述語論理型言語 Prolog の紹介

杉　本　英　二

目　次

0. 知識を持つコンピュータ
A コンピュータ利用の現状
B 意味を処理するコンピュータ
C 知識を扱うためのプログラム言語
1. 論理プログラミング
A 記号論理による推論
B データベースと推論
C 発見的探索と再帰的定義
D 節の順序づけ
2. Prolog の利用法
A マルセイユ版 Prolog の移殖
B Prolog の起動法
C 構文と基本述語
3. Prolog プログラム例題集
A リストの接続
B リストの反転
C オペレータ宣言
D 事実データを全部表示する
E データのキー入力
4. 参考文献

0. 知識を持つコンピュータ

A. コンピュータ利用の現状

今日のコンピュータ利用は，OA (Office Automation) ブームといわれるほ

原稿受領日　1982年5月10日

どに社会の各層に広がってきている。その利用分野は，気象衛星，天体観測のためのパラボラアンテナ制御，原子炉設計などのように大規模な数値計算を必要とするものから，国鉄みどりの窓口，銀行オンラインなどのように大量データの即時処理，あるいはオフィス文書作成からホビー用ゲームマシンに至るまで広がり，その利用形態は著しく多様化している。

コンピュータの利用人口と利用分野がこのように広がってくると大量かつ多様な需要にソフトウエアの生産が追いつかなくなるというソフトウエア危機が来ると予測されはじめてほぼ15年になる。これまでソフトウエア技術者の育成と再教育，あるいは生産法の改善で，そのような危機に直面せずにすんでいるように見える。しかし，まもなくソフトウエア技術者の充足率は70％から30％までに落込むだろうとの予測もあって，無事回避されたと言い切れないようである。

このようにソフトウエアの高級技術者が容易には集められなくなる状況が進めば，現在のような人海戦術的なシステム開発法から，もっと省力化できる方式に移行せねばなるまい。ここにコンピュータを数値計算機としてでなく，人間が作り出してきた様々の知識の宝庫として，あるいは細部をトコトン突き詰めてくれる道具として利用しようという新しい方式が模索される。

B. 意味を処理するコンピュータ

東京大学の大須賀節雄教授は，情報処理学会の佳作論文［Ohs］の中で，「初期の計算機に課せられた要求は「数学的に定義された関数の値を数値的に高速に求めること」であり，「計算機の基本設計思想は今日までほとんど変化していない」と指摘している。そのような特殊分野向けの機能を用いて，新しい分野で生ずる広汎な要求を満たすように努力するのが応用技術であり，このような努力も限界に達しようとしていると現状を認識し，新しい計算機システムとして，知識型システムを提案している。

知識型システムが登場する背景として，特に①問題が発生時点で計算機に与えられるので，以前のプログラム開発の定石のように「問題を数学的体系という整った枠組内の表現に変えるための抽象化や意味処理を人手で行う」という

訳にはいかないこと，②問題の解法が必ずしも与えられていないので，解法を計算機自体が自立的に見つけることが必要なこと，以上の2点が大須賀によって挙げられている。

この背景から問題が人間の側に近い形で与えられるので，人間が与える問題を理解するための知識を前もって知識ベースとして持っているべきこと，また知識を組合わせながら推論によって問題を解くという意味処理を行なうことが知識型システムとして必要である。

大須賀はこのような知識型システム実現のための方式を提案した [Ohs] が，最近のこの分野の研究開発はめざましく進み，知識工学といわれる分野を形成している。このような例として，専門家の知識を計算機にたくわえ計算機と対話しながら，仕事を進めるというエキスパートシステムなどがあって，今後ますます実用化が期待されている。

C. 知識を扱うためのプログラム言語

知識型システムは，知識を人間に近い形で表現でき，それらの知識を用いて仮説の成立を推論するという機能を持つ。このような機能は従来のプログラム言語 (COBOL, FORTRAN PASCAL) などでは対応できない。これまで，このような目的のために使われてきた言語として LISP がある。LISP は人工知能の研究の中心的言語であり，知識を扱うほとんどのシステムはこの LISP 言語で書かれている。

ところが最近 Prolog という言語が注目を集めている。通産省が中心になって推進している第5世代コンピュータシステムの研究開発プロジェクトでは，将来のコンピュータの中心になるプログラム言語は，述語論理型の言語とする報告を出している [Uch]。

Prolog は述語論理型の言語の代表的な言語で，その名前の由来は PROgram＋LOGic から来ている。この Prolog が LISP 以上に高く評価されたのは，その言語が内蔵している問題解決・推論機能と，プログラム作成上生産性が高いという特徴からである [Fur]。

本レポートは，このプログラム言語の紹介を目的としている。特に教育的視

点から見て，論理性を中心とする言語によるプログラム作成の経験は，従来の手続言語教育では果せなかった

- 論理的構成，
- データ構造と再帰的呼出し，
- 知識表現，
- 知識のメタ操作，

などの点で効果が期待できる。本レポートでは1章で論理プログラムの方法を紹介し，2章では実際の Prolog 言語の利用法を述べる。このような言語にまったくの初心者は，1章から読まれることを進める。特に論理学を必要とするものではないから気軽に入っていけると思う。

研究開発，たとえば DSS (Dicision Support System) や，QAS (Question-Answering System) などのデータベースとの対話システムを開発する場合にも，従来の言語による方法の半分以下のプログラミングですむと思われるので開発手段としては非常に有効であろう。現在，本学に移殖した Prolog は Fortran によるインタープリタ方式で実行速度がかなり遅いのが欠点である。将来 Prolog 言語のコンパイラが用意されると思われるが，それにはまだ時間がかかるであろう。実際に大量のデータベースを作るのでなくて，動作実験程度の試作ならば十分に有効であろう。

1. 論理プログラミング

A. 記号論理による推論

次の三段論法を例に用いよう：

[例 A-1]

(a)	犬は動物である。	(a)′	$\forall x\,(\mathrm{Dog}\,(x) \rightarrow \mathrm{Animal}\,(x))$
(b)	ポチは犬である。	(b)′	Dog (POCH)
(c)	ポチは動物である。	(c)′	Animal (POCH)

三段論法は，(a) を大前提，(b) を小前提として，結論 (c) の成立を主張するものである。右側の表現は左側を記号化したもので，特に (a′) には，全称

記号 ∀ があり，Dog (x) を満たすどんな x についても，Animal (x) が成り立つことが表現されている。従って，(b)′ が成り立つなら (a′) で x をPOCHとした特定の場合も成り立つ；Dog(POCH)→Animal(POCH) (x=POCH) こうして，(c′) が得られる。

このような論理計算を自動化する方法を次に示そう。そのためにまず新しい記法を導入する。

〔節形式〕(a′) のように，限量記号が全称のみである場合は，それを省略して，

Animal (x)← Dog (x)

と書くことにする。これを節形式という[1]。

節形式の一般形は，

$$A_1 \vee A_2 \vee \cdots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n$$

である。これで表現されたものを節という。

〔ホーン形式〕節形式のうち，$m \leqq 1$ なる場合，それをホーン形式という。論理プログラミングでは，ホーン節のみを用いる。よって次の4のつタイプの節がある。

(1) $A \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n$ (略記 $A \leftarrow B_1, B_2, \cdots, B_n$)

(2) $A \leftarrow$

(3) $\leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n$ (略記$\leftarrow B_1, B_2, \cdots, B_n$)

(4) $\leftarrow$

〔句〕節を構成するさらに小さい単位，$A, B_1, B_2, \cdots, B_n$ などをそれぞれ句という。特に記号←の左辺を正の句，右辺を負の句とよぶことがある。その理由は←記号を or (∨) と not (～) で表わせば，(1) は，

$$A \vee \sim B_1 \vee \sim B_2 \vee \cdots \vee \sim B_n$$

となるからである。

1) 一般的には，述語論理式の表現には，2つの限量記号 ∃ と ∀ とが使われる。例えば，任意の整数 x に対して，x^2 に等しい正の数 n が存在することを，$\forall x \exists n\ (n=x^2,\ n>0)$ と表わすなどのように。しかし，スコーレムの定理〔Man〕によれば，すべての存在記号を取除く同値変形法があって，どんな論理式も節形式に変換できることがわかっている。

〔頭部と本体〕正の句を節の頭部とよび，残りを節の本体とよぶ。

〔手続〕ホーン節のうちタイプ (1) のものを手続とよぶ。その理由は，もし A の成立を主張するなら，B_1, B_2,…, B_n のすべての成立の確認を示すものだから。

〔主張〕ホーン節のうちタイプ (2) のものを主張とよぶ。これは事実情報を示すと考えられるから。

〔ゴール節〕ホーン節のうちタイプ (3) のものをゴール節，あるいは，プログラムとしてコマンドとよぶ。この節は証明の目標を与えるものと考えられるから。

〔空節〕タイプ (4) は空節とよばれ，矛盾を示す。

例 A-1 をホーン節で表現しなおす。

[例 A-2]

(a″) が手続，(b″) が事実，(c″)がゴール節である。

(a″)	Animal(x)←Dog(x)
(b″)	Dog(POCH)←
(c″)	←Animal(POCH)

〔反証による証明〕

数学には背理法という証明法がある。それは証明したい結論を否定し，これを前提に加えて矛盾を示すというのである。この反証の過程で得られる反例を解とするのが，論理プログラミングである。

例 A-2 を用いてこれを説明する。(c″) が結論の否定であり，論理式としては，～Animal (POCH) と同値になっている。以下に証明プロセスを追って説明する。

(Step 0) 〔ゴールの設定〕証明の目標を結論の否定として設定する。

←Animal(POCH)……………(c″)

(Step 1) 〔手続の探索〕ゴール節の句と同じ述語を頭部にもつ節を探す。

Animal(x)←Dog(x)…………(a″)

(Step 2) 〔パターン照合〕2つの述語の引数がまったく同一のパターンになるように，変数に対する代入を決める。この場合 x=POCH となる。

Animal(POCH)← Dog(POCH)

(Step 3) 〔手続のゴール節への置換〕パターン照合によって得られた新しい手続の本体で，ゴール節の句を置換する。つまり，

{ ←Animal(POCH)　(ゴール節)
Animal(POCH)← Dog(POCH)　(新しい手続)

⇩

←Dog (POCH)　(新しいゴール節)

これ以下は再び繰返しになる：

(Step 0′)　←Dog(POCH)

(Step 1′)　Dog(POCH)←…………(b″)

(Step 2′)　すでにパターンは一致している。

(Step 3′)　{ ←Dog(POCH)　(ゴール節)
Dog (POCH)←　(新しい手続)

⇩

←　(新しいゴール節)

ここで得られた新しいゴール節は左右両辺の句がない。これを空節とよぶ。空節は「無条件に矛盾が生じる」ことを示すもので，反証は成功したことになる。これによって論理プログラムの証明は停止する。

以上のプロセスの中で得られたゴール節のみを並べて簡略に次のように表わすことにしよう：

以上の計算手続 (step 1, 2, 3) によって推論，すなわち反証が自動化されていることがわかる。

①←Animal (POCH)	
②←Dog (POCH)	((a″) による)
③←	((b″) による)

B. データベースと推論

事実情報を蓄積したデータの集合をデータベースという。データには，データの間にデータそれ自体で表わす情報以外に，関係という情報がある。例えば親という関係があれば，これを重ねることで祖父という関係を定義することが

可能である。だから祖父という関係を表わすデータが直接得られていなくても祖父の定義から計算で作り出せる。

データの間の基本的関係を定義しておき，これらを自由に組合せて必要とされる情報をさらに定義できるなら，データベースは知的だと言えよう。

[例 B-1]

(d 1)　太郎は二郎の親である。 (d 2)　二郎は山彦の親である。 (d 3)　花子は山彦の親である。 このとき，山彦の祖父母は誰か？

ところで，祖父母とは，親の親のことだから，Parent (x, y) を x は y の親ということにすれば，

(axiom g) Grandparent(x, z)←Parnt(x, y), Parent(y, z) となる。3つのデータは，

(d 1′)　Parent(TARO, JIRO)←

(d 2′)　Parent(JIRO, YAMAHIKO)←

(d 3′)　Parent(HANAKO, YAMAHIKO)←

となる。このとき，質問は次のようになる：

←Grandparent(*parson*, YAMAHIKO)

反証の目標としてこれを解釈すると，「YAMAHIKO の祖父母となる人(*parson*) は存在しない」という意味になる。証明は，*parson* を特定した反例を作ればよいことなる。以下に証明のプロセスを示す：

①←Grandparent(*parson*, YAMAHIKO)

②←Parent(*parson*, y), Parent(y, YAMAHIKO)　(axiom g による)

　　(この部分目標を始めに決める)

③←Parent(JIRO, YAMAHIKO)……(d1′ による，*parson*=TARO)

④←　　　　　　　　　　…………(d 2′ による)

②は (axiom g) によって一つの目標を二つの部分目標の and 条件に分解している。このような目標の分解は問題解決システムの開発には必須の機能のひとつであって，論理プログラムの特徴である。

③に移る際に，ゴール節中に複数の目標があれば，左端から選択が行なわれ，パターン照合される。d 1′ とのパターン照合によって，*parson* は TARO と

決まるが同時に y も決められ，その情報は②の Parent(y, YAMAHIKO) の y にも伝えられる。結果として③が得られる。その後③から空節が導びかれることによって，③で得られた *parson*=TARO が求めている解になる。

C. 発見的探索と再帰的定義

B において，論理プログラムの特徴である目標を部分目標の論理積に分解することを説明したが，ここでは問題解決システムにとって非常に強力な発見的探索と再帰的定義を示す。

B の例を拡張して次のようにする：

(c 1)	Parent (JIRO, YAMAHIKO)←
(c 2)	Parent (TARO, ICHIRO)←
(c 3)	Parent (TARO, JIRO)←
(c 4)	Ancestor (x, z)←Parent (x, z)
(c 5)	Ancestor (x, z)←Parent (x, y), Ancestor (y, z)

このとき，太郎が山彦の祖先かどうかを知るには，Ancestor (TARO, YAMAHIKO) が示されればよいから以下のようになる。

①←Ancestor(TARO, YAMAHIKO)

②← Parent(TARO, YAMAHIKO) ((c 4 より))

ここで Parent (TARO, YAMAHIKO)←なる節は見つからないので，(c 4) を使ったのは，好ましくはなかったことがわかる。そこで新ためて，(c 5) を採用する。つまりバックトラックする。

②←Parent(TARO, y), Ancestor(y, YAMAHIKO)……(c 5 による)

③←Ancestor(ICHIRO, YAMAHIKO)……(c 2 により)

④←Parent(ICHIRO, YAMAHIKO)…… (c 4 により)

④←Parent(ICHIRO, y), Ancestor(y, YAMAHIKO)……(バックトラックして，c 5 による)

ここでも一郎を親とする節はないので，バックトラックする。③に続くものをすべて試みたから，そもそも③が誤まりであったとわかるから②へもどる。

③←Ancestor(JIRO, YAMAHIKO)……(バクトラックして，c3による)

④←Parent(JIRO, YAMAHIKO)……(c4による)

⑤← ……(c1による)

ようやく証明が終った。このような試行錯誤による探索を発見的探索といい，ゲームのプレーなどによく見られるものである。

問題解決システムでは，問題の解法アルゴリズムがわかっていないことが多く，この発見的探索を自動的に行ってくれる機能は他のプログラム言語にはない特徴である。

また再帰的定義は，構造をもつデータを取扱うには必須の機能である。この例では (c5) がその再帰的定義になっている。つまり，人 y が人 z の祖先なら，人 y の親である人 x も人 z の祖先である。再帰的というのは，左辺の述語の定義を行うのに，定義される述語がその節の本体にも現すするということにある。

再帰的定義は強力で便利な機能であるが，その用法には十分な注意を払う必要がある。再帰的定義は，数学的帰納法と同様であるから類似的に示せば，まず出発点の定義が必要で，次に再帰的定義の本体部分の定義となる。ここの例では (c4) が出発点の定義（証明の進行から見ると再帰的定義の呼び出しの終りを定義するものだから，終点でもある）で，(c5) が本体部分である。再帰的定義はこのように2つの部分が必要で，出発点を省略すれば証明は果てしなく進行し，ついに異常終了になる。

D. 節の順序づけ

これまで節の順序づけについては，明示的な形では述べてこなかった。しかし節の順序はコンピュータ上での証明を進める手順と効率において重要な役割を持っている。

証明の進行の基本ルールは，ゴール節の左端の句と同じ述語を頭部としてもつ節をパターン照合のため呼び出すことに始まる。このとき呼び出される相手の節が複数あれば，上にあるものから順に選択される。パターン照合が成功す

るまで順次下の節が選択されることになる。計算機による証明の場合，この選択をうまく行うために，**同じ述語を頭部とする複数の節はひとまとめにすることが要請される。**

では，このひとまとまりの節の中での互いの順序はどのようにして決められるのだろうか？　それらがたとえデタラメな順序で並べられていたとしても，バックトラックという機能によって結局は適切な節が選択されることが予想されるが，その場合の効率と副作用あるいは無限ループの可能性から，順序づけは考慮される必要がある。

●効率について：原則は証明の探索空間をできるだけ広げないことだが，それを単純に達成することは初心者にとってはかなり難しいことになる。おおよそ，**簡単なチエックですむものから並べることと，解に到達しそうな節から並べる**という両立しにくい二つの基準に従えばよいであろう[2)]。

●副作用について：データの読取りや印刷などのように，再びもとの状態（読まなかったことにするとか，印刷してもしなかったことにする）に復帰できない動作を起す述語を副作用をもつという。この場合バックトラックしても，もとにはもどらないから**必要な判定をできるだけすましてから選択されるように**，節を順序づける必要がある。

●無限ループ：再帰的定義によるループはその**ループの終点**（つまり再帰的定義の出発点）**の定義をする節を再帰的定義の節の直前に置く**必要がある。これを逆に置くとループの終点を定義する節は評価されないから，無限ループに入りこむことになる。このような単一の節の中の再帰的定義による無限ループは以上のチエックで検出できるが，複数個の節に渡る再帰的な呼び出しは，その呼び出しの論理をよく検討することが必要である。

次に示すような例なら無限ループに入っていると容易にチエックできるだろう：

$$\begin{cases} \text{Parent}\ (x, y) \leftarrow \text{Child}\ (y, x) \\ \text{Child}\ (A, B) \leftarrow \text{Parent}\ (B, A) \end{cases}$$

2) この問題は最も重要な研究テーマであり，Kwalski [Kwa] に詳細な論述がある。

2.　Prolog の利用法

A.　マルセイユ版 Prolog の移殖

述語論理プログラムを実際に使うには，そのために作られた言語プロセッサを用意しなければならない。そのようなプロセッサとして最も古いのはマルセイユ大学で開発された Prolog である。最近，東京大学の中島秀之氏がポータブル Prolog を公開している [Nak]。この中島氏の Prolog は LISP がすでに用意されている計算機ならかなり容易に移殖ができるということである。

以前，著者は法文や条例文などの無矛盾性解析のために，記号論理学の応用を試みたことがある。その際に用いた証明法は節形式による入力導出法 [Nil] [Cha] であった。ところがこれだと，各法文・条文の記号論理式への翻訳結果が，人間にとって実に読みにくいものであったことと，矛盾がないことの証明プロセスを制御することが不可能であることなどが明らかになった。

このような事情があって，ようやく Prolog にめぐりあったのが昨年の4月であった。5月に，筑波の電子技術総合研究所の古川康一氏に移殖の可能性についての問合せを行い，6月には Fortran で書かれたソースリストを受け取ることができた。7月，8月には私のゼミの学生達8名ほどにソースプログラムの入力を手伝ってもらい，9月にはプログラムは完成した。ところが，このプログラムでは入出力符号が ASCII 符号を予定してあって，本学の COSMO 700 S の EBCDIC 符号と異なるものであったために，符号変換を入出力命令の前後で行うようにするなどの作業の後，10月末頃に移殖作業はほぼ完了した[3)]。

マルセイユ版 Prolog を使った感じでは，田島守彦氏の指摘 [Taj] と同様に，
①データベースの内容を見るための，逆インタプリタの必要性
②任意の位置の節の消去ができるようにデータベース機能の拡張
があるとよいと思われる。さらに，基本述語の名前が英語名であればミスパン

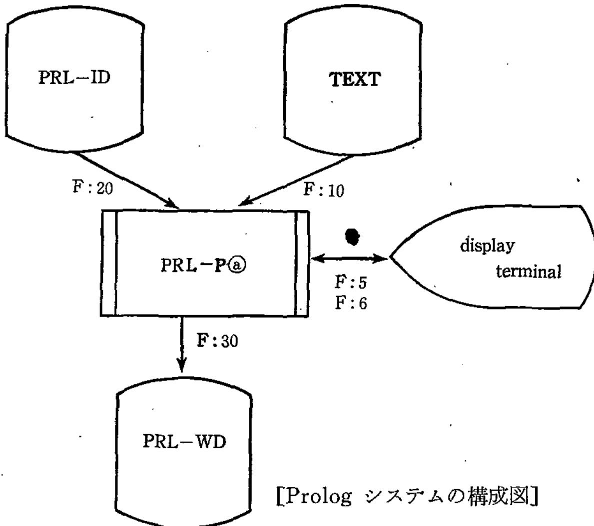
3）この移殖において，私のゼミ学生一同さらに新田将人君（現在富士通㈱）に感謝したい。特に新田君は移殖の作業の全体を一慣して受持ってくれたことを記しておく。

チを防げること，デバックのためにトレース機能が欲しいなどが感じられる。特に教育用に使うにはこのことが重要であろう[4]。研究開発用には，Prolog の出力が TSS 端末以外にもう一つ欲しいこと，また利用者が定義できる基本述語があれば処理の高速化と拡張性に有用であろう。

B. Prolog の起動法

Prolog システムは下図の構成になっている。PRL-P@ はロード・モジュールの Prolog プロセッサである。

PRL-ID は PRL-P@ の実行の直後に読みこまれるシステム生成用のデータファイルで，これはシステムで用意されている。TEXT は利用者がエディタなどで作成した Prolog ソーステキストである。display terminal は利用者が利用している端末装置であり F：5と F：6が既定値となっている。PRL-WD は利用者が Prolog システムの利用を一時中断するが，その後，その中断点から再開できるように現在のシステムの状態を保存するために用いるファイルである。この保存は中断のためのコマンド-SAUVE. によって行なわれる。中断



[Prolog システムの構成図]

4）本学の若林信夫助教授からの指摘があった。

```
! SET    F: 10/TEXT; IN……Prolog ソースは TEXT とした
! SET    F: 20/ PRL-ID
! RUN    PRL-P@
      ≀
***  MARSEILLE PROLOG (COPY 1981)***
? -TTY-BOOLISTE-LIREFICHIER-BOOLISTE.……TEXT
      ≀                                 の内容を読み込むコマンド入力
                              ここに TEXT の内容が表示される
+FIN.                         …TEXT file の最後のレコードは必
                              ず+FIN. である
?     ≀                       ?はシステムからのコマンド入力要求
                              である。
                              ここに prolog コマンドを入れて,
                              Prolog システムを利用する
?-STOP.……Prolog の利用を終わるコマンド
FIN
*STOP*O                   注) イタリック活字はキー入力を示す
```

を再開するには，F : 20 の割付を先に- SAUVE. によって作られた保存用のファイルにすればよい。このときには，システムが用意している PRL-ID は不用である。もちろん，中断するような使い方はせず，いつも TEXT を読み込むことから始めるならば，F : 30 のの割付は不用である。

〔標準的利用法（中間結果を保存しない場合)〕

標準的な利用例を上に示す。

この利用に先立ち，PRL-ID と PRL-P@ を著者のファイルからコピーする必要がある。現在，本学の COSMO 700 の共用ファイルに登録中だが，登録が終了すれば，

```
! SET  F: 10/TEXT; IN    ……F: 20 の割付の既定値は
! PROLOG                     PRL-ID となっている。
      ≀
*** MARSEILLE PROLOG (COPY 1981)***
  以下 Prolog システムを自由に利用する。
```

のように使えるようになる。

C. 構文と基本述語

マルセイユ版 Prolog システム利用のために，構文と基本述語について説明する。構文とはプログラムを書く上で許されている書き方を述べているものである。基本述語はプログラムの中に書かれる述語のうちで，特定の動作をもたらすように前もって定義されている述語のことである。この特定の動作は主に次の4つの働らきをもたらし，その働らきによってシステムの状態が変わることを副作用という。①入出力②算術式と文字式③制御④データベース操作。

(i)〔構文〕

マルセイユ版の利用手引 [Mar] による構文の紹介は初心者には難解であるので，中島氏の方法 [Nak] で説明しよう。

Prolog では，プログラムもデータも同じ形をしていて特に区別しない。それらを**項**と，よぶと，項は次の3種類に分けられる。

①**アトム**：

- 名前（英文字で始まる任意の長さの英数字列，例　ATOM JAL 236)
- 数値（特に正の整数のみ，例 236)
- ストリング（任意の長さの文字列を ” で囲んだもの，例（“ATOM DESU”)

②**変数**：名前に*マークをつけたもの。

③ **複合項**：f を名前あるいは変数，t_1, t_2,……，を項としたとき，f (t_1, t_2, …t_n) の形をしたものが複合項である。f を述語または関数，全体を述語呼出しまたはパターンということもある。

複合項の名前は，いくつかの項を組合せて新しくひとつの項を作っていることに由来がある。これは「構造」ともよばれ，構造をもつ対象を取扱うときに用いられる。特に**リスト**として用いられることが多い。

リストの場合には，複合項 f をドット．で代用する。つまり，A と B を項とすれば，そのリストは，.(A, B) である。このリストは，ひとつの項だから，他の項と再び複合項を作ることもできる。つまり .(.(A, B), C)。このようにリストは，つぎつぎにアトムを鎖のように繋ぐものであり，構造をもつ対

象を有効に処理できるので重要である。

リスト表現を簡略にするために，オペレータ宣言がある。例えば，+. (DG, 5). と宣言すれば，. (*A*,. (*B*, *C*)) は *A*. *B*. *C* と表わせ，. (. (*A*. *B*,) *C*) は(*A*. *B*). *C* と表わすことができるようになっている。

プログラムは節を並べたもので，最後に終りを示すものとして，+FIN. を置いたものである。次に節の表わし方をⅠ章のホーン節に従って (1) 手続，(2) 主張，(3) ゴール節の順に示す。

(1) $+A-B_1-B_2-\cdots\cdots-B_n.$

(2) $+A.$

(3) $-B_1-B_2-\cdots\cdots-B_n.$

頭部となる句は＋で始まり，本体の句は－で始まっている。節の終りはピリオッドである。

ゴール節はプログラムの中には決して書かれない。これは，プログラムがシステム内に読取られた後に，システムからのコマンド入力要求があってから，キー入力されなければ無効である。

［例 B-1］

```
Ⅰ の例 A-2 を Prolog で表現する。
プログラム：+ANIMAL (*X)-DOG (*X).
            +DOG (POCH).
            +FIN.
コマンド  ：-ANIMAL (POCH).
```

［例 B-2］

```
Ⅰ の例 B-1 を Prolog で表現する。
プログラム：
+PARENT (TARO, JIRO).
+PARENT (JIRO, YAMAHIKO).
+PARENT (HANAKO, YAMAHIKO).
+GRANDPARENT (*X, *Z)-PARENT (*X, *Y)-PARENT (*Y, *Z).
+FIN.
コマンド：
-GRANDPARENT (*PARSON, YAMAHIKO).
（注意）1行の長さは69文字である。長い節は何行にも渡って書いてもよい。
```

実は，これらのプログラムの部分は，Prolog システムの構成図で示した TEXT に書き込まれたものである。この TEXT は次の２つの入力コマンドのどちらかでシステムに読込まれる。

TEXT ファイル（プログラム）の読込コマンドみ：

(a) −TTY−BOOLISTE−LIREFICHIER−TTY−BOOLISTE.

(b) −TTY−LIREFICHIER−TTY.

(a) はプログラムの内容を表示し，(b) は表示しない。(b) は表示しないだけに速いのが特徴である。

(ii)〔基本述語〕

さて，TEXT ファイルの入力に使われた，−TTY'−BOOLISTE, あるいは −LIREFICHIER などが基本述語であった。これらは入力のための特定の働らきをするように，システム内に組み込まれているので，組込述語ともよばれている。

○入出力用基本述語

・**−TTY**：入力先を端末にするか，TEXT（F：10 が割当てられているもの）にするかの切換スイッチになっている。反転スイッチになっている。

・**−BOOLISTE**：入力された内容を端末に表示する。反転スイッチになっている。

・**−LIREFICHIER**：現在つながっている入力先から節を＋FIN. になるまで，データベースにつけ加える（入力先は−TTY で決められる)。

節の入力は−LIREFICHIER でやれるが，節以外の情報の入力は次の２つしかない。それも１文字ずしつか入らない。

・**−LU**(*C)：入力口から１文字が *C に読み込まれる。

・**−LUB**(*C)：入力口から読み込まれた空白以外の最初の文字が *C に読み込まれる。

（例）４文字入力する場合

＋READ−LUB (*C 1)−LU(*C 2)−LU(*C 3)−LU(* C 4).

としたとき，－READ. コマンドでの入力が △*ABC* だったら，*C 1＝*A*，*C 2＝*B*，*C 3＝*C*，*C 4＝△ になる。ただし，△は空白のマークとして表示した。

・－**SORTER**(*X)：項 *X の内容が適力な書式で出力バッファに用意される。

・**SORCHA**(“文字列”)：” で囲まれた文字列が，出力バッファに用意される。

・**ECRIT**(*C)：項 *C は1文字である。その1文字が出力バッファに用意される。

・－**LIGNE**：出力バッファに用意された内容が端末上に改行して出力される。

(例)　＋IOTEST－LUB(*C)－LUB(*D)－SORCHA(“INPUT＝”)
　　　－ECRIT(*C)－ECRIT(*D)－LIGNE.

なるプログラムで，－IOTEST. コマンドでの入力が，OK であったら，結果は改行されて，INPUT＝OK と出力されてくる。

・－**SAUVE.**：システムの状態が PRL-WD ファイルに残されて，実行が終る。

・－**STOP.**：システムの実行が終る。

○算術式と文字式の基本述語

・－**PLUS**　(*N, *M, *R)：*N＋*M＝*R が計算される。

(例)　－PLUS(1, 3, *R)－SORTER(*R)－LIGNE.

の結果は，4と端末に出力される。この例はプログラムなしの，コマンドだけの例である。

・－**MOINS**(*N, *M, *R)：*N－*M＝*R が計算される。*N－*M の結果がマイナスになってはいけない。もちろん，*N も *M もマイナスの値は数値として許されない。

・－**MULT**(*N, *M *R)：*N×*M,＝*R が計算される。

・－**DIV**(*N, *M, *R)：[*N÷*M]＝*R が計算される。割算の結果は整数となる。

・**−RESTE**(*N, *M, *R) : *R が *N÷*M の余りとなる。

このような算術に関するパラメータは，正の整数か，変数でなければならない。

・**−INF**(*X, *Y) : *X と *Y は，整数か，一つの文字であること。*X の EBCDIC コードの値が *Y のそれよりも小さいときに，成功と評価される。文字は数値より小さいと定義されている。また文字と文字の大小は，次のようになっている。小から大へと並べる：

空白，.,<, (, +, |, &, !, $, *,), ;; ｢, ;−,／, ,,, %, —, >, ?, :, #, @, ’, =, ”

・**−LETTRE**(*C) : *C が一文字で，アルファベットのとき成功と評価される。

・**−CHIFFRE**(*C) : *C が一文字で，アラビア数字のとき成功と評価される。

```
(例) +WHICH (*C)−LETTRE (*C)−SORCHA ("LETTER")
         −LIGNE.
     +WHICH(*C)−CHIFFRE(*C)−SORCHA("NUMERIC")
         −LIGNE
     +WHICH (*C)−SORCHA("SYMBOL")−LIGNE.
```

のとき WHICH (*C) は文字か数字か記号かに *C を分類して表示する。このプログラムを FORTRAN 的に書くならば，

```
IF (*C が文字) WRITE ("LETTER"); RETURN
IF (*C が数字) WRITE ("NUMERIC"); RETURN
WRITE ("SYMBOL"); RETURN
```

とでもなるであろう。この順に評価されるので節の順序は大切である。

その他，特殊記号を利用するために，*C にそれらの文字を代入してくれる基本述語を表にする。

空白……**−BLANC** (*C)	* ……**−ETOILE** (*C)
, ……**−VIRG** (*C)	(……**−PARG** (*C)
) ……**−PARD** (*C)	” ……**−GUILLEMET** (*C)

○制御

・−/: カット操作を行う。つまり，証明がこの句まで到達して，次の句で失敗したとき，この句を含む節を呼び出した親句を失敗にする。

(例) +CUT−AA−BB.

+AA−SORCHA ("FIRST AA")−LIGNE.

+AA−SORCHA ("SECOND AA")−LIGNE.

+BB−CC−/−FAIL.

+CC−SORCHA ("CUT OPERATION")−LIGNE.

+CC−SORCHA ("NO PASS")−LIGNE.

なるプログラムのとき，−CUT. コマンドの実行結果は次のようになる：

FIRST AA
CUT OPERATION
SECOND AA
CUT OPERATION
?

となる。カット記号−/を通過した直後の句−FAIL が失敗したので，この節+BB−…を呼び出した−BB を失敗にする。−BB の失敗は，−AA のやりなおし，つまり+AA−…の2つの節のうち，第2の節を選ぶことになる。

もし，+BB−CC−FAIL. となっていたら，FAIL での失敗は，CC のやりなおしになり，NO PASS が出力されることになるが，CC のやりなおしがカットされている。これをカット操作とよび，無駄な証明の省略に有効な手段として使われている。

・−/ (*L): 今まで成功してきたゴール節呼出しのうち，*L とパターン照合が成功する最新のものまで，バックトラックする[5]。

(例)

```
          ①   ②
   + a − b − c.
   + b − d.
   + d.
                   ③
   + c − e − /(−b) − FAIL.
```

左の例で，③により，①の句が失敗となる。もし，③が (−c) となっていれば，②の句が失敗され，−/−と同様になる。

その他に，－ANCETRE (*L)：*L は句を表わす項で，最新の句と照合されるもの，あるいは現在の状態を取出してくれる－ETAT (*S) がある。

○データベース操作

ここでのデータベースとは，世によく言われているデータベースシステムのことではない。プログラムとデータとを区別せずに納めている部分を指している。Prolog ではプログラムもデータも区別がないから，データベースを操作するとは，プログラム自体が自分自身を変更できることを意味する。知識工学的に述べれば，新しい知識を取入れながら今までの知識構造を変更するという，メタ知識，あるいはメタ操作ということが可能となる。

・**－AJOUT** (*L)：節 *L をデータベースに附加する。*L は句を意味する項の複合項，つまりリストであること。*L は節と解釈され頭部が同じ節のトップに附加される。

・**－AJOUTB** (*L)：[5]

・**－AJOUTC** (*L)：－AJOUT (*L) は頭部が同じ節のトップに附加されたが，これは最後尾に附加される。

・**－SUPP**(*L)：節 *L をデータベースから削除する。ただし，*L は－AJOUT(*L) の *L と同じく，節を表わす句のリストである。また削除され

```
*** MARSEILL PROLOG(COPY 1981) ***
?-TTY-BOOLISTE-LIREFICHIERE-BOOLISTE-TTY.
+WRITE-SORCHA ("PRINT")-LIGNE.
+OP-AJOUT (.(+(EXEC),.(-(WRITE), NIL)))
     -EXEC
     -SUPP (.(+(EXEC),.(-(WRIT), NIL))).
+FIN.
?-OP.
                        ┌-AJOUT(. (+(EXEC), ·(- (-(WRITE), NIL)))
                        │の実行により，新しい節
                        │+EXEC-WRITE.
PRINT      ......       │がデータベース中して附加され，これを-EXEC コ
? -EXEC.                │マンドで実行したので，+WRITE-…の節によっ
                        └て PRINT と印刷された。
```

5) この機能は現在のところ定義どうりには動作していないようである。移殖中に虫が紛れ込んだからもれない。現在不明である。

? .... ? −*STOP.*	上の−EXEC. コマンドに対する結果表示。これは−EXEC. の失敗を表示している。(−SUPP (…) によって，+EXEC−WRITE. はもうデータベース中に存在しないから。)

るのは，*L で指定された節のトップの節とパターンが一致したときに限られる。だから，任意の位置の節を削除できる訳ではない。

(例) AJOUT と SUPP を使う例を上に示す[6]。

• −***X**: 変数が句と解釈され，その句の実行をする。

• −**UNIV**(*X, *Y): 項 *X の分解が項 *Y となる。

(例) −UNIV (A, *Y)−SORTER (*Y)−LIGNE.

の解は，. (. (A, NIL), NIL)

−UNIV (F (A), *Y)−SORTER (*Y)−LIGNE.

の解は，. (. (F, NIL), . (A, NIL)) となる。

(例) UNIV と−*Xの例を次ページに示す。

• −**ATOME** (*X, *Y) と−**EGALF** (*X, *Y) は [Mar] には記述があるが，その意味と用法が不明である。ほぼ，− **ATOME** は − **UNIV**と同じであり，**EGALF** は *X と *Y が等しいとき成功する。

```
*** MARSEILLE PROLOG (COPY 1981) ***
?−TTY−BOOLISTE−LIREFICHIER−BOOLISTE−TTY.
+. (DG, 5).
+DATA 1 (W. R. I. T. E. NIL). A. NIL).
+DATA 2 (WRITE (B)).
+WRITE (*X)−SORTER (*X)−LIGNE.
+OP 1−DATA 1 (*X)−UNIV (*Y, *X)−*Y.
+OP 2−DATA 2 (*X)−UNIV (*X, *Y)−SORTER (*Y)−LIGNE.
+FIN.
?−OP 1.
A
?−OP 2.
(W.R.I.T.E.NIL).B.NIL
?−STOP.
```

6) 新田将人君の卒業から引用した。

3. Prolog プログラム例題集

A. リストの接続

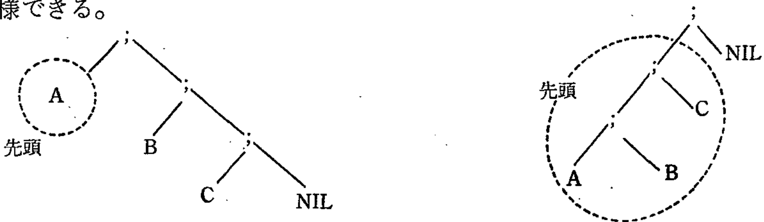
```
+. (DG, 5).
+APPEND (NIL, *X, *X).
+APPEND (*A . *X, *Y, *A . *Z)-APPEND (*X , *Y , *Z).
+FIN.
?-APPEND (APPLE . ORANGE . NIL , LEMON . GRAPE . NIL , *X)
                                                        (改行)
?       -SORTER (*X)-LIGNE.
APPLE . ORANGE . LEMON . GRAPE . NIL
?-APPEND (THIS . IS . NIL , *Y, THIS . IS . A . LIST . NIL) (改行)
        -SORTER (*Y)-LIGNE.
A. LIST. NIL
?-STOP.
```

B. リストの反転

```
+. (DG, 5).
+REVERSE (*LIST, *REVERSED)-REV1 (*LIST, NIL,
                                              *REVERSED).
+REV1(NIL, *RESULT, *RESULT).
+REV1(*E . *REST, *STACK, *RESULT)-REVI (*REST,
                                  *E. *STACK , *RESULT).
+FIN.
?-REVERSE (1 . 2 . 3 . NIL, *X)-SORTER(*X)-LIGNE.
3 . 2 . 1 . NIL
?-REVERSE (*X, A . B . NIL)-SORTER (*X)-LIGNE.
B . A . NIL
?-STOP.
```

C. オペレータ宣言

リストオペレータを ; として A; B; C; NIL なるリストの解釈が次のように様できる。



この違いはオペレータの宣言で行なわれる。これは+ ; (DG, 5). か+ ; (GD, 5). かのオペレータ宣言の違いで出てくる。その例を示す。

```
(ここにオペレーター宣言のどちらかが入る)
+LIST (*W, *X; *Y, *W; *Z)-LIST (*X, *Y, *Z).
+LIST (*X, NIL, *X; NIL).
+CAR (*X)-LIST (*CAR, *CDR, *X)
    -SORCHA ("CAR=")-SORTER (*CAR)-LIGNE.
+FIN.
?-CAR (A; B; C; NIL).
```

としたときの解は，+ ; (DG, 5). のときは，A のみであり，+ ; (GD, 5). のときは，A ; B ; C となる。

もちろん一方の宣言だけで内容的にはどちらの表現も可能である。たとえば，+ ; (DG, 5). の宣言で，右図のリストを表わすには，

((A ; B) ; C) ; NIL

とすればよい。そのときの先頭は (A ; B) ; C である。

D. 事実データを全部表示する

```
+PARENT(JIRO, YAMAHIKO).
+PARENT(TARO, ICHIRO).
+PARENT(TARO, JIRO).
+PRINTPARENT-PARENT (*X, *Y)
    -SORCHA ("PARENT△=△")-SORTER (*X)-SORCHA ("△△△△")
    -SORCHA ("CHILD△=△")-SORTER (*Y)-LIGNE-FAIL.
+FIN.
?-PRINTPARENT.
PARENT=JIRO      CHILD=YAMAHIKO
PARENT=TARO      CHILD=ICHIRO
PARENT=TARO      CHILD=JIRO
?7)
?-STOP.
```

7) +PARENT のすべてのデータを見終っても，FAIL による失敗があって，-PRINTPARENT. のコマンドが失敗したことを示す。しかし出力は副作用をもつので，バクットラック中のすべてのデータは出力される。

8) 確かに-KBIN. で入力したデータが，+DATA (OTARU). としてデータベースにあるか調べる。

E. データのキー入力

キー入力は1文字ずつしか許されていないから，数個の文字をひとつのアトムにまとめる必要がある。その方法を示そう，またこうして得たデータを，+DATA (……). としてデータベースに附加する。

```
+.(DG, 5).
+KBIN-LUB (*C1)-LU (*C2)-LU (*C3)-LU (*C4)-LU (*C5)
   -UNIV (*DATA, (*C1.*C2.*C3.*C4.*C5.NIL). NIL)
   -AJOUT (+(DATA (*DATA)).NIL).
+FIN.
?-KBIN.
?OTARU……データのキー入力
?-DATA (*X)-SORTER (*X)-LIGNE.8)
OTARU
?-STOP.
```

4. 参　考　文　献

[Cha] Chang and Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[Fur] 古川康一，第5世代コンピュータにおける問題解決. 推論機能，数理科学 1982年4月号，サイエンス社。

[Kwa] Kwalski, R, *Logic for Problem Solving*, North-Holland, 1979.

[Man] Manna, Z, *Mathematical Theory of Computation*, Mcgrow-Hill Kogakusha, 1974.

[Mar] PROLOG 利用の手引（他の論文発表の引用では，著者は佐藤㈱エスジーとなっている。これは電総研の古川氏，新田氏より送ってもらったが，手書きによる青焼コピーで，不鮮明なものであった。翻訳されたらしく，文章の意味をたどれない所があり，基本述語の定義が不明なところも，ここに原因している。）

[Nak] 中島秀之，Prolog 入門，bit 1982年4月号共立出版。

[Nil] ニルソン，人工知能，コロナ社，1973.

[Ohs] 大須賀節雄，次世代計算機システムに関する一考察，情報処理 Vol. 21 No. 5 1980.

[Taj] 田島守彦，PROLOG によるゲームプログラムの構成，人工知能と対話技法研究会資料 17，情報処理学会，1980.

[Uch] 内田俊一，第5世代コンピュータのハードウエア，数理科学，1982年4月号，サイエンス社，