RT-11 PROLOG の紹介

中 川 正 宣*

AN INTRODUCTION TO THE RT-11 PROLOG

Masanori NAKAGAWA

The present paper is an introduction for the RT-11 Prolog (Edinburgh version), a logical program language developed by A.I. Department of Edinburgh University, on the LSI-11 mini-computer system in our laboratory. We introduce a small Prolog program as an example, which describe the family relation ship of the cats in my house. In this program, we use English sentences to communicate with the computer system as if the computer understood English. We are now in preparation for the further complicated Prolog program in order to control a psychological experiment using a natural language.

1. は じ め に

そもそも PROLOG とは第一階述語論理 (特に Horn 節) の証明手続 (特に SNL; Selective negative linear resolution)¹⁾ をプログラミング言語化したものであるが, ちょうど自動車の機械的構造やその理論的背景を熟知していなくとも運転することはできるように, 論理学の知識がなければ PROLOG が使えないということはない。

単に一つのプログラミング言語と考えるとマニュアル²⁾ 片手に気軽にミニコンを操作しているうちにある程度までは使えるようになるものである。しかも自動車の場合と違って, 少々ミスをしたところで操作している人も含めてそのハードウェアには何ら支障はない。まして人身事故で逮捕されるような心配はない (パソコン用英文ワープロの代表的ソフトである WORD-STAR のマニュアルには, 警官に拉致されるユーザーの写真があり, その上にわざわざ「どんなエラーを犯してもこんなことになる心配はありません」との説明が入っているが)。

以下で当研究室のミニコン LSI 11/2 の RT11 上で稼動するエジンバラ版 PROLOG の簡単なプログラム例を紹介しよう (もちろんこのプログラムを走らせるために多くのエラーを出したが逮捕されるということとはなかったようである)。

2. Rule と Fact

PROLOG はその各単位が, Rule と呼ばれるある規則の定義と Fact と呼ばれる一種の事実の宣言とから成り立っている。Rule と Fact を合わせて Clause (節) と呼ぶが, PROLOG のプログラムとは単にこれらの Clause の集合であるともいえる。たとえば以下の例 1 での, 「sister (X, Y): -female (X), parents (Y, M, F), parents (X, M, F), X/=Y.」等が Rule であり, 「male (camus).」, 「female (dossy).」等が Fact である。

Rule はある関係や事実を一般的に定義するのに用いられる。たとえば「甲は乙の姉か妹である。」という関係を論理的に定義する場合を考えてみよう。この場合, 「甲が女性でありかつ乙の両親が甲の両親と同じであれば, 甲は乙の姉か妹である。」と定義するのが一つの方法であろう。この時「甲は乙の姉か妹である。」を「姉妹 (甲, 乙).」と略記し, 「甲は女性である。」を「女性 (甲).」, 「甲の両親は丙と丁である。」を「両親 (甲, 丙, 丁).」と略記すると, 上記の定義は「女性 (甲) か

* 北海道大学文学部行動科学科 (Department of Behavioral Science, Faculty of Letters, University of Hokkaido)

1) 長尾, 淵 (1983) 参照

2) Clocksin, Mellish (1981) 参照

```

father(X, Y) : -parents(Y, X).
mother(X, Y) : -parents(Y, X, _).
sister(X, Y) : -female(X), parents(Y, M, F), parents(X, M, F), X \= Y.
brother(X, Y) : -male(X), parents(Y, M, F), parents(X, M, F), X \= Y.
male(camus).
male(calif).
male(patch).
male(maro).
male('frosti-lad').
male(aro).
female(akane).
female(bannette).
female('jo-jo').
female(dossey).
female(dasha).
parents(camus, 'jo-jo', maro).
parents(calif, 'jo-jo', maro).
parents(patch, 'jo-jo', maro).
parents(dossey, 'jo-jo', maro).
parents('jo-jo', bannette, aro).
parents(maro, akane, 'frosti-lad').
parents(dasha, 'jo-jo', maro).

```

例 1

つ両親 (乙, 丙, 丁) かつ両親 (甲, 丙, 丁) ならば姉妹 (甲, 乙).」と表現される。ここで「かつ」を「,」で、「ならば」を「-:」で表現し、左右を入れかえると以下ようになる。

「姉妹 (甲, 乙) : 一女性 (甲), 両親 (乙, 丙, 丁), 両親 (甲, 丙, 丁).」

これは「姉妹」を「sister」に、両親を「parents」に、「女性」を「female」に、「甲, 乙, 丙, 丁」, を各々「X, Y, M, F」, におきかえれば、そのまま前記 PROLOG のプログラムの Rule の一部になっている³⁾。ただし上記の規則では、甲が女性である場合、甲自身も甲の姉妹と見なされてしまう。そこで実際のプログラムでは「X は Y とは異なる (X \= Y).」という規則が付け加えられている。

ここで、甲, 乙, 丙, 丁 (実際の PROLOG では X, Y, M, F) とともに、ある特別の対象を示しているわけではなく、任意の対象を想定していることに注意されたい。つまり、甲, 乙, 丙, 丁は一種の変数 (変項) として使われているわけである。

一般に PROLOG の Rule は「 $P : -Q_1, Q_2, Q_3, \dots, Q_n$ 」のような形式を持っており、「 P が成立するためには $Q_1, Q_2, Q_3, \dots, Q_n$ が同時に成立しなければなら

ない」こと、あるいは「 P は『 Q_1 かつ Q_2 かつ Q_3 かつ... かつ Q_n 』によって定義される」ことを意味している。

ここで P は Rule の head と呼ばれ、 $Q_1 \sim Q_n$ は body と呼ばれる。head は変数を含んでいてもよいし、また body はその head に含まれる変数を共有していてもよい (前記 sister での X, Y など)。

一方、先程の「女性 (甲)」 (female (X)) や両親 (甲, 丙, 丁) (parents (X, M, F)) の甲, 丙, 丁 (X, M, F) の部分を具体的な対象の名前で置きかえたものが、PROLOG での Fact と呼ばれる節である。つまり前記の female (dossey) は「dossey is female.」という具体的事実を宣言していることになる⁴⁾。

3. PROLOG の動き (Camus の母をたずねて)

ではこのような Rule や Fact の集りがあったとして、実際 PROLOG はどのように働くのであろうか。今、例 1 の Rule や Fact が既に入力されているものとしよう。ここで mother の定義を言語に直してみると「X が Y の mother であるとは、Y がその第一項にある

3) 実際に漢字表記の PROLOG も考案されている。

4) Rule や Fact における形式 $A(X)$ は Predicate (述語) と呼ばれるが、これをある具体的な対象の集合から「X は A である」という文への関数であると考えられることもできる。この意味で、形式 $A(X)$ の A を functor と呼ぶ。

parents で、第二項がXであること」と表現できよう。

しかし、既に述べたようにXやYは変数であって、そのまま実際の対象を示しているわけではない。そこでこの「mother」なる Rule を厳密に説明すると、冗長ではあるが以下のように表現されよう。「Xの中身がYの中身の mother であるとは、Yの中身がその第一項を占める parents という Fact の第二項がXの中身であることである。」

今、このような Rule の意味を念頭において、Yに「camus」という対象名が代入されている時、上記の Rule や Fact を使って「camus」の母親(名)⁵⁾を探索する事態を考えてみよう。この場合、探索対象であるXという変数の中身は未定であるとする。この時 mother という Rule を使ってこの Rule の body におけるYの項を全て「camus」に置き換えると、問題は「parents (camus, A, B) という Fact でAの位置(第二項)にある対象は何か。」と読み換えることができる。

さしづめ人間ならこの問題の答を見つけるために上記の例1の Fact の集合をながめ、parents (camus, A, B)と宣言されている Fact を上から順に捜してその時のAを読み取るであろう。しかもそれを見つけたところで、「それはjo-joです。」と答えてくれるであろう。PROLOGは実はそのような人間の探索法とある意味では同等の方法で対象を捜し、かつ答えてくれるのである。

この場合の PROLOG の動きを列挙すると以下のようになる。PROLOGは一般にインタープリターによって処理されているから、まずシステム起動直後は入力待ち状態(質問待ち状態)になっている。入力待ち状態では、プロンプティング記号(入力促し記号)として「?-」が出力されている。そこで例1のプログラムが既に入力されているものとして、「mother (X, camus).」と入力する。これは「camusのお母さんは誰ですか?」と質問していることに相当する。そうすると、システムはプログラムの中で mother という head を持つ Rule か、もしくは mother という Fact を探索しはじめる(この時、探索の対象になる Rule の head や Fact を goal と呼ぶ)。

もし今のプログラム例とは異なり、「mother (fox, camus).」という Fact が存在すれば、質問の functor である「mother」とこの Fact の functor である

「mother」が同一であることが確認され(これをパターンマッチングと呼ぶ)かつ質問の「camus」と Fact の「camus」のマッチングが行われると同時に、Xに「fox」が代入されて「X=fox」と出力される⁶⁾(つまり「それは fox です。」と答えてくることに相当する)。

例1のプログラムの場合のように「mother (X, Y)」という head を持つ Rule が見つかり、head のYに「camus」が代入され、それが body でのYにも適用されて、parents (Y, X,)⁷⁾は parents (camus, X,)と変化する。そこでシステムは parents (camus, A, B)という形の Fact を順にマッチングし、最初の parents (camus, A, B) で宣言されている対象A(ここではjo-jo)をXに代入し、この探索は終了する。さらに mother (X, camus) のXには、この「jo-jo」が代入され、この Rule は「探索対象が見つかった。」という意味で、「成功した。」と称せられる。この時は Rule「mother」は成功し、「X=jo-jo」と出力応答されて、めでたしめでたしとなるわけである。

では「?- sister (X, camus).」という質問の場合はどうであろうか。この場合は Rule でのYには camus が代入され、まず body の female(X)を満たすXが見つけれられる。ここでプログラム中の並び順に従って、最初に「akane」がマッチする。そこで次に parents (camus, M, F)で M, F が探索される。これは parents (camus, jo-jo, maro) という Fact とマッチングすることで、Mには「jo-jo」がFには「maro」が代入される。次に、parents (X, Y, F)のXには「akane」が、M, F には各々「jo-jo」「maro」が置き換わり、parents (akane, jo-jo, maro)の形式の Fact が存在するかどうか check される。もちろんここではそのような Fact は存在しない(そもそも akane は maro の母親である)。よって、この Y, M, F の代入は「失敗」だったわけである。しかしながらこんなことでシステムはあきらめない。さらに実行を逆もどりして(これをバックトラックと呼ぶ)、parents (camus, M, F)を満たす別の Fact が探索される⁸⁾が、このプログラムではそのような Fact はさっきの一つしかない(camusの両

6) 実際はRT11システムでは「** (top) PROVED: mother (fox, camus)?」と出力される。

7) 「 」はどのような変数の内容ともマッチングするダミー変数である。

8) このような場合、各 Fact や Rule について既にどこまで探索されたかがポインターに記憶されている。

5) 以下では対象と対象名を同一視することにする。

親は一対しかいない!), だからその探索も失敗するわけだがシステムはまだまだあきらめない。さらにバックトラックして `female(X)` を満たす Fact が探索される。

このようにして最終的に「`X=dossy`」の時, `female(dossy)`, `parents(camus, jo-jo, maro)`, `parents(dossy, jo-jo, maro)`, `dossy/=camus` が全て満たされ, 「`X=dossy`」と応答される。ここで今出力された「`X=dossy`」の直後に「;」を入力すると, 今探索された次の Fact から探索が続行される。ここでは「`female(dossy)`」の次の「`female(dasha)`」がマッチングし, 「`Y=dasha`」と応答してくれる。

質問のしかたとしては, 「`?-sister(dasha, Y).`」と入力することも可能である。その場合には当然「`Y=camus`」と出力され, さらに「;」を入力することで, 順に「`Y=calif`」, 「`Y=patch`」と出力される⁹⁾。また「`?-sister(dasha, camus).`」という質問に対しては, この Rule が成功すれば「yes」, そうでなければ「no」と応答される(たとえば「`?-sister(fox, camus).`」という質問には「no」と答えられよう)。

しかしここで「no」と答えられた場合, それは単に今存在しているプログラムで相当する事実がなかったことを意味しているにすぎない。つまりプログラムに Fact が付け足されれば「yes」となる可能性もあるわけである。実際, このプログラムに「`female(fox).`」, 「`parents(fox, jo-jo, maro).`」という二つの Fact を付け足すだけで, 上記の質問「`?-sister(fox, camus).`」には「yes」という答が返って来る。

このように PROLOG ではプログラムを変更することなく容易に関連する Fact を増やしてゆくことができる。また既存の Rule をもとにして, 様々な形で新しい Rule を定義してゆくことも可能である。

たとえば `grandmother` という Rule は前記の Rule を使って以下の様に定義できる。

```
grandmother(X, Y): -mother(X, M),
mother(M, Y).
```

```
grandmother(X, Y): -mother(X, F), father(F, Y).
```

9) このように PROLOG では `R(X, Y)` という一つの宣言 (Fact や Rule) で, 「`Y` に対応する `X` を求める」という手続 (機能) と 「`X` に対応する `Y` を求める」という手続の二つを同時に満たすことができる (一般に変数の数に応じてこのような手続数は増えてゆく)。

この Rule の意味は説明するまでもないであろう。更に今 Fact として宣言されている事実を一つの Rule として定義し直すことも可能である。

たとえば, `male` や `female` を Rule として定義することもできよう (この場合, どのような事実を組み合わせればよいか, またどのような Rule を付け足せばよいのかには, 厳密には生物学の高度な知識が必要となろう)。

しかもこのようにして新たに Rule を付け足す際に, このプログラムには何の変更も必要としないのである。male という Rule が新たに付け足される場合に「`male(camus)`」等の Fact が同時に存在していても, プログラムの働きの上からは何の問題も生じない。

PROLOG は, 既存のプログラムからさらに階層的に複雑な構造を持つプログラムを構成する上でも, このように誠に都合よくできたシステムであると言える。これは, コンピュータ上に自己増殖システム, あるいは学習システムを構成する場合には極めて重要な機能を発揮する特長であろう。

4. プログラムからの入出力とリスト処理 (Do you understand me?)

もし PROLOG の入出力が前記の質問と応答の形式に限られているとしたら, いかに複雑な構造を持つプログラムを作ったところでその機能は限られていよう。もちろん PROLOG ではシステムとしてのインタープリターとは別にそのプログラム内で入出力を定義することができる。最も基本的な入出力のための組み込み機能¹⁰⁾は「`read`」と「`write`」であるが, さらにここでは便宜上, 「`read-in`」という predicate も既にシステムに組み込まれているものと仮定しよう¹¹⁾。

`read(X)` が実行されると入力待ち状態になる。そこで何らかの方法 (RT-11 ならコンソールターミナルからのキー入力) で文字列を入力し, 最後に「`.`」を入力するとその文字列が `X` に代入される。逆にその様にして入力された `X` に対して `write(X)` を実行すると, `X` の中身が文字列として出力される。また `write('hello!')` 等を実行し, 引用符ではさまれた文字列を「hello!」という風に出力することも可能である。

しかし `read` では単に文字列を一つのかたまりとし

10) PROLOG では built-in predicate と呼ばれる。

11) 「`read-in`」は実際は一つの PROLOG の Rule として定義される (付録参照)。

```

do: -repeat,nl,nl,write('Have you any questions about the family of camus?'),nl,nl,read_in(S),dt(S).
dt([stop,'.']).
dt(S): -act(S).
act(S): -select(X,Y,D,S),set(X,Y,Z,D).
select(X,_,love,[who,loves,X,'?']): !.
select(X,Y,who,[who,is,the,X,of,Y,'?']): -!.
select(,_,,): -nl,write('I don't understand what you are saying!'),!,fail.
set(X,Y,Z,D): -D=who,W=...[X,Z,Y],call(W),pr(X,Y,Z,D),fail.
set(X,Y,Z,D): -D=love,love(Y,X),pr(X,Y,_,D),fail.
pr(X,Y,Z,who): -nl,write(Z),write('is the'),write(X),write('of'),write(Y),write('.'),!.
pr(X,Y,_,love): -nl,write(Y),write('loves'),write(X),write('.'),!.
love(X,Y): -male(X),female(Y).
love(X,Y): -female(X),male(Y).
father(X,Y): -parents(Y,_,X).
mother(X,Y): -parents(Y,X,_).
sister(X,Y): -female(X),parents(Y,M,F),parents(X,M,F),X\=Y.
brother(X,Y): -male(X),parents(Y,M,F),parents(X,M,F),X\=Y.
male(camus).
male(calif).
male(patch).
male(maro).
male('frosti-lad').
male(aro).
female(akane).
female(bannette).
female('jo-jo').
female(dossy).
female(dasha).
parents(camus,'jo-jo',maro).
parents(calif,'jo-jo',maro).
parents(patch,'jo-jo',maro).
parents(dossy,'jo-jo',maro).
parents('jo-jo',bannette,aro).
parents(maro,akane,'frosti-lad').
parents(dasha,'jo-jo',maro).

```

例2

て入力するだけであって、何らかの構造を持った記号列(たとえば英語の文章、あるいは PROLOG のプログラム)を入力後、構造的に処理するには不便である。

PROLOG ではこのような構造を持った記号列の処理のために便利な方法として、リストという形式が用意されている。リストは一般に $[a, b, c, \dots]$ という形をしている。この場合 a や b がまたリストであることも可能であるし、 $[\]$ も空リストとしてリストに含まれる。

「read-in(X)」という predicate は入力された文字列を空白を区切りとしてこのリスト形式に変換してくれる。この場合「.」、「?」、「!」が終了記号と見なされる。たとえば「read-in(X)」が実行されて入力待ち状態の時に「who is the mother of camus?」と入力

すると X には、 $[who, is, the, mother, of, camus, ?]$ というリストが構成される。PROLOG では文字列をいったんこのリスト形式に変換することで、プログラム内で容易に処理することができる。以下では実際に例2のプログラムにそってリスト処理を説明しよう。

まず、システムの入力待ち状態(プロンプティング「?-」が出力されている状態)で「?-do.」と入力する。これはプログラム処理上一つの質問として受け付けられ、システムは「do」なる head を持つ Rule か、Fact を探しはじめる。そこで「do; -repeat, nl, nl, write('Have you any questions about the family of camus?'), nl, nl, read_in(S), dt(S).」という Rule が見つかり、その各 goal が順に実行される。その際の動きの詳細は後に述べるとして、ここでは read-in

(S) により S 内に先の要領で [who, is, the, mother, of, camus, '?'] というリストが構成されたものとする。そこで select (X, Y, D, S) が実行されると, S に上記のリストが代入される形でプログラム中の select なる Rule の head が順に探索される。ここでは上から 6 番目の Rule「select (X, Y, who, [who, is, the, X, of, Y, '?']): -!」がマッチングされ, D に「who」が, X には「mother」, Y には「camus」が代入される。

このように変数(ここでは X, Y)をリストの一部に含むことによって, 他のリストをマッチングした際にその変数が対応する部分(ここでは mother と camus)を変数の値として代入することができる。この機能は条件に応じてリストから特定の部位を取り出し処理する場合には極めて便利である(このことは, 人間がリストを処理する場合, 条件に応じて必要な部分だけを抽出してくる場合に比較すると極めて興味深い)。

X, Y, D に各々値が代入されたところで次に set (X, Y, Z, D) が実行される。ところでプログラム中には, set という Rule は「D=who」という predicate を含むものと「D=love」を含むものの二つがある。システムは今 D に代入されている値に応じて, 「D=who」か「D=love」かをこの部分で識別するわけである。つまり「D=who」であればその Rule が, また「D=love」であればそちらの Rule が継続して実行される。ここでは先の select によって D には「who」が代入されているから「D=who」が正しいと判断され, 「D=who」を含む set が続けて実行される。

つまり「D=who」の次の「W=..[X, Z, Y]」が実行されるわけだが, この goal は W という変数の中に「X (Z, Y)」という形の goal を形成することを意味している。こうしておいて call (W)を実行すると, X (Z, Y) が実行される。たとえばここでは X には先の select により「mother」が代入されているわけだから, W には「mother (Z, Y)」という文が代入され, call (W) で実際には mother (Z, Y) を実行することになる。

この方法を用いれば X という変数の内容に応じて任意の Rule や Fact を指定することができる。つまり, 今までの変数(たとえばここでの Z や Y)は単にその内容として具体的対象(たとえば「maro」や「camus」)を予想し, 一般的な形で Rule が構成されたわけであるが(たとえば sister (X, Y) や father (X, Y) 等), この方法を用いると Rule や Fact の functor そのものを変数として扱うことができる。このことは, 今ま

での Rule が単に変数を用いて一般化された具体的対象の関係に基づいて定義されていたのに対して, その関係そのものを対象化していることを意味している¹²⁾。

さらにこのことは上記の構造的意味合いからだけでなく, 機能的な側面から言っても大変重要な意味を持っている。たとえば 3. の最後の例のように grandmother という Rule がこのプログラムに付け足された時, このプログラムでは select 及び set 共に何の変更も必要としない。「who is the grandmother of camus?」という入力に対して, select で X に「grandmother」が代入され, set で「D=who」と判別された後, W には「grandmother (X, camus)」という文が代入され, call でそれが実行される。そうして最終的に「bannette is the grandmother of camus.」と出力してくれる。

つまりこの形式は X (Z, Y) で表現される head をもつ Rule や Fact (理論的には無限個)に普遍的に機能することができるわけである。

mother (X, camus) が呼び出されてから mother を head とする Rule での動きは 3. で述べた通りである。

5. 反復とカット

set で call (W)が実行され, 実際には W の中身が実行されると, 変数 X, Y, Z, D, にはそこまでの実行の結果として各々「mother」, 「camus」, 「jo-jo」, 「who」が代入されている。そこで prr (X, Y, Z, D) が実行されると, プログラム中 D の部分が「who」になっている prr がマッチングされ, その中での write predicate に従って, 「jo-jo is a mother of camus.」と出力される。

問題はその後である。この後には「fail」という predicate が実行されるわけだが, この predicate は常に失敗する。そうするとシステムはバックトラックをしはじめる。つまり逆もどりをしながら各 goal を実行して行く。ただし, バックトラックではシステム組み込みの入出力 predicate は全て飛ばして実行される(実際バックトラックの時にまたまた入出力が実行されてはおかしくなってしまう)。

12) このことは実は PROLOG が一階論理だけでなく, 高階論理の操作をも含む得ることを意味している。

.RUN PROLOG

Prolog-11

for LSI-11 with EIS

at E302

?-[q×0:cat'].

qx0: cat consulted.

yes

?-do.

Have you any questions about the family of camus?

Who is the mother of camus?

jo-jo is the mother of camus.

Have you any questions about the family of camus?

Who is the sister of camus?

dossy is the sister of camus.

dasha is the sister of camus.

Have you any questions about the family of camus?

Who loves dossy?

camus loves dossy.

calif loves dossy.

patch loves dossy.

maro loves dossy.

frosti-lad loves dossy.

aro loves dossy.

Have you any questions about the family of camus?

Who is the brother of calif?

camus is the brother of calif.

patch is the brother of calif.

Have you any questions about the family of camus?

Who loves calif?

akane loves calif.

bannette loves calif.

jo-jo loves calif.

dossy loves calif.

dasha loves calif.

Have you any questions about the family of camus?

stop.

yes

?-~Z

例 3

この時、組み込みでない入出力はどうすればよいだろうか。そのような場合のためにカット「!」という predicate が組み込みで用意されている。たとえば今の場合、バックトラックして prr が再び実行されるようにするわけだが、prr の最後には「!」が付けられているため、バックトラックの際には無条件で実行は失敗する（つまり飛ばして実行されるのと同じ結果になる）。

このように「!」を使うと、バックトラックで実行されると都合の悪い部分を文字通り「カット」することができる。一般的には「 $P: \neg Q_1, Q_2, !, Q_3, Q_4$ 」という Rule があって、 Q_4 で実行が失敗しバックトラッ

クが始まった場合、 Q_2, Q_4 は実行されない。またこの時 P を含むような Rule（たとえば $R: \neg S, P, T$ ）という Rule の T からバックトラックが始まった場合、 P は実行されない（ S は実行される）。

ここでは camus の mother は jo-jo だけであるから（実際には camus には別に foster mother がいるのだが）、バックトラックは全て失敗し、Rule「do: -」の中の repeat（この predicate は常に成功するが何も実行しない）から実行が再開されて、read-in で入力待ち状態となる。

そこで mother についてではなく「who is the sister of camus?」という入力を行うと、まず「dossy」を見

```

read_in([ w, ws]): -get0(c),readword(c,w,c1),restsent(w,c1,ws).
restsent(w,[ ]): -lastword(w),!.
restsent(w,c,[ w1!_ws]): readword(c,w1,c1),restsent(w1,c1,ws).
readword(c,w,c1): -single_character(c),!,name(w,[_c]),get0(c1).
readword(c,w,c2): -
    in_word(c,newc),!,
    get0(c1),restword(c1,cs,c2),name(w,[_newc|cs]).
readword(c,w,c2): -get0(c1),readword(c1,w,c2).
restword(c,[_newc|_cs],c2): -
    in_word(c,newc),!,
    get0(c1),
    restword(c1,cs,c2).
restword(c,[ ],c).
single_character(44).
single_character(59).
single_character(58).
single_character(63).
single_character(33).
single_character(46).
in_word(c,c): -c>96,c<123.
in_word(c,1): -c>64,c<91,1 is c+32.
in_word(39,39).
in_word(45,45).
lastword('.').
lastword('!').
lastword('?').

```

付録

つけて「dossy is the sister of camus.」と出力した後、「fail」でバックトラックし、次に「dasha」を見つけて「dasha is the sister of camus.」と出力する。

この後更に「fail」でバックトラックが行われるが、プログラムには dasha 以降 camus の sister は記述されていないわけで(現実には山ほどいるのだが)、バックトラックで do の Rule の中まで戻って repeat で read-in の待ち状態となる。つまり「repeat」と「fail」の間は無限に反復されるわけである。

実際には無限に反復されて困るので、read-in での入力に応じてこのプログラムの実行を終了してくれるのが望ましい。そこで *dt* (S) という Rule を do の最後に付け加えて、S が [stop, '.'] であれば Fact 「dt ([stop, '.'])」がマッチし、そのまま何も実行しないでプログラムが終了することになる。もちろん S が [stop, '.'] 以外のリストであれば、「dt(S): -act (S)」がマッチし、今まで述べて来たように act (S) 以降が実行され続けるわけである¹³⁾。

6. 実行結果

最後に前記のプログラム例2の実行例を挙げておこう(例3参照)。例3でのアンダーラインの部分が入力行である。RT-11では「RUN 文件名」で指定されたファイルのオブジェクトモジュールが実行される(例3の「PROLOG」はPROLOGシステムのオブジェクトモジュールのファイル名である)。「?-[gxo: cat].」では例2のプログラムの入った「gxo: cat」というファイルを入力している。その後、先に述べたように、「?-do.」の入力で例2のプログラムが実行される。

「love」というRuleの働きや無意味な文(あくまでこのプログラムにとって)が入力された時の出力については例2のプログラムから類推していただきたい。

もちろんPROLOGはここに述べて来た以上の様々な機能(プログラム内でのファイルの入出力、あるいはプログラム内容の変更等)を含んでいるが、それらの機能を駆使することによって将来心理学あるいは行動科学にとって十分意義のある、かつ今までの方法で

13) PROLOGにおけるこのような判別、分岐、反復動作の一般的手続型言語(たとえばPASCAL)との対応関係については長尾、淵(1983)、及び

横井、淵(1982)に詳しい。

は困難であったようなモデルや実験制御のプログラムが構成できるものと期待されよう(たとえば, 自然言語による応答システムをベースにした, 複雑な階層構造を持つ概念の学習等).

当研究室(基礎行動学研究室)では, 現在このような実験制御用 PROLOG プログラムを準備中である. そのような実験制御の具体的プログラムが実現した段階で, 新たにその内容を紹介したいと希望している.

参 考 文 献

- Clocksin, W.F., Mellish, C.S. (1981) *Programming in Prolog*, Springer-Verlag, New York.
- 長尾 真, 淵 一博(1983)「論理と意味」岩波講座情報科学 7
- 横井俊夫, 淵 一博(1982)「推論機講を内蔵した述語論理型言語 Prolog」日経エレクトロニクス 9月27日号 (No. 300) pp. 267-286.

—1983年6月 受付—