に選んで変項 X に割り当てる

述語 random2(R,X) を構成したいのであれば、それを図６．３５の様に定義して図６．３６の様に用いることもできる。　［疑似乱数列の項を実行の度に１個ずつ生成／出力する述語を定義したいというのであれば、図６．３３,図６．３５の様に知識ベース操作の述語が必要になる。しかし、疑似乱数列を（永久に）出力し続けるプログラムが欲しいというのであれば、知識ベース操作の述語を用いなくても単に再帰法により容易にプログラムを記述できる。念のため。］



```
:- lc.                                % ファイル読み込みの際の環境設定

/*************************************************
* 例６．１７(その１)　線形合同式法による疑似乱数の生成 *
*        ---  x(0)=0                             *
*             x(n)=3*x(n-1)+1 mod 10             *
*************************************************/
val(x, 0).

random :-
    retract(val(x, Y)),
    write(Y),
    New_Y is (3*Y+1) mod 10,
    assert(val(x, New_Y)).
```

図６．３３　疑似乱数を発生させるプログラム

```
<user> ?-  random.
0
yes

<user> ?-  random.
1
yes

<user> ?-  random.
4
yes

<user> ?-  random.
3
yes

<user> ?-  random.
0
yes

<user> ?-  random.
1
yes
```

図６．３４　疑似乱数発生プログラム（図６．３３）の実行例

```
:- lc.                                   % ファイル読み込みの際の環境設定

/*****************************************************************
* 例６．１７(その２)　線形合同式法による疑似乱数の生成                 *
*       --- 漸化式 x(0)=293, x(n)=30725*x(n-1)+1 mod 1048576 によって *
*           定まる疑似乱数列 {x(n)} と任意に与えられた正整数Ｒを基に、*
*           区間［１，Ｒ］内の整数を（一見したところ）無作為に選ぶ。   *
*****************************************************************/
seed(293).

random2(R, X) :-
    retract(seed(S)),
    X is (S mod (2*R))//2+1,           % X is (S mod R)+1 とすると、
    New_S is (30725*S+1) mod 1048576,  % Rが偶数の時、偶数と奇数が
    assert(seed(New_S)),               % 交互に選ばれるという規則性
    !.                                 % が生じてしまう。
```

図６．３５　指定範囲の疑似乱数を発生させるプログラム

```
<user> ?-  repeat, random2(10, X).

X = 7  ;

X = 10  ;

X = 6  ;

X = 7  ;

X = 9  ;

X = 8  ;

X = 10  ;

X = 5  ;

X = 3  ;

X = 2

yes
```

図６．３６　疑似乱数発生プログラム（図６．３５）の実行例

例６．１８(平均値の計算)　膨大な事実を記憶／加工するためにProlog言語処理系を用いることもできる。例えば、例５．３のプログラムは各国の面積や人口を記憶するためのデータベースとしての役割も果たしている。また、

	身長	体重	年齢	
北勝海	１８１ｃｍ	１５１ｋｇ	２８歳	（１９９２年春場所直前，小結以上）
小錦	１８７	２６２	２８	
霧島	１８７	１２７	３２	
曙	２０４	１９７	２２	
貴花田	１８６	１２７	１９	
栃乃和歌	１９１	１５６	２９	
若花田	１８０	１２５	２１	
水戸泉	１９５	１８６	２９	

という事実から成るデータベースは例えば図６．３７のPrologプログラムで表すことができる。

これらのデータベースの各項目についての平均を計算するプログラムは、assert や retract述語を用いて例えば図６．３８の様に構成できる。　そして、このプログラムを上の大相撲のデータベース（ data0618 という名前のファイルに記憶されているとする）に対して適用している様子は図６．３９に示される。　［ここで、文字列 =.. はユニブ(univ)と呼ばれる述語の名前であり、通常、構造体とリストの間の変換を行う際に用いられる。より正確に言うと、原子式 =..(α，β) は

　　βは構造体αの関数（または述語）名と引数から成るリストである

という命題を表し、α=..β と略記される。例えば、f(a,b) =.. [f,a,b] である。］

```
/*******************************************************
* 例６．１８のプログラムで使うデータ．                 *
*       --- 力士の身長，体重，年齢から成るデータベース．*
*           １９９２年春場所直前（３月），小結以上．   *
*******************************************************/
height(北勝海  , 181).   weight(北勝海  , 151).   age(北勝海  , 28).
height(小錦    , 187).   weight(小錦    , 262).   age(小錦    , 28).
height(霧島    , 187).   weight(霧島    , 127).   age(霧島    , 32).
height(曙      , 204).   weight(曙      , 197).   age(曙      , 22).
height(貴花田  , 186).   weight(貴花田  , 127).   age(貴花田  , 19).
height(栃乃和歌, 191).   weight(栃乃和歌, 156).   age(栃乃和歌, 29).
height(若花田  , 180).   weight(若花田  , 125).   age(若花田  , 21).
height(水戸泉  , 195).   weight(水戸泉  , 186).   age(水戸泉  , 29).
```

図６．３７　大相撲のデータベースを表すPrologプログラム

```
:- lc.                                   % ファイル読み込みの際の環境設定

/************************************************
* 例6.18 与えられた項目についての平均値の計算 *
************************************************/
average(Key, Ave) :-
      atom(Key),
      assert(pair_of_num_and_sum(0, 0)),
      Fact =.. [Key, Name, Val],
      call(Fact),
      retract_without_backtrack(pair_of_num_and_sum(N, S)),
      New_N is N+1,
      New_S is S+Val,
      assert(pair_of_num_and_sum(New_N, New_S)),
      fail.
average(Key, Ave) :-
      retract(pair_of_num_and_sum(N, S)),
      (N =:= 0 -> Ave = undefined
                ; Ave is float(S)/float(N) ).

retract_without_backtrack(X) :-
      retract(X),
      !.
```

図6.38 与えられた項目について平均値を計算するプログラム

```
<user> ?-  consult(data0618).
data0618 consulted 611 cells 0.0959680 sec.

yes

<user> ?-  average(height, Ave_height),
>          average(weight, Ave_weight),
>          average(age,    Ave_age).

Ave_height = 188.875
Ave_weight = 166.375
Ave_age = 26.0000  ;

no
```

図6.39 平均値計算プログラム（図6.38）を
大相撲データベース（図6.37）に適用している様子

例６．１９(連想計算によるフィボナッチ数列の計算)　　漸化式

$$(3)\quad \begin{cases} a_1 = a_2 = 1 \\ a_n = a_{n-1} + a_{n-2} & (n \geqq 3) \end{cases}$$

によって決まる数列 $\{a_n\}$ をフィボナッチ数列(Fibonacci sequence)という。例えば、

$a_1 = 1$,

$a_2 = 1$,

$a_3 = a_2 + a_1 = 1 + 1 = 2$,

$a_4 = a_3 + a_2 = 2 + 1 = 3$,

$a_5 = a_4 + a_3 = 3 + 2 = 5$

である。［実は、$a_n = \{((1+\sqrt{5})/2)^n - ((1-\sqrt{5})/2)^n\}/\sqrt{5}$ である。］

正整数 n をパラメータとして受け取りそれを基にフィボナッチ数列の第 n 項を計算／出力するプログラムとして、最初に思いつくのは図６．４０の様なものであろう。このプログラムは、フィボナッチ数列の定義式(3)に忠実に計算するものであり、その実行には相当の（正確には $O(2^n)$ の，すなわち漸近的に 2^n に比例する）計算時間がかかる。例えば、a_5 を計算／出力する際の実行の追跡・表示の様子は図６．４１の通りであるが、この計算のためにこのプログラムは図６．４２の様に非効率的な手続き呼出しを行ってしまう。

```
:- lc.                                  % ファイル読み込みの際の環境設定

/****************************************************
* 例6.19(その1)　フィボナッチ数列の第n項目の計算 *
*       --- 定義式に忠実に計算する方法             *
****************************************************/
fib1(1, 1).
fib1(2, 1).
fib1(N, A) :-
     N > 2,
     N1 is N-1,    N2 is N-2,
     fib1(N1, A1), fib1(N2, A2),
     A is A1+A2.
```

図６．４０　フィボナッチ数列の第 n 項を計算するプログラム

```
<user> ?-  spy fib1.
Enter debug mode.
Spy point is set on fib1/2

yes

<user> ?-  fib1(5, A5).
[0] * Call:  fib1(5,_21)  ?  |
[1] * Call:  fib1(4,_271)  ?  |
[2] * Call:  fib1(3,_2187)  ?  |
[3] * Call:  fib1(2,_2303)  ?  |
[3] * Exit:  fib1(2,1)
[3] * Call:  fib1(1,_2311)  ?  |
[3] * Exit:  fib1(1,1)
[2] * Exit:  fib1(3,2)
[2] * Call:  fib1(2,_2195)  ?  |
[2] * Exit:  fib1(2,1)
[1] * Exit:  fib1(4,3)
[1] * Call:  fib1(3,_279)  ?  |
[2] * Call:  fib1(2,_2520)  ?  |
[2] * Exit:  fib1(2,1)
[2] * Call:  fib1(1,_2528)  ?  |
[2] * Exit:  fib1(1,1)
[1] * Exit:  fib1(3,2)
[0] * Exit:  fib1(5,5)

A5 = 5

yes
```

図6.41　図6.40のフィボナッチ数列計算プログラムの実行例

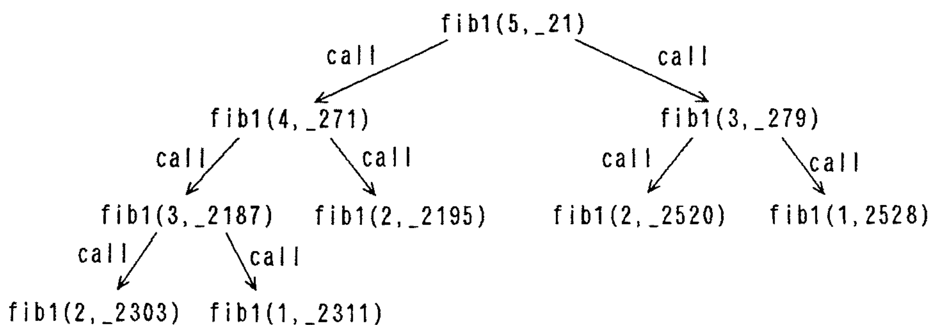
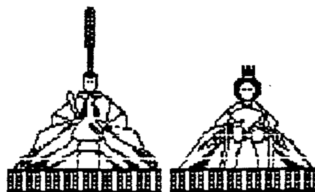


図6.42　図6.41のプログラム実行の際に起こる手続き呼出しの様子
［ここで、_21,_271,_279,... は図6.41に現れた変項である。］

そこで、プログラムの実行効率化について次に考えてみよう。図6.40のプログラムの基本構造を変えたくない場合には、例えば assertaを述語を用いてプログラムを図6.43の様に少しだけ書き換えることにより、実行効率化を図れる。実際、このプログラムは図6.40のものと同様に“フィボナッチ数列の定義式(3)に忠実に計算する”ものであるが、（図6.44の実行例からも分かる様に）その計算時間は $O(n)$に改善されている。この様な計算方式を連想計算(associative computation)という。［プログラムの基本構造を変えてもよいのであれば、図6.40のプログラムの実行効率化のために知識ベース操作の述語は不要である。例えば図6.45〜46のプログラムを考えれば、これらの計算時間は $O(n)$になる。ただ、これらのプログラムにはもはやフィボナッチ数列の定義式(3)の形が反映されていない。］

```
:- lc.                              % ファイル読み込みの際の環境設定

/***********************************************************
* 例6.19(その2)　フィボナッチ数列の第n項目の（連想）計算 *
*         --- 定義式に忠実に、しかもO(n)の手間で計算する方法 *
***********************************************************/
fib2(1, 1).
fib2(2, 1).
fib2(N, A) :-
     N > 2,
     N1 is N-1,     N2 is N-2,
     fib2(N1, A1), fib2(N2, A2),
     A is A1+A2,
     asserta((fib2(N, A) :- !)).
```

図6.43　フィボナッチ数列の第 n項を連想計算するプログラム
［下線部は図6.40と違う部分］



```
<user> ?-  spy fib2.
Enter debug mode.
Spy point is set on fib2/2

yes

<user> ?-  fib2(5, A5).
[0] * Call:  fib2(5,_21)  ?  |
[1] * Call:  fib2(4,_271)  ?  |
[2] * Call:  fib2(3,_2202)  ?  |
[3] * Call:  fib2(2,_2333)  ?  |
[3] * Exit:  fib2(2,1)
[3] * Call:  fib2(1,_2341)  ?  |
[3] * Exit:  fib2(1,1)
[2] * Exit:  fib2(3,2)
[2] * Call:  fib2(2,_2210)  ?  |
[2] * Exit:  fib2(2,1)
[1] * Exit:  fib2(4,3)
[1] * Call:  fib2(3,_279)  ?  |
[1] * Exit:  fib2(3,2)
[0] * Exit:  fib2(5,5)

A5 = 5

yes
```

図6.44 図6.43のフィボナッチ数列計算プログラムの実行例

```
:- lc.                                  % ファイル読み込みの際の環境設定

/*************************************************************
 * 例6.19(その3) フィボナッチ数列の第n項目の計算              *
 *         --- 定義式に忠実ではないが、O(n)の手間で計算する方法 *
 *************************************************************/
fib3(N, A) :-
    f(N, A1, A).             % A1は第(N-1)項目の値；Aは第N項目の値

f(1, 0, 1).
f(N, A1,A) :-
    N > 1,
    N1 is N-1,
    f(N1, A2, A1),
    A is A2+A1.
```

図6.45 フィボナッチ数列の第 n 項を $O(n)$ の手間で計算するプログラム

```
:- lc.                                 % ファイル読み込みの際の環境設定

/*****************************************************
* 例6.19(その4) フィボナッチ数列の第n項目の計算  *
*        --- 定義式に忠実ではないが、                *
*           末端再帰法によりO(n)の手間で計算する方法 *
*****************************************************/
fib4(N, A) :-
    f(N, A, 1, 0, 1).

f(N, A, N, A1, A).
f(N, A, M, B1, B) :-       % B1は第(M-1)項目の値;Bは第M項目の値
     M < N,
     M1 is M+1,
     M1th_term is B1+B,
     f(N, A, M1, B, M1th_term).
```

図6.46 フィボナッチ数列の第n項を末端再帰法で計算するプログラム

演習問題６

6．1　ACOS上のPrologには、与えられたリストの長さ（すなわち要素の個数）を計算するための述語 length があらかじめ組み込まれている。この組込み述語が内部でどの様に定義されているか考えよ。すなわち、次の様な意味の述語 length を（is述語などを用いて再帰的に）定義せよ。

length(L, n)　⇔　Lはリスト，nはLを構成する要素の個数である。

［例えば、length([2,1,3],3)，length([2,2,2],3)，length([[2,2],2],2)は全て真である。］

6．2　リストに関する次の２つの述語をProlog上で定義せよ。

(1)　member(a, L)　⇔　Lはリスト，aはLの要素である。

(2)　subset(L_1, L_2)　⇔　L_1, L_2はリストであり、L_1の要素は全てL_2の要素でもある。

6．3　点と線だけで構成される図を例６．３の様にリストで表すことにする。この表記法の下で、点と線だけで構成される図とその中の２つの点 v, w が引数として任意に与えられた時、vから有限個の線を経由してwへ至れるかどうかを判定し、もしそれが可能ならvからwへの経路を見つけるPrologプログラム

を作成せよ。

6．4　覆面算

```
  BASE
 +BALL
 ------
 GAMES
```

を解くPrologプログラムを作成せよ。

6．5　覆面算

```
    しか
   ×とら
   ------
    こぐま
   こうし
  --------
   しまうま
```

を解くPrologプログラムを作成せよ。

6.6　それぞれの正整数について素数であるかどうかを判定する方法で、

与えられた正整数以下の素数のリストを生成するPrologプログラム

を作成せよ。

6.7　次の2つの英文を構文解析するためのPrologプログラムを作成せよ。

① the giraffe runs.

② the giraffe eats apples.

6.8　例5.5（図5.4）で示した階乗計算のPrologプログラム

```
factorial(0, 1).
factorial(X, Y) :-
    X > 0,
    X1 is X-1,
    factorial(X1, Y1),
    Y is X*Y1.
```

について、

(1) カット述語を用いて効率化／簡素化を図ってみよ。

(2) 条件分岐述語を用いて効率化／簡素化を図ってみよ。

6.9　例5.5（図5.8）で示した最大公約数計算のPrologプログラム

```
gcd(0, Y, Y).
gcd(X, 0, X).
gcd(X, Y, Z) :-
    X > 0,
    Y > 0,
    Remainder is X mod Y,
    gcd(Y, Remainder, Z).
```

を実行制御の述語を用いて効率化／簡素化せよ。

6.10　max(x, y, z)⇔zは{x, y}の最大値、という意味の述語 max を定義することによって、

2つの数値の最大値を求めるPrologプログラム

を作成せよ。

6.11　factorize(k, L)⇔kは正整数でありLはkの素因数を要素とする（すなわち$p_1 \times p_2 \times \cdots \times p_n = k$となる様な素数$p_1, p_2, \ldots, p_n$を要素とする）リストである、という意味の述語 factorize を再帰的に定義することによって、

与えられた正整数を素因数分解するPrologプログラム

を作成せよ。

6.12　２つの述語

i_sort(L_1 , L_2)　⇔　L_1は整数を要素とするリストであり、L_2はL_1の要素を小さい順に並べ替えて得られるリストである，

insert(a , L_1 , L_2) ⇔　aは整数，L_1は小さい順に並べられた整数から成るリストであり、L_2は“小さい順”を保つ様にL_1にaを挿入して得られるリストである

を再帰的に定義することによって、

任意に与えられた整数のリストについて

その要素を小さい順に並べ替えて昇順の整数リストを生成するPrologプログラム

を作成せよ。

6.13　**宣教師と人喰い人間の問題**(missionaries-and-cannibals problem)，すなわち

３人の宣教師と３人の人喰い人間が定員１～２人のボートで川を渡ろうとしている。その際、人喰い人間の数が宣教師の数を越えれば宣教師たちは食べられてしまうので、川のどちら側でもそうならない様にしないといけない。安全に川を渡るにはどの様な手順でボートを使えばよいか？

という問題を解くPrologプログラムを作成せよ。

6.14　n個のデータ　$x_1, x_2, \ldots, x_n$ について、次の式で定義される数値　μ, V, σ をそれぞれ**平均**(mean)，**分散**(variance)，**標準偏差**(standard deviation)という。

$\mu = (x_1 + x_2 + \cdots + x_n) / n$,

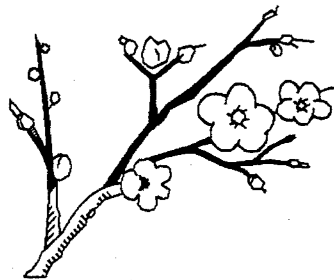
$V = \{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_n - \mu)^2\} / n$,

σ = Vの平方根

これらの定義式に従って、

与えられた項目についての平均，分散，標準偏差を計算するPrologプログラム

を作成せよ。［平方根を求めるためには、例えば漸化式　$a_1 = 1$，$a_{n+1} = (a_n + V / a_n) / 2$　($n \geq 1$) によって定まる数列 $\{a_n\}$ の極限値が Vの平方根であることに注意すればよい。従って、$|(a_n - a_{n-1}) / a_n|$ が十分小さければ　$\sigma \fallingdotseq a_n$ と近似できる。］



付録　組込み関数，組込み述語一覧

ここでは、ACOS上のPrologにあらかじめ組み込まれている関数,述語についてまとめておく。[一部の述語については省略する。]　以下では、(　)の中のページ番号は本文での参照頁を表す。

(1) 組込み関数

[算術演算関数]

- +X ……………… (94頁) Xと同じ値を持つ。
- -X ……………… (94頁) Xの符号を反転した値を持つ。
- abs(X) ………… (94頁) Xの絶対値。
- X+Y …………… (95頁) XとYの和。
- X-Y …………… (95頁) XからYを引いた値。
- X*Y …………… (95頁) XとYの積。
- X/Y …………… (95頁) XをYで割った時の商。[実数型の値が得られる。]
- X//Y …………… (95頁) XをYで割った時の商。[整数型の値が得られる。]
- mod(X, Y) ……… (95頁) XをYで割った時の余り。

[論理演算関数]（ここでは、X, Yの内部表現の成すビット列を考える。）

- ¥X ……………… (95頁) Xにビット毎の否定を施して得られる値。
- X/¥Y …………… (95頁) XとYにビット毎の論理積を施して得られる値。
- X¥/Y …………… (95頁) XとYにビット毎の論理和を施して得られる値。
- X@Y …………… (95頁) XとYにビット毎の排他的論理和を施して得られる値。
- X<<N …………… (95頁) XをNビットだけ左にシフトして得られる値。
- X>>N …………… (95頁) XをNビットだけ右にシフトして得られる値。

[型変換関数]

- float(X) ………… (95頁) Xの値を実数化したもの。
- floor(X) ………… (95頁) Xの値を切り捨てて整数化したもの。
- ceiling(X) ……… (95頁) Xの値を切り上げて整数化したもの。
- truncate(X) …… (95頁) Xの値を絶対値切り捨てによって整数化したもの。
- round(X) ………… (95頁) Xの値を四捨五入して整数化したもの。

(2) 算術式の評価／比較のための組込み述語

[算術式の評価]

- X is Y ………… (96頁) 算術式Yを評価してその値とXを単一化する。

[算術式の値の比較]

X=:=Y ・・・・・・・・・・・・・ (96頁) 算術式X, Yの評価値は等しい。
X=¥=Y ・・・・・・・・・・・・・ (96頁) 算術式X, Yの評価値は等しくない。
X>Y ・・・・・・・・・・・・・・・ (96頁) 算術式Xの値は算術式Yの値より大きい。
X>=Y ・・・・・・・・・・・・・・ (96頁) 算術式Xの値は算術式Yの値より大きいか等しい。
X<Y ・・・・・・・・・・・・・・・ (96頁) 算術式Xの値は算術式Yの値より小さい。
X=<Y ・・・・・・・・・・・・・・ (96頁) 算術式Xの値は算術式Yの値より小さいか等しい。

(3) 単一化

X=Y ・・・・・・・・・・・・・・・ (92頁) 項Xと項Yを単一化する。

(4) 実行制御のための組込み述語

[述語呼び出し]

call(α) ・・・・・・・・・・・ (149頁) ゴールαを実行する。
not(α) ・・・・・・・・・・・・ (149頁) ゴールαを実行しその成功／失敗の結果を反転する。

[実行制御]

fail ・・・・・・・・・・・・・・・ (148頁) 常に失敗する。
true ・・・・・・・・・・・・・・・ (149頁) 最初の一回だけ成功する。
! ・・・・・・・・・・・・・・・・・ (149頁) カット述語。ホーン節の中でこの述語の左側に現れるゴールについての試行錯誤を無効にする。
α->β ・・・・・・・・・・・・・ (150頁) 条件分岐述語。まずゴールαを実行し、これが成功すればβを実行する。αの実行が失敗すればこの述語の実行も失敗する。
α->β;γ ・・・・・・・・・ (149頁) 条件分岐述語。まずゴールαを実行し、これが成功すればβを，失敗すればγを実行する。
repeat ・・・・・・・・・・・・・ (150頁) 実行，再実行の度に無条件に成功する。

(5) 入出力のための組込み述語

[ストリームの生成／消去／操作]

open(F, M, S) ・・・・・・・・・・・・・ ファイルFをモードM (:read, :write, :append のいずれか) でオープンし、Fと結合したストリームSを生成する。
close(S) ・・・・・・・・・・・・・・・・・ ストリームSを消去し、Sと結合していたファイルをクローズする。
stream_name(S, F) ・・・・・・・・・ ストリームSと結合したファイルの名前をFと単一化する。
openfiles(L) ・・・・・・・・・・・・・・ 現在オープン中の非標準ストリームのリストをLと単一化する。
stream(S) ・・・・・・・・・・・・・・・・・ “Sはストリームである”という意味を持つ。
input_stream(S) ・・・・・・・・・・・ “Sは入力ストリームである”という意味を持つ。
output_stream(S) ・・・・・・・・・・ “Sは出力ストリームである”という意味を持つ。
current_input(S) ・・・・・・・・・・ “Sはカレント入力ストリームである”という意味を持つ。
current_output(S) ・・・・・・・・・ “Sはカレント出力ストリームである”という意味を持つ。
set_current_input(S) ・・・・・・ ストリームSをカレント入力ストリームに設定する。
set_current_output(S) ・・・・・ ストリームSをカレント出力ストリームに設定する。

[文字の入力]

ttyget0(C) ・・・・・・・・・・・・・・・・ 端末 (入力ストリーム) から次の文字を読み込みCと単一化す

る。

get0(*C*) ・・・・・・・・・・・・・・・・・・・・ カレント入力ストリームから次の文字を読み込み *C*と単一化する。

get0(*C*, *S*) ・・・・・・・・・・・・・・・・ 入力ストリーム *S*から次の文字を読み込み *C*と単一化する。

ttyget(*C*) ・・・・・・・・・・・・・・・・・ 端末（入力ストリーム）から空白以外の次の印字文字を読み込み *C*と単一化する。

get(*C*) ・・・・・・・・・・・・・・・・・・・・ カレント入力ストリームから空白以外の次の印字文字を読み込み *C*と単一化する。

get(*C*, *S*) ・・・・・・・・・・・・・・・・・ 入力ストリーム *S*から空白以外の次の印字文字を読み込み *C*と単一化する。

ttyskip(*C*) ・・・・・・・・・・・・・・・・ 文字型の項 *C*で表される文字が読まれるまで、端末（入力ストリーム）のデータを読み飛ばす。

skip(*C*) ・・・・・・・・・・・・・・・・・・・ 文字型の項 *C*で表される文字が読まれるまで、カレント入力ストリームのデータを読み飛ばす。

skip(*C*, *S*) ・・・・・・・・・・・・・・・・ 文字型の項 *C*で表される文字が読まれるまで、入力ストリーム *S*のデータを読み飛ばす。

[文字の出力]

ttyput(*C*) ・・・・・・・・・・・・・・・・・ 文字型の項 *C*で表される文字を端末（出力ストリーム）に書き出す。

put(*C*) ・・・・・・・・・・・・・・・・・・・・ 文字型の項 *C*で表される文字をカレント出力ストリームに書き出す。

put(*C*, *S*) ・・・・・・・・・・・・・・・・・ 文字型の項 *C*で表される文字を出力ストリーム *S*に書き出す。

ttynl ・・・・・・・・・・・・・・・・・・・・・・ 改行コードを端末（出力ストリーム）に書き出す。

nl ・・・・・・・・・・・・・・・・・ (114頁) 改行コードをカレント出力ストリームに書き出す。

nl(*S*) ・・・・・・・・・・・・・・・・・・・・・ 改行コードを出力ストリーム *S*に書き出す。

ttytab(*N*) ・・・・・・・・・・・・・・・・・ *N*個の空白を端末（出力ストリーム）に書き出す。

tab(*N*) ・・・・・・・・・・・・ (115頁) *N*個の空白をカレント出力ストリームに書き出す。

tab(*N*, *S*) ・・・・・・・・・・・・・・・・・ *N*個の空白を出力ストリーム *S*に書き出す。

[項の入出力]

read(*T*) ・・・・・・・・・・・ (114頁) カレント入力ストリームから項を読み込み *T*と単一化する。

read(*T*, *S*) ・・・・・・・・・・・・・・・・ 入力ストリーム *S*から項を読み込み *T*と単一化する。

display(*T*) ・・・・・・・・・・・・・・・・ 項 *T*を端末（出力ストリーム）に書き出す。

write(*T*) ・・・・・・・・・・ (114頁) 項 *T*をカレント出力ストリームに書き出す。

write(*T*, *S*) ・・・・・・・・・・・・・・・ 項 *T*を出力ストリーム *S*に書き出す。

writeq(*T*) ・・・・・・・・・・・・・・・・・ 項 *T*をカレント出力ストリームに書き出す。書き出す際は、再度読み込み可能になる様に必要に応じて引用符が挿入される。

writeq(*T*, *S*) ・・・・・・・・・・・・・・ 項 *T*を出力ストリーム *S*に書き出す。書き出す際は、再度読み込み可能になる様に必要に応じて引用符が挿入される。

(5) 知識ベース操作のための組込み述語

[ホーン節の登録]

assert(α) ・・・・・・・・・ (162頁) ホーン節 α を知識ベースに追加登録する。

asserta(α) ・・・・・・・・ (162頁) ホーン節 α を知識ベースの最初に追加登録する。

assertz(α) ・・・・・・・・ (162頁) ホーン節 α を知識ベースの最後に追加登録する。

[ホーン節の検索]

clause(α, β) ······ (163頁) α :- β. という形のホーン節を知識ベースの中から探し出す。

[ホーン節の削除]

abolish(F, N) ····· (163頁) 述語名が F, 引数の個数が Nの原子式を頭部に持つホーン節を全て知識ベースから削除する。

retract(α) ········ (163頁) αという形のホーン節を知識ベースの中から探し出して削除する。

[内部データベースへの項の登録]

recorda(K, T, R) ·········· Kを鍵として項 Tを内部データベースの最初に登録する。そして、登録データに対するレコードを Rと単一化する。

recordz(K, T, R) ·········· Kを鍵として項 Tを内部データベースの最後に登録する。そして、登録データに対するレコードを Rと単一化する。

[内部データベースの検索]

recorded(K, T, R) ········· Kを鍵として Tという形の項（が登録されたレコード）を内部データベースの中から探し出し、見つかったレコードを Rと単一化する。

instance(R, T) ············ レコード Rの中に登録された項を Tと単一化する。

[内部データベースからの項の削除]

erase(R) ·················· レコード Rを削除する。

（6）実行環境に関する組込み述語

[Prolog言語処理系からの脱出]

halt ················ (87頁) Prolog言語処理系との会話を終了させる。

[プログラムの読み込み]

consult(F) ········· (85頁) ファイル F上のプログラムをProlog言語処理系の中に読み込む。但し、既に登録されているホーン節は削除されない。

reconsult(F) ·············· ファイル F上のプログラムをProlog言語処理系の中に読み込む。但し、新たに読み込まれた手続きについては旧版は削除される。

[F_1, F_2, ... , F_n] ········ n個のファイル F_1, F_2, ... , F_n 上のプログラムをProlog言語処理系の中に読み込む。但し、ファイル名の前にハイフン(-)を付けるとそのファイルについては reconsult, ファイル名の前に何も付けないとそのファイルについては consult述語の読み込みが行われる。

[実行環境の制御]

lc ·················· (84頁) Prolog言語処理系を英小文字使用モードに切り替える。

'NOLC' ····················· 英小文字の代わりに英大文字を使用するモードにProlog言語処理系を切り替える。

prompt(P_1, P_2) ············ 端末入力の際のプロンプトを P_2という文字列（またはアトム）に変更し、古いプロンプトを P_1と単一化する。

[演算子の登録／登録抹消／一覧表示]

op(P, T, N) ·············· アトム Nを優先順位 P（$1 \leqq P \leqq 1200$），結合性 T（fx, fy, xf,

yf,xfx,xfy,yfx のいずれか）の演算子として登録する。
delete_op(*N*, *T*) ………… 結合性 *T* の演算子 *N* を登録抹消する。
current_op(*P*, *T*, *N*) ……… “優先順位 *P*,結合性 *T* の演算子 *N* が現在登録されている”という意味の述語。

[読み込まれたプログラム／手続きの表示]

listing …………… (87頁) その時点に登録されているホーン節を全てカレント出力ストリームに出力する。
listing(*P*) ……………… その時点に登録されているホーン節の内、*P* で指定されたものを全てカレント出力ストリームに出力する。*P* としては、手続き名を表すアトム，手続き名/引数の個数 という形の構造体，もしくはそれらのリストを指定できる。

[実行状態の保存と復元]

save(*F*) ………………… その時点の実行状態をファイル *F* に保存する。
restore(*F*) ……………… ファイル *F* に保存されている実行状態を復元する。

[プログラム実行の制御]

abort …………………… その時点に実行中のPrologプログラムを強制終了させる。
suspend ………………… その時点に実行中のPrologプログラムについて、実行を中断して新たな質問待ちの状態にする。中断した実行はresume述語の実行により再開される。
resume …………………… suspend述語によって中断したプログラム実行の内、最新のものを再開させる。

[デバッガ]

debugging ………… (112頁) その時点のデバッグ環境の情報を端末（出力ストリーム）に出力する。
debug …………… (112頁) Prolog言語処理系が標準の動作モードの時、処理系を非トレースのデバッグモードに切り替える。
nodebug ………… (112頁) Prolog言語処理系の動作モードを標準モードに切り替える。
trace …………… (112頁) Prolog言語処理系の動作モードをトレースモードに切り替える。
notrace ………… (112頁) Prolog言語処理系の動作モードがトレースモードの時、処理系を非トレースのデバッグモードに切り替える。
spy *P* ………… (112頁) *P* で指定された処理の箱をスパイ点に指定する。*P* としては、手続き名を表すアトム，手続き名/引数の個数 という形の構造体，もしくはそれらのリストを指定できる。
nospy *P* ……… (112頁) *P* で指定された処理の箱をスパイ点の指定から外す。*P* としては、手続き名を表すアトム，手続き名/引数の個数 という形の構造体，もしくはそれらのリストを指定できる。
leash(*M*) ……… (112頁) 外部制御モードを *M* に設定する。すなわち、*M* で指定した出来事の直後に情報表示の中断/再開を制御する様にする。

[性能評価]

runtime(*T*, *I*) ………… Prolog言語処理系の起動時からのＣＰＵ時間を *T* と，この述語の前回の実行時からのＣＰＵ時間を *I* と単一化する。
statistics ……………… その時点のメモリ使用量,その時点までのＣＰＵ時間,等を端末（出力ストリーム）に出力する。

statistics(*K*, *V*) ・・・・・・・・・・ その時点のメモリ使用量，その時点までのＣＰＵ時間，等の内、*K*で指定した項目の情報を *V*と単一化する。*K*としては:code, :data, :heap, :number, :global_stack, :local_stack, :trail, :runtime, :garbage_collection のいずれかを指定できる。

（７） その他の組込み述語

[項の識別]

atom(*T*) ・・・・・・・・・・・・・・・・・・・ “項 *T*はアトムである”という意味の述語。
integer(*T*) ・・・・・・・・・・・・・・・・ “項 *T*は整数である”という意味の述語。
float(*T*) ・・・・・・・・・・・・・・・・・・ “項 *T*は浮動小数点表示の実数である”という意味の述語。
number(*T*) ・・・・・・・・・・・・・・・・・ “項 *T*は整数，または浮動小数点表示の実数である”という意味の述語。
character(*T*) ・・・・・・・・・・・・・・ “項 *T*は文字型のデータである”という意味の述語。
string(*T*) ・・・・・・・・・・・・・・・・・ “項 *T*は文字列型のデータである”という意味の述語。
var(*T*) ・・・・・・・・・・・・・・・・・・・・ “項 *T*は値の全く決まっていない変項である”という意味の述語。
nonvar(*T*) ・・・・・・・・・・・・・・・・・ var(*T*)の否定述語。
atomic(*T*) ・・・・・・・・・・・・・・・・・ “項 *T*は原子定項（すなわち整数，実数，アトム，または文字型のデータ）である”という意味の述語。
nonatomic(*T*) ・・・・・・・・・・・・・・ atomic(*T*)の否定述語。
constant(*T*) ・・・・・・・・・・・・・・・ “項 *T*は定項（すなわち整数，実数，アトム，文字型のデータ，文字列型のデータ，ストリーム，レコード，ホーン節コード，またはリードテーブル）である”という意味の述語。
nonconstant(*T*) ・・・・・・・・・・・・ constant(*T*)の否定述語。
list(*T*) ・・・・・・・・・・・・・・・・・・・ “項 *T*はリストである”という意味の述語。
structure(*T*) ・・・・・・・・・・・・・・ “項 *T*は構造体である”という意味の述語。

[アトムの生成／操作]

gentemp(*A*) ・・・・・・・・・・・・・・・・ 文字列“t”で始まる新しいアトムを生成し *A*と単一化する。
gentemp(*A*, *P*) ・・・・・・・・・・・・・ 文字列 *P*で始まる新しいアトムを生成し *A*と単一化する。
atom_name(*A*, *N*) ・・・・・・・・・・・ アトム *A*の印字名を *N*と単一化する。
name(*A*, *L*) ・・・・・・・・・・・・・・・・ アトム *A*と，*A*の印字名を構成する文字のコードのリスト，の間の変換を行う。例えば、a,b,c の文字コードはそれぞれ 97, 98,99 だから name(abc,N) を実行すると N=[97,98,99] と単一化され、name(A,[97,98,99]) を実行すると A=abc と単一化される。
name_char(*A*, *L*) ・・・・・・・・・・・ アトム *A*と，*A*の印字名を構成する文字（を表す文字型データ）のリスト，の間の変換を行う。例えば name(abc,[#¥a,#¥b,#¥c]) は真である。

[構造体の生成／操作]

functor(*T*, *F*, *N*) ・・・・・・・・・・ 項 *T*の主ファンクタ（すなわち *T*の最も外側の関数名または述語名）を *F*と，主ファンクタの引数の個数を *N*と単一化する。あるいは、ファンクタが *F*，引数の個数が *N*の構造体（またはアトム，*N*＝０の場合）を生成し *T*と単一化する。
arg(*N*, *T*, *A*) ・・・・・・・・・・・・・ “構造体 *T*の主ファンクタに対する *N*番目の引数は *A*である”

という意味の述語。

T=.. L ………… (168頁) ユニブ述語。すなわち、構造体 Tと，Tの主ファンクタとその引数の成すリスト L，の間の変換を行う。

[リストの操作]

length(L, N) …………… リスト Lの長さ（すなわち要素数）を Nと単一化する。

[文字列の生成／操作]

string_element(S, N, C) … 文字列 Sの N番目の文字を Cと単一化する。

string_equal(S_1, S_2) …… “２つの文字列 S_1, S_2が等しい”という意味の述語。

list_to_string(L, S) …… 文字型データを要素とするリスト Lからそれらの文字を並べて得られる文字列を生成し、Sと単一化する。

string_to_list(S, L) …… 文字列 Sを構成する文字（を表す文字型データ）のリストを Lと単一化する。

[標準順序に基づく項の比較／整列化]（変項への新たな代入は許さない。）

compare(C, X, Y) ………… ２つの項 X, Yを標準順序に基づいて比較した結果（３つのアトム <,>,= のいずれか）を Cと単一化する。

X== Y …………… (94頁) “標準順序の下では項 Xは項 Yと等しい”という意味の述語。

X¥== Y …………… (94頁) X== Yの否定述語。

X@< Y ………………… “標準順序の下では項 Xは項 Yより小さい”という意味の述語。

X@=< Y ………………… “標準順序の下では項 Xは項 Yより小さいか等しい”という意味の述語。

X@> Y ………………… “標準順序の下では項 Xは項 Yより大きい”という意味の述語。

X@>= Y ………………… “標準順序の下では項 Xは項 Yより大きいか等しい”という意味の述語。

sort(L, S) ……………… リスト Lの要素を標準順序に基づいて小さい順に並べ替えた結果のリストを Sと単一化する。但し、並べ替えの際に同じ要素が重なった場合は、１個だけ生かして残りは除去される。

keysort(L, S) …………… リスト Lの要素（鍵-項 という形をしている）を要素の鍵の部分が標準順序で小さい順になる様に並べ替え、その結果のリストを Sと単一化する。但し、並べ替えの際に同じ要素が重なった場合でも、要素は除去されない。

[解の集合を求める]

setof(X, α, L) ………… 条件 αを満たす Xの集合を Lと単一化する。すなわち、ゴール αを実行した時の Xに関する解のリストを Lと単一化する。但し、重複解がある場合は１個だけを Lの要素とする。

bagof(X, α, L) ………… ゴール αを実行した時の Xに関する解のリストを Lと単一化する。但し、重複を許してそれぞれの解を Lの要素とする。

[集合演算]

setand(L_1, L_2, L) ……… リスト L_1, L_2を集合と見なしてそれらの共通部分を Lと単一化する。すなわち、リスト L_1, L_2に共通する要素のリストを Lと単一化する。

setor(L_1, L_2, L) ………… リスト L_1, L_2を集合と見なしてそれらの和集合を Lと単一化する。すなわち、リスト L_1, L_2のいずれかに属する要素のリスト（重複は許さない）を Lと単一化する。

setdif(L_1, L_2, L) ……… リスト L_1, L_2 を集合と見なしてそれらの差集合 $L_1 - L_2$ を L と単一化する。すなわち、リスト L_1 には属するがリスト L_2 には属さない要素のリストを L と単一化する。

[ファイルの存在を調べる]

exist_file(F) …………… “ファイル F が存在する”という意味の述語。

文 献

（第１～２章，論理）

［１］　Ｗ．Ｏ．クワイン（杖下隆英 訳）：　現代論理入門 ―ことばと論理― ，２３８頁，大修館書店，１９７２．
（１．１～３節，１．８節の一部，１．９節，２．２～３節，２．９節の考え方については、この本をかなり参考にした。）

［２］　沢田允茂：　（岩波新書 ４５２） 現代論理学入門，２３２頁，岩波書店，１９６２．

［３］　Ｊ．オールウド，Ｌ．-Ｇ．アンデソン，Ö．ダール（公平珠躬，野家啓一 訳）：　日常言語の論理学，２２４頁，産業図書，１９７９．

［４］　Ｌ．チャイカ（飛田就一，木戸正幸 訳）：　現代論理学の基礎 ―経済学での応用例―，１３８頁，富士書店，１９８３．

［５］　長尾真，淵一博：　（岩波講座 情報科学 ７） 意味と論理，２１２頁，岩波書店，１９８３．

［６］　C.-L. Chang, R. C.-T. Lee（長尾真，辻井潤一 訳）：　コンピュータによる定理の証明，３６０頁，日本コンピュータ協会，１９８３．

（第３章，３．１～４節，公理化）

［３．１節，公理と演繹による数学の体系化］

［７］　吉田洋一，赤攝也：　数学序説 改訂版，２９６頁，培風館，１９６１．

［８］　Ｓ．Ｆ．バーカー（赤攝也 訳）：　（哲学の世界 ６） 数学の哲学，１９２頁，培風館，１９６８．

［９］　Ｗ．Ｃ．サモン（山下正男 訳）：　（哲学の世界 １） 論理学 三訂版，２１４頁，培風館，１９８７．

［10］　Euclid（中村幸四郎，寺坂英孝，伊東俊太郎，池田美恵 訳・解説）：　ユークリッド原論，５７６頁，共立出版，１９７１．

［11］　D. Hilbert, F. Klein（寺坂英孝，大西正男 訳・解説）：　（現代数学の系譜 ７） ヒルベルト 幾何学の基礎，クライン エルランゲン・プログラム，４３８頁，共立出版，１９７０．

［12］　G. Peano（小野勝次，梅沢敏郎 訳・解説）：　（現代数学の系譜 ２） ペアノ 数の概念について，２００頁，共立出版，１９６９．

［３．２～４節，命題論理，述語論理の公理系］

［13］　杉原丈夫：　数学的論理学，１９０頁，槙書店，１９６７．

［14］　井関清志：　記号論理学（命題論理），３２０頁，槙書店，１９６８．

［15］　井関清志：　記号論理学（述語論理），２８４頁，槙書店，１９７３．

[16] Elliott Mendelson: Introduction to Mathematical Logic 2nd Edition, ３３６頁, D. Van Nostrand Company, １９７９.
[17] Alonzo Church: Introduction to Mathematical Logic, ３８８頁, Princeton University Press, １９５６.

(3.5節, 知識ベース; 第４～６章, Ｐｒｏｌｏｇ)

[全般]

[18] 長尾真 他（編）: 岩波情報科学辞典, １１７６頁, 岩波書店, １９９０.
(用語／訳語については、この本を主に参考にした。)

[3.5節, 知識ベース]

[19] 人工知能学会（編）: 人工知能ハンドブック, １０９８頁, オーム社, １９９０.
[20] 情報処理学会（編）: 情報処理ハンドブック, １６１０頁, オーム社, １９８９.
[21] 上野晴樹: 知識工学入門（改訂２版）, ２４２頁, オーム社, １９８９.
[22] Ｐ. ハーモン, Ｄ. キング（諏訪基 訳）: (Information & Computing-８) エキスパートシステムズ, ４９６頁, サイエンス社, １９８６.
[23] 長尾真: （岩波講座 ソフトウェア科学 １４）知識と推論, ３４６頁, 岩波書店, １９８８.

[Ｐｒｏｌｏｇ, 標準化]

[24] 中村克彦: 最終段階にきたＰｒｏｌｏｇ標準化の成果とポイント, *Ｉｎｔｅｒ* ＡＩ No.27(1992年2月号,国際ＡＩ財団／エーアイ・ビジネス)の17～22頁.

[Ｐｒｏｌｏｇ, 初心者向け（？）]

[25] 後藤滋樹: （ソフトウェアライブラリ １）ＰＲＯＬＯＧ入門 ―知識情報処理の序曲―, ２００頁, サイエンス社, １９８４.
(3.5節, 第4～6章の例題作成にあたっては、この本をかなり参考にした。)
[26] 塚本龍男: （情報処理入門シリーズ １２）わかる：－Ｐｒｏｌｏｇ, １１４頁, 共立出版, １９８９.
(我々と同じく、ACOS-Prologを用いてプログラムとその実行例を説明している。)
[27] 小川東: Ｐｒｏｌｏｇによる論理プログラミング入門, １８６頁, 啓学出版, １９９０.
[28] 塩野充: 例題演習Ｐｒｏｌｏｇ入門, １７６頁, オーム社, １９９１.
[29] 小谷善行: (Software Technology １０) 知識指向言語Ｐｒｏｌｏｇ ―人工知能プログラミングへの序曲―, ２００頁, 技術評論社, １９８６.
[30] 鑰山徹: RUN/Prologを用いたＰｒｏｌｏｇプログラミング入門, １６０頁, 工学図書, １９８７.
[31] 杉原敏夫: ＲＵＮ／Ｐｒｏｌｏｇとその応用, ２２４頁, 工学図書, １９８７.

[Ｐｒｏｌｏｇ, 中級以上（？）]

[32] W. F. Clocksin, C. S. Mellish（中村克彦 訳）: Ｐｒｏｌｏｇプログラミング［改訂第3版］, ３３６頁, マイクロソフトウェア／日本コンピュータ協会, １９８８.
(DEC-10 Prolog の標準的な教科書／参考書。DEC-10 Prolog については、この本を主に参考にした。)
[33] 新田克己, 佐藤泰介: （人工知能用言語シリーズ １）Ｐｒｏｌｏｇ, １９６頁,

昭晃堂，１９８６．
(DEC-10 Prolog のコンパクトな教科書。)
[34] 太細孝，鈴木克志，伊草ひとみ，佐藤裕幸： Ｐｒｏｌｏｇ入門，２４０頁，啓学出版，１９８４．
[35] 中島秀之： (コンピュータサイエンスライブラリ) Ｐｒｏｌｏｇ，１７４頁，産業図書，１９８３．
(日本で最初に出版された Prolog の入門書／教科書。色々なプログラミング技法が説明されていて読み易いが、DEC-10 Prolog と少し違う。)
[36] 安部憲広： Ｐｒｏｌｏｇプログラミング，２０８頁，共立出版，１９８５．
[37] 古川康一： Ｐｒｏｌｏｇ入門，１９８頁，オーム社，１９８６．
[38] 中村克彦： Ｐｒｏｌｏｇと論理プログラミング，１５４頁，オーム社，１９８５．
[39] I. Bratko (安部憲広 訳)： (Prolog と AI １) Ｐｒｏｌｏｇへの入門，２３２頁，近代科学社，１９９０．
[40] 玉井浩： (Information & Computing-３２) ＴＵＲＢＯ Ｐｒｏｌｏｇプログラミング，１４４頁，サイエンス社，１９８９．
[41] Ｒ．バラス (斉藤重光，舟本奨 訳)： Ｐｒｏｌｏｇ詳説 —対話形式によるアプローチ—，２２４頁，啓学出版，１９９０．
[42] Ｄ．シェーファー (北脇和夫，北脇庸子 訳)： 入門ＴＵＲＢＯ ＰＲＯＬＯＧ，３１２頁，啓学出版，１９８９．
[43] 日本電気： (ＡＣＯＳソフトウェア) ＰＲＯＬＯＧ言語説明書 第５版，２００頁，日本電気，１９９１．
(日本電気のオペレーティングシステム ACOS-4/AVP XR, ACOS-4/MVP XE, ACOS-4/XVP, ACOS-6/MVXⅡ の下の Prolog について説明したマニュアル。)

[Ｐｒｏｌｏｇ，変わり種]

[44] 福田敏宏，田村三郎，田中正彦： ＳＦ的Ｐｒｏｌｏｇの世界 —コンピュータ・ウィルス盛衰記—，２０８頁，現代数学社，１９９０．
[45] 加賀山茂： 法律家のためのコンピュータ利用法 —論理プログラミング入門—，３４４頁，有斐閣，１９９０．
[46] 飯高茂： Ｐｒｏｌｏｇで作る数学の世界 —Prolog そして集合-位相-群—，２３６頁，朝倉書店，１９９０．

[Ｐｒｏｌｏｇ例題／プログラム集]

[47] Helder Coelho, Jose Carlos Cotta, Luis Moniz Pereira： How to Solve it with Prolog 4th edition，２２４頁，Laboratorio Nacional de Engenharia Civil，１９８５．
(多方面に渡る例題が豊富に (全部で１２０題) 揃っていて、３．５節，第４〜６章の例題作成の際はこの本をかなり参照した。)
[48] Ｂ．フィリピッチ (中島誠，伊藤哲郎 訳)： Ｐｒｏｌｏｇユーティリティ ライブラリ，１６０頁，海文堂，１９９０．
[49] 山田眞市： (Information & Computing-７) 人工知能のためのｍｉｃｒｏ−ＰＲＯＬＯＧ プログラムコレクション，２７２頁，サイエンス社，１９８６．

[Ｐｒｏｌｏｇ，応用]

[50] 溝口文雄 (監著)，武田正之，畝見達夫，溝口理一郎： Ｐｒｏｌｏｇとその応用 ２ —プログラム作成支援／エディタ設計／自然言語処理／データベース—，３３０頁，総研

出版，１９８５．

[Ｐｒｏｌｏｇを用いた論理プログラミング，考え方／技法の説明]

[51] 黒川利明：（岩波コンピュータサイエンス）Ｐｒｏｌｏｇのソフトウェア作法，２８０頁，岩波書店，１９８５．

[52] L. Sterling, E. Shapiro（松田利夫 訳）：Ｐｒｏｌｏｇの技芸，５９４頁，構造計画研究所／共立出版，１９８８．

[Ｐｒｏｌｏｇ／論理プログラミングの原典]

[53] Ｒ．コワルスキ（浦昭二 監；山田眞市，菊池光昭，桑野龍夫 訳）：（情報処理シリーズ ８）論理による問題の解法 ―Ｐｒｏｌｏｇ入門―，４１６頁，培風館，１９８７．

[その他]

[54] D. E. Knuth（渋谷政昭 訳）：（KNUTH The Art of Computer Programming 第３分冊）準数値算法／乱数，２６２頁，サイエンス社，１９８１．

[55] Ｊ．Ａ．Ｈ．ハンター（藤村幸三郎，田村三郎 訳）：（ブルーバックス ４４８）パズル・ショートショート ―数学“悶”題集―，２８８頁，講談社，１９８１．

[56] 大駒誠一，武純也，丸尾学：虫食い算パズル７００選，２２４頁，共立出版，１９８５．

[57] 外務省外務報道官（編）：世界の国一覧表 １９９２年版，４０頁，世界の動き社，１９９２．

[58] ＪＲ運賃研究会（編）：ＪＲ運賃の大研究，２２４頁，風濤社，１９８８．

[59] Ｈ．フィッシャー（瀬田貞二 訳）：（グリム童話）ブレーメンのおんがくたい，３０頁，福音館書店，１９６４．

索　引

あ 行

アッケルマン ………… 59
アトム ……………… 75

一階 ………………… 164
一階論理式 ………… 37
一般化規則 ………… 63

ヴェブレン ………… 57
ウカシェヴィッチ …… 59,63

ACOS ………………… 73
英小文字使用モード …… 84
英-和の間の単語翻訳 ……… 117,160
エキスパートシステム …… 66
エディタ …………… 84
N-Queens問題 …… 140
エラー ……………… 96
エラトステネスの篩い ……… 138,159
演繹可能 …………… 62
演繹定理 …… (13,44,)62,65
演繹的推論 ………… 55
演繹不能 …………… 81
演算子 ……………… 164

オーム返し(のプログラム) ……… 115,154
大相撲のデータベース … 168
置き換え法則 ……… 15,45
オペレータ ………… 164

か 行

改行(の述語) ……… 103
解釈 ………………… 9,40
階乗計算 …………… 98
確定節 ……………… 77
型変換関数 ………… 95
カット(述語) ……… 148,149
カット述語を用いた場合分け ……… 156
カレント出力ストリーム ‥114
カレント入力ストリーム ‥114
含意 ………………… 4
関数文字 …………… 36
冠頭標準形 ………… 48
冠頭選言標準形 …… 48
冠頭連言標準形 …… 48

偽 …………………… 1
疑似乱数の生成 …… 165
規則 ………………… 77
帰納的推論 ………… 55
基本論理式 ………… 37
吸収律 ……………… 15
共通概念 …………… 56

空リスト …………… 131
組込み述語 ………… 85
クリーネ …………… 59
繰り返し(の述語) … 150

型式 ………………… 28
決定的 ……………… 159
結合詞 ……………… 2
結合子 ……………… 6
結合律 ……………… 16,47
原子開放言明 ……… 35
原子言明 …………… 2
限嗣再帰法 ………… 100
原子式 ……………… 75,76
原子命題 …………… 2
原子論理式 ………… 37
ゲンツェン ………… 59,63
限定された全称限量詞 …… 32
限定された存在限量詞 …… 33
言明 ………………… 1
言明文字 …………… 8
限量詞 ……………… 29
限量子 ……………… 31
限量的帰結 ………… 50
限量的構造 ………… 50
限量的に偽 ………… 50
限量的に真 ………… 50
限量的に同値 ……… 50
原論 ………………… 56

ゴール ……………… 77
ゴール節 …………… 77
項 …………………… 37,76
高階 ………………… 164
交換律 ……………… 16,47
後件 ………………… 4
後者 ………………… 57
項出力(の述語) …… 114
公準 ………………… 56
恒真 ………………… 10,41

恒真式 …… 10
構造 …… 77
構造体 …… 77
項入力(の述語) …… 114
構文解析 …… 143
公理 …… 56,57
公理系 …… 57
公理主義 …… 57
個体文字 …… 36
コマンド …… 77,84
コルモゴロフ …… 58

さ 行

再帰法 …… 98
再実行 …… 117
最大公約数の計算 …… 101
差分リスト …… 143
算術演算関数 …… 94
算術式の評価(の述語) …… 96
算術比較(の述語) …… 96

ＪＲ普通運賃の計算 …… 156
字下げ(の述語) …… 115
事実 …… 77
次数 …… 135
実行制御の述語 …… 148
実行(過程の)追跡 …… 91,103
実数(データの表し方) …… 76
失敗 …… 94
質問 …… 77
充足可能 …… 10,42
充足不能 …… 10,41
終端再帰法 …… 100
自由変項 …… 37
重リスト …… 143
シュタイニッツ …… 57
出現チェック …… 94
述語文字 …… 36
条件分岐述語 …… 149,150
証明 …… 61
証明図 …… 61
証明図の探索 …… 78
処理の繰り返し …… 154,160
真 …… 1
人口密度の計算 …… 97
真理関数的 …… 2
真理関数的帰結 …… 21
真理関数的型式 …… 8
真理関数的に偽 …… 21
真理関数的に真 …… 21
真理関数的に同値 …… 21
真理値 …… 1
真理表 …… 10

推論エンジン …… 66
スパイ点 …… 108

成功 …… 94
整数(データの表し方) …… 76
全解の自動生成 …… 124,156
宣教師と人喰い人間の問題 …… 177
線形合同式法 …… 165
線形合同法 …… 165
選言 …… 3
選言標準形 …… 18
前件 …… 4
全称限量詞 …… 29
前置部 …… 48
専門家システム …… 66

素式 …… 76
素数の生成 …… 138,159
素論理式 …… 37
存在限量詞 …… 30

た 行

対合律 …… 16,47
代入 …… 46
代入法則 …… 15,46
タルスキー …… 59
タレス …… 55
単位元律 …… 16,47
単一化 …… 92

知識システム …… 66
知識準拠システム …… 66
知識ベース …… 66
知識ベースシステム …… 66
知識ベース操作(の述語) …… 162
注釈 …… 77

データベース …… 66
ｄ-リスト …… 143
定理 …… 61
DEC-10 Prolog …… 73
手続き的解釈 …… 103
デバッガ …… 108
デバッグ …… 91
デバッグモード …… 108

同値(の結合詞) …… 4
頭部 …… 77
閉じた(一階論理式) …… 37
ドモルガンの法則 …… 16,47
トレースモード …… 108

な 行

ニコド …… 59

は 行

バード …… 103
排他的選言 …… 4
箱モデル …… 104
ハノイの塔問題 …… 124
ハンティングトン …… 57

引数 …… 36
ピタゴラス …… 55
否定(結合詞) …… 3
否定述語 …… 149
一筆書き …… 135
非トレースモード …… 108
非ユークリッド幾何学 …… 57
表述 …… 1
標準偏差 …… 177
標準モード …… 108

開いた(一階論理式) …… 37
ヒルベルト ………… 57,59

ファイル ………… 84
フィボナッチ数列(の計算) …… 170
４-Ｑｕｅｅｎｓ問題 …… 120
複合言明 ………… 2
複合項 ………… 76
複合命題 ………… 2
覆面算 ………… 136
プライア ………… 63
プラトン ………… 55
フレーゲ ………… 59
ブレーメンの音楽隊(の公理系／プログラム) …… 67,74,80,84,103,104,109,111
プログラム ………… 77
プログラムの簡素化 …… 156
Prolog ………… 73
Prolog言語処理系 …… 78
プロンプト ………… 85
分散 ………… 177
分配律 ………… 16,47
分離規則 ………… 59

ペアノ ………… 57
ペアノの公理系の下での加算 …… 68,74,84
平均値の計算 ………… 168
べき等律 ………… 16,47
別形 ………… 47
ベルナイス ………… 63
変形箱モデル ………… 104
変項 ………… 32,75

ホーン節 ………… 75,77
補元律 ………… 16,47
母式 ………… 48
ボディ ………… 77
while-do ………… 160
本体 ………… 77

ま 行

末端再帰法 ………… 100

虫食い算 ………… 118
虫取り ………… 91
矛盾 ………… 10,41
矛盾式 ………… 10
無矛盾 ………… 10,42
無矛盾性＆完全性定理 …62,65
無名変項 ………… 75

命題 ………… 1
命題論理式 ………… 8
メレディス ………… 59

モーダスポネンス ………… 59
文字(データの表し方) …… 76
文字列(データの表し方) … 76

や 行

ユークリッド ………… 55
ユークリッドのアルゴリズム …… 101
ユークリッドの互除法 … 101
ユニブ(述語) ………… 168
ユニフィケーション …… 92

ら 行

ラッセル ………… 59,63

リスト ………… 76,131
リストの結合・分離 …… 132
リストの反転 ………… 133
両立的選言 ………… 3

連言 ………… 3
連言標準形 ………… 18
連想計算 ………… 172

ロサー ………… 59

論理演算関数 ………… 95
論理的帰結 ………… 13,44
論理的に同値 ………… 14,45

英数字・記号

[用語]

⊨に関する演繹定理 … 13,44
∀-導入規則 ………… 64,65
４-Ｑｕｅｅｎｓ問題 …… 120
ACOS ………… 73
ｄ-リスト ………… 143
DEC-10 Prolog ………… 73
ＪＲ普通運賃の計算 …… 156
N-Ｑｕｅｅｎｓ問題 …… 140
Prolog ………… 73
Prolog言語処理系 …… 78
while-do ………… 160

[論理記号]

～ ………… 6
∧ ………… 6
∨ ………… 6
⊕ ………… 6
→ ………… 6
↔ ………… 6
∀ ………… 31
∃ ………… 31
$\models_I$ ………… 9,40
$\not\models_I$ ………… 9,40
⊨ ………… 10,13,41,44
≡ ………… 14,45
$\vdash_{\mathfrak{L}}$ ………… 61,62
true ………… 16,47
false ………… 16,47

[Prolog内の組込み関数]

* ………… 95,179
+ ………… 94,95,179
- ………… 94,95,179
/ ………… 95,179
// ………… 95,179
/¥ ………… 95,179
<< ………… 95,179

>> ………………… 95,179
@ ………………… 95,179
¥ ………………… 95,179
¥/ ………………… 95,179
abs ………………… 94,179
ceiling ………………… 95,179
float ………………… 95,179
floor ………………… 95,179
mod ………………… 95,179
round ………………… 95,179
truncate ………………… 95,179

[Prolog内の組込み述語]
! ………………… 149,180
-> ………………… 150,180
-> ; ………………… 149,180
< ………………… 96,180
= ………………… 91,180
=.. ………………… 168,185
=:= ………………… 96,180
=< ………………… 96,180
== ………………… 94,185
=¥= ………………… 96,180
> ………………… 96,180
>= ………………… 96,180
¥== ………………… 94,185
abolish ………………… 163,182
assert ………………… 162,181
asserta ………………… 162,181
assertz ………………… 162,181
call ………………… 149,180
clause ………………… 163,182
consult ………………… 85,182
debug ………………… 112,183
debugging ………………… 112,183
fail ………………… 148,180
halt ………………… 87,182
is ………………… 96,179
lc ………………… 84,182
leash ………………… 112,183
listing ………………… 87,183
nl ………………… 114,181
nodebug ………………… 112,183
nospy ………………… 112,183
not ………………… 149,180
notrace ………………… 112,183
read ………………… 114,181
repeat ………………… 150,180
retract ………………… 163,182
spy ………………… 112,183
tab ………………… 115,181
trace ………………… 112,183
true ………………… 149,180
write ………………… 114,181