知識工学と人工知能 57 − 6
（１９８８. ３. １０）

Blackboard in Prolog

赤間 清　　滝川 雅巳

（北海道大学　文学部　行動科学科）

大規模なプログラムにおいては，いくつかの情報をアクセスするサブルーチンの種類が多い場合や，サブルーチンの呼出し場所の種類が多い場合には，各サブルーチンから直接にアクセスできる記憶機構が重要になる．ここではそれを黒板と呼ぶ．

節を assert や retract で操作すれば，Prolog において黒板のいくつかの機能を実現できるが，節の中で変数を正しく表現できない．変数で表すのをやめてアトムなどを使って対象を表せばこの問題は避けることができるが，それでは対象の一致をユニフィケーションで表現できないため，Prolog の最大の利点を放棄することになる．

本論文では，継承階層 prolog：PAL に導入されている黒板について記述する．そして黒板が，自然言語の意味処理などにおいて必須の役割を果すことを意味処理の例を用いて示す．

Blackboard in Prolog

Kiyoshi AKAMA & Masami TAKIKAWA

(Faculty of Letters, Hokkaido University, Sappro-shi, 060, Japan)

In large programs where there are subroutines that must access various types of data or where the subroutines are called from many different places, it is important to have memory area that can be accessed directly from each subroutine. Here we call such a memory area a "blackboad".

Using assert and retract to store and delete clauses or facts in the usual Prolog system, we can perform some of the functions of a blackboard, but it has serious problems in dealing with variables, because when assert is called, unbound variables are not correctly represented in the new clause.

One way to avoid this is to use atoms rather than variables, but it makes the use of subsequent unification impossible, which is the best advantage of Prolog.

In this paper we will describe the blackboard mechanism in the extended Prolog PAL, and explain how the blackboard plays an important role in the semantic processing of natural language.

1. まえがき

大規模なプログラムをつくるにあたっては，いくつかの情報をアクセスするサブルーチンの種類が多い場合や，サブルーチンの呼出し場所の種類が多い場合には，各サブルーチンから直接にアクセスできる記憶機構が重要になる．本論文ではそれを黒板と呼ぶ．

節を assert や retract で操作すれば，Prolog において黒板のいくつかの機能を実現できるが，節の中で変数を正しく表現できない．変数で表すのをやめてアトムなどを使って対象を表せばこの問題は避けることができるが，それでは対象の一致をユニフィケーションで表現できないため，Prolog の最大の利点を放棄することになる．

本論文では，継承階層 prolog：PAL に導入されている黒板について記述する．そして黒板が，自然言語の意味処理などにおいて必須の役割を果すことを例を用いて示す．意味処理の例は，理解から生成までの基礎的な部分をひととおり扱うことのできる自然言語処理実験システムTALK[3]が行なうものを用いる．

2. Prolog への黒板の導入

2．1 黒板の必要性

ある程度以上の規模のプログラムを書くには，全体の情報処理を適当な区切りで分割して，いくつかの部分的な処理のプログラム（サブルーチン）をつくり，それらをうまく組み上げることが行なわれる．そのとき，各サブルーチンの入出力情報は，サブルーチンの呼出しにそって受渡しされるのが基本である．しかし情報をアクセスするサブルーチンの種類が多い場合や，サブルーチンの呼出し場所の種類が多い場合には，各サブルーチンから直接にアクセスできる記憶機構が重要になる．ここではそのような記憶機構を黒板と呼ぶことにする．

もちろん局所変数を持ち歩けば，黒板を使うのと等価な情報処理を原理的には実現できるが，それはプログラムを非常に繁雑にする可能性が高い．たとえば Prolog でそれをやるには，すべての述語に２つの引数を追加する必要がある．２つの引数のうち，１つは黒板のはじめの状態を表す引数，もう１つは黒板の終りの状態を表す引数である．述語への引数の追加が必要なのは，直接その引数を更新，あるいは，利用している述語だけではない．たとえば，図１の述語４と述語７で黒板を更新利用したい場合を考えると，述語４で更新した結果は述語７まで引渡す必要があるために，１から７までの述語すべてに２つの引数を追加しなければならない．さらに黒板に書込んで各述語から共通にアクセスしたい情報の種類はいろいろありうるので，知的情報処理をやろうとする場合などでは，２引数の追加の方法は非常に繁雑なプログラムを導いてしまうことになる．

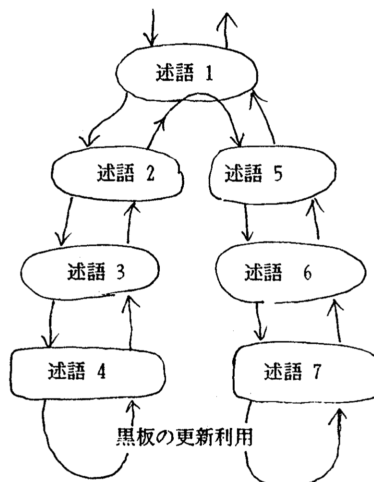

図１ サブルーチンの呼出し

2．2 Prolog の問題点

いくつかのプログラム言語では，黒板はごくあたりまえの機構として実現される．たとえば，FORTRAN の common 変数 や lisp の大域変数などがそれである．しかしながら Prolog の変数はすべて局所変数なので，それらと同じ方法は使えない．Prolog ではそのかわりに述語が用いられる．太郎の年が１２才だという情報は，たとえば，

```
(assert (age taro 12))
```

というように宣言され，通常の述語呼出しで利用され，

```
(retract (age taro ?))
```

で取消される．これは述語であるから，各サブルーチンから直接アクセスすることが達成されている．

しかしいくつかの問題が考えられる．１つ目は assert や retract などは副作用として実現されているので，backtrack してももとに戻らないことである．しかしこれは組込述語 onbt （on backtracking) を用いるなどして backtrack 時の処理を追加すれば一応解決できる．２つ目の問題は，推論速度である． assert や retract は対象のコピーを引起こすので効率が悪くなる．最も強調したいのが３つ目の問題である．assert は，情報の中に入っているかもしれない変数の環境を断ち切ってしまうことである．たとえば，「ある男 *x がある女 *y を愛している」という情報：

```
(love *x^man *y^woman)
```

をassert を用いて

(assert (love *x^man *y^woman))

というように登録しても，データベースに新しく作られるファクトに含まれる変数は，assert を呼出す述語中で指し示している *x や *y （そこでは *x は man であるだけでなく，名前が太郎だということも知られていたかもしれない）とはまったく異なる変数となる．そしてそのファクトを普通の述語呼出しで使うならば，「すべての男がすべての女を愛する」という意味になってしまう．

節の中の変数は完全に局所的であり，節の中の変数と述語実行中の変数との同一性は表現できない．つまり情報の意味をかえたくなければ，変数が含まれる情報を assert を用いて宣言してはならないのである．残された方法は，変数を含まない形に情報を変換することである．したがってたとえば，*x^man のかわりに man#1 を，また，*y^woman のかわりに woman#2 を扱うことになる．しかしこれでは変数やクラス束縛変数の持つ利点を失うことになる．変数やクラス束縛変数を使うことは，Prolog や PAL が lisp などに優越する大きな利点の１つであるが，この扱いはそれを放棄するものである．

２．３　Prolog-PALの黒板

Prolog：PAL には黒板が導入されている．黒板は，その名前と，情報の有限列からなる．名前はシンボルアトムで，情報は任意のＳ式である．たとえば，*1, *2, *3 の３つの変数が，それぞれ，犬のポチ，犬のシロ，猫のミケであるという３つの情報を内容として持ち，名前が name_memo である黒板は，

黒板：name_memo

((*1^dog poti) (*2^dog siro) (*3^cat mike))

のような概形を持つと考えればよい．黒板に書かれる各情報は任意のＳ式でよく，先頭に述語が来る必要はない．情報をどう解釈するかはプログラマーに任されている．

黒板は，述語 bcreateで作りだし，述語 bputなどで情報を追加し，述語 bgetなどで情報を取り出す．以下に関連する組込述語をあげる．> は入力変数，< は出力変数に限ることを示す．* は入出力どちらでもよい．

(1) (bcreate >name)

>nameという名前の黒板を作る．この黒板はここまでバックトラックした時に消滅する．上記の例では，(bcreate name_memo) となる．

(2) (bput >name >x)

>nameという名前の黒板の最後に >x で示される情報を追加する．たとえば，

(bput name_memo (*1^dog poti))

とすれば，name_memo という黒板に (*1^dog poti) が追加される．

(3) (bget >name <x)

>nameという名前の黒板から <x と単一化できる情報を取りだす．<x はその情報と単一化される．backtrack によりすべての可能性が得られる．たとえば上の黒板で，

(bget name_memo (*x poti))

とすれば，

(bget name_memo (*1^dog poti))

と一回だけ成功するが，

(bget name_memo (*x^dog *name))

ならば，２回成功し，ポチとシロが得られる．

(4) (bget >name >pred <x)

>nameという名前の黒板から，<x と >pred の関係にある情報を取りだす．>pred は２項述語であり，<x はその情報と >pred によって処理されたものとなる．たとえば，

(as (top_eqeq (*x . *r) (*y . *r)) (== *x *y))

で定義された top_eqeq という述語は，最初の要素がまったく同じ(==)という制限のもとで２つの対象をユニファイする．したがって，

(bget name_memo top_eqeq (*2^dog *name))

からシロだけが得られる．>pred が ＝ になると３引数の bget と一致する．backtrack により複数の答えを返し得ることも３引数の bget と同様である．

top_eqeq という述語を使うことにより，定数アトムで表された対象（potiなど）だけでなく，通常の変数やクラス束縛変数においても，対象の同一性／非同一性を正確に扱うことができる．これに対して，通常のユニフィケーションを実現する ＝ 述語は，ある犬 *1^dog と別の犬 *2^dog をユニファイさせてしまい，別の犬と認識されている２つの対象に関する情報を両方とも取って来ることになる．３引数の bget （それは ＝ 述語が使われる）ではなく，４引数の bget を（top_eqeq 述語と組合せて）使うことの意味がここにある．

２．４　黒板情報の動的な変化

黒板の情報は，ユニフィケーションによって動的に変化する．たとえば，ある時点において，animal という名前の黒板の内容が，

((*1^犬　名前　ポチ)
(*2^犬　買主　岡田)
(*3^猫　名前　ミケ))

であったとする．その後新たな情報が得られ，それをもとにして *1 と *2 が同一視され，それらがユニファイされたとすると，その瞬間に黒板の内容も，

((*1^犬　名前　ポチ)
(*1^犬　買主　岡田)
(*3^猫　名前　ミケ))

に変化する．これは，名前がポチ，買主が岡田である犬と，名前がミケである猫という２つの対象だけが存在することを記述している．

逆もまた真である．*1 と *2 がユニファイされた時点まで backtrack し，それらの同一視が解除されると，黒板の内容は瞬時にもとに戻る．黒板は再び３つの動物についての情報を表現することになる．

このような種類の情報処理を言語処理系のレベルで，明快にかつ高速にサポートするところに，論理変数を扱う Prolog あるいは PAL が lisp に優越しうる大きな利点がある．しかしながら既存の自然言語処理の研究では，Prolog を基礎としたものですら，この枠組みの重要性や有効性をはっきりとは認識していない．そして PAL でわれわれが *1^dog と表現する対象を，dog#001 というように表してしまっている．たとえば，もう１つの犬を dog#002 としたとしよう．また，それら２匹の犬が同一視されたとき，その事実は

(as (eq_hypo dog#001 dog#002))

の実行により，節として表現されたとする．すると以後同一視された dog#001 (= dog#002) の情報を取りだすには，この節を用いて，dog#001 と dog#002 で書かれた情報を別々に探索しなければならない．このような同一視は，２つだけでなくいくらでもおこりうる．それをユーザーのプログラムレベルで処理するのでは，高度な意味処理は極めて困難になると考えられる．

このようなことは，lisp でも smalltalk でも起り得る．一方，Prolog の論理変数は，それを言語処理系のレベルでうまく扱う可能性がある．それには論理変数を使うことが必要であり，dog#001 のような扱いは Prolog の枠組みの可能性を放棄するものである．

ただし，残念なことに実際には，たとえ論理変数のその意味での重要性に気付いたとしても，現在の標準的な Prolog ではそれが充分うまく実現できない．集合束縛変数と黒板などが完備している Prolog-PAL ならば，情報の動的な変化における論理変数の有効性を自然に利用することができる．

３． 単語の解釈の整合性保持のための黒板の利用

３．１ 単語の解釈の整合性

自然言語の意味処理をうまく実現するためには，各単語の解釈の仮説を整合的に扱うための情報処理が重要となる．各時点で得られた仮説を共通の黒板に書いておき，仮説が必要な場合に，すでにある仮説が使えるか，また新しい仮説が既存の仮説に矛盾しないかを検証すれば，仮説の整合性を保つことができる．

例をあげよう．「医者」という単語を含んだ文として，

「太郎は職業のなかで医者が最高と思っている」
「若い医者を選択した」
「若い医者が妊娠した」

を考える．それらの文における「医者」の解釈は，それぞれ，

「医者という職業」 (*2^doctor)
「職業が医者である人間」
(*1^human (==>job *2^doctor))
「職業が医者である女性」
(*1^woman (==>job *2^doctor))

のようになると考えられる．

はじめの２つの解釈のちがいは次のようにして生れる．TALK では，「医者」という単語の辞書どうりの意味は，「医者という職業」にあたる doctor だけになっている．したがって「医者」だけを入力すれば，TALK はその意味として (*2^doctor) だけを出力する．ところが「若い医者」を入力すれば，TALK は図２のような意味表現を返す．このなかで「医者」は (*1^human (==>job *2^doctor)) つまり「職業が医者である人間」と解釈されている．図２の中には「医者」のこの解釈に対応する意味表現が２箇所に含まれている．この意味表現は，「若いという形容詞の actor としては human しかとれない」という知識と，「職業は人間を表しうる」という知識が相互作用した結果できるものである．

```
(*1^human
  (==>job *2^doctor
      (z *3^sentence
          (verb young)
          (mod 24 i)
          (actor *1^human (==>job *2^doctor)))))
```

図２ TALK が与える「若い医者」の意味表現

「若い医者」の解釈を作る述語を仮に main と呼ぶ．main における情報処理は次のように進む．「若い」という形容詞から young が得られ，「医者」という名詞から (*2^doctor) という意味表現が得られる．main は young と (*2^doctor) を述語 merge に送り，それらを組合せて文としての意味表現を作ることを命じる．次に，young の格情報に基づいて，(*2^doctor) が youngの actorになれるか否か(条件は humanになれること）が point 述語で検討される．point 述語は，（職業としての）医者が人間になりうることを確かめ，そのとき得られる新しい意味として，(*1^human (==>job *2^doctor)) を返す．したがって，me

rge のつくる意味表現は，図３のようになる．問題は次にある．main は図２の意味表現を作りたいが，merge からの情報だけではうまくいかない．なぜなら「医者」の解釈が merge の中で変更されてしまったのに，その情報を受取っていないからである．このままでは，図４のような意味表現を作ってしまうことになろう．

```
(*3^sentence
    (verb young)
    (mod 24 i)
    (actor *1^human (==>job *2^doctor)))
```

図３　merge が作る「医者が若い」にあたる意味表現

```
(*2^doctor
   (z *3^sentence
      (verb young)
      (mod 24 i)
      (actor *1^human (==>job *2^doctor))))
```

図４　main がmerge の情報だけから作る「若い医者」の意味表現

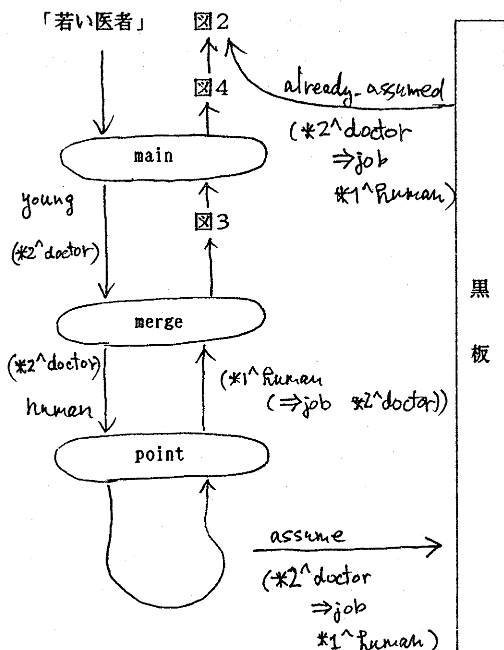

図５　「若い医者」の解釈過程の全体像

問題はたとえば，「若い医者を選択する」という文を解釈するときに起る．この場合，「医者」は２つの述語と関係している．１つは「若い」という形容詞で，このとき actor である「医者」は人間でなければならない．もう１つは「選択する」という動詞で，このときobject である「医者」は人間でも職業でもよい．しかしながら１つの文として解釈するときには，「医者」は人間でなければならない．「医者」という単語が「若い」との関係では人間として解釈され，「選択する」との関係では職業として解釈されるように，１つの文で同じ単語の解釈が異なることは，かけ言葉の例はあるが，普通は起らないと思われる．したがってその制限をうまく反映するためには，文を読み進める時点である単語がどう解釈されているかを示す仮説が記憶され，それとの整合性を考慮して後の仮説処理を進める必要がある．

３．２　黒板を利用した解決方法

黒板を使ってこれを解決しよう．それには図６のように定義された述語を経由して黒板を利用すればよい．ここでは assume という名前の黒板を使う．assume 述語は，*sex と top_eqeq の関係にある仮説が存在しなければ *sex で示される仮説を記録する．already_assumed 述語は，*sex と top_eqeq という関係を持つ仮説が存在すればその情報を取りだして成功，存在しなければ失敗する．

```
(as (assume *sex)
    (unless (bget assume top_eqeq *sex)
            (bput assume *sex)))
(as (already_assumed *sex)
    (bget assume top_eqeq *sex))
```

図６　単語の解釈を整合させるために使われる述語の定義

単語の解釈の整合性を保つためには，単語の意味に関する仮説を assume によって記憶し，already_assumed によってそれを呼出し，すでに存在する仮説を尊重すればよい．たとえば point 述語を直接呼出すかわりに，図７の POINT 述語を経由して point 述語を呼出せば，point 述語で作られる分の仮説は整合性が保たれる．このとき，上記の main から (already_assumed (*2^doctor *r *o)) という呼出しをすれば，「医者」に関して作られた仮説：

(*2^doctor　==>job　*1^human)

が読みだせるので，「若い医者」の意味を図４から図２に変更できる（図５）．

```
(as (POINT (*x . *r) *zc *newobj)
    (already_assumed (*x *rel *y)) (cut)
    (point (*y (*rel . (*x . *r))) *zc *newobj))
(as (POINT (*x . *r) *zc *newobj)
    (point (*x . *r) *zc *newobj)
    (assume (*x *rel *y)))
```

図7　単語の解釈の仮説の整合性を保つPOINT述語

```
(*3^sentence
 (verb draw)
 (mod 96 ke)
 (object *0^object
         (==>shape *1^circle
                   (z *2^sentence
                      (verb white)
                      (mod now)
                      (actor *0^object
                             (==>shape *1^circle))))))
```

図8　「白い丸をかけ」の意味解釈結果

4.　情報の伝播による問題解決

たとえば，「白い丸をかけ」などの命令を受取って，それを解釈，実行する場合を考えてみよう．その命令の TALK による意味解釈結果は図８のようになる．図８の木の形をした意味表現を１列に展開すれば，簡単に図９のプログラムに変換される．ただし color や ==>shape や draw_obj_place は，それぞれ，

(color *x *y)　　.... *x（物体）の色は *y である
(==>shape *x *y) .... *x（物体）の形は *y である
(draw_obj_place *1 *2 *3)
　　.... *1（主体）が*2（対象）を*3（場所）にかく
を意味する述語である．

```
(and (color *0^object white_c)
     (==>shape *0^object *1^circle)
     (draw_obj_place *2 *0^object *3))
```

図9　図8を変換して得られるプログラム

このプログラムを実行して，白い丸をかくには，color や ==>shape や draw_obj_place などは，どのような述語と考えればよいだろうか．たとえば，

(color *0^object white_c)

のなかで，color は *0^object の性質を記述している．もし *0 がすでに具体的にあるもの，Ｘだとわかっているなら，Ｘが白いかどうかを検査し，その結果に応じて成功または失敗すればよい．しかし，Ｘはまだ存在しない．color はこのとき，物体 *0 の色に関する情報を追加する役割をはたすべきである．このとき，黒板の機構が使える．

```
(as (add_info *x *info)
    (bput memo (*x . *info)))

(as (read_info *x *info)
    (bget memo top_eqeq (*x . *info)))
```

図１０　情報を操作するための基本的な述語

まず基本的な述語を図１０のように定義する．add_info は *x の情報として *info を追加する．read_info は *x の *info の形の情報を１つずつ読みだす．これらを用いれば color と ==>shape は情報の追加述語として図１１のように定義できる．color は *color で示される色名情報を *obj の color 情報に付加える．==>shape は*figure の束縛クラスで示される形態情報を取りだし，対象 *obj の figure 情報につけ加える．draw_obj_place は，*obj に関して蓄えられた情報を読みだして実行するとして定義する．これで図９が意図どおりにうまく実行できる．

```
(as (color *obj *color)
    (add_info *obj (color *color)))

(as (==>shape *obj *figure)
    (bc *figure *fig)
    (add_info *obj (figure *fig)))
```

図１１　color と ==>shape の定義

もう少し複雑な例として，「半径が５０の赤い丸をかけ」の場合を考えてみよう．得られるプログラムは図１２のようになる．それを変換して得られるプログラム（図１３）に出現する述語（図１４）を説明しよう．be は等しいことを意味する．２つの引数が両方ともアトムか両方とも変数の場合には，両者はユニファイされる．そうでないときな，情報の移動がおこる．その情報は，情報量の多い方から少ない方向に移動させる．<==hankei は第１引数の情報を読みだし，第２引数の情報を追加する．

```
(*7^sentence
 (verb draw)
 (mod 96 ke)
 (object *4^object
         (z *6^sentence
            (verb be)
            (mod 24 dantei)
            (theme *4^object)
            (actor *2^number
                  (<==hankei *4^object
                             (==>shape *3^circle
                                      (z *5^sentence
                                         (verb white)
                                         (mod 24 i)
                                         (actor *4^object
                                               (==>shape *3^circle))))))
            (comp *1^number (value 50)))
         (==>shape  *3^circle
                   (z *5^sentence
                       (verb white)
                       (mod 24 i)
                       (actor *4^object
                              (==>shape *3^circle))))))
```

図１２　「半径が５０の赤い丸をかけ」から得られる意味表現

```
(and (be *1^number 50)
     (color *4^object white_c)
     (be *2^number *1^number)
     (<==hankei *2^number *4^object)
     (==>shape *4^object *3^circle)
     (draw_obj_place *5 *4^object *6))
```

図１３　「半径が５０の赤い丸をかけ」の意味表現（図１２）から得られる PAL プログラム

意味表現を変換して得られるプログラムは，はじめに与えられた順序で機械的に実行されるわけではない．次に実行する述語呼出しを決定する前に，すべての述語呼出しがその時点で変数の少ない順にソートされ，実行可能性を検査される．上記の例の場合，これによって，情報は図１５に示すように伝播して，ついには *4^object にすべての情報が集まる．最後に draw_obj_place によって *4^object の情報が読みだされ，「半径が５０の赤い丸」が描かれることになる．

```
(as (be *x *y) (atom *x) (atom *y) (cut)
    (= *x *y))
(as (be *x *y) (var *x) (var *y) (cut)
    (= *x *y))
(as (be *x *y) (atom *y) (var *x) (cut)
    (add_info *x (val *y)))
(as (be *x *y) (atom *x) (var *y) (cut)
    (add_info *y (val *x)))


(as (<==hankei *hankei *obj)
    (read_info *hankei (val *val))
    (add_info  *obj (hankei *val)))
```

図１４　be と <==hankei の定義

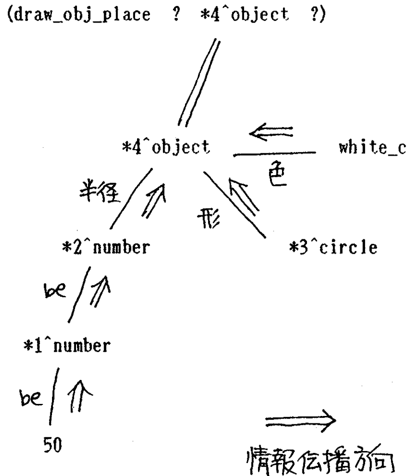

図１５　情報伝播による実行

５．　黒板の実現方法

Prolog を拡張して，どの述語からでも同一の名前でアクセスできる変数を追加するのは容易である．それを大域変数と呼ぼう．普通の変数との違いは，１つの名前を持つということだけである．しかし，それは論理変数なので，たとえば，値がＡだったのをＢに更新することはできない．

しかし図１６のようにストリームの考え方と使えば，論理変数 *bb のなかに情報をつぎつぎに蓄えていくことができる．その過程では，
*bb に含まれる最後の変数が，（情報n . *new_v）という形のコンスにユニファイされることが繰返される．

```
1: *bb = *0
2: *bb = (情報1 . *1)
3: *bb = (情報1  情報2 . *2)
4: *bb = (情報1  情報2  情報3  . *3)
```

図１６　１つの論理変数による情報の蓄積

黒板を実現するための最も簡単な方法は，黒板をそのような形で情報の追加される大域変数だと考えることである．bcreate はその大域変数の名前を定義する．bput は，黒板名と情報が与えられて，黒板名で示される大域変数の値のＳ式に含まれる最後の変数を，（情報 . *new_v）という形のコンスにユニファイする．bget は黒板名とＳ式がなどが与えられて，黒板名で示される大域変数の中の情報とＳ式をユニファイし，成功すればそれを取りだす．

情報を取消す機能も追加したいときには，図１６とは別の表現が必要である．たとえば，図１６の 情報i の部分を，(*i . 情報i) として，*i が変数ならば，情報i は有効であり，*i が dead に束縛されたら無効で存在しないとみなせばよい．

Prolog-PAL ではこれらのことをもっと簡単に実現できる．PAL には，制約付変数を扱うための枠組みができている．それは変数に制約と呼ばれる任意のＳ式を割当てることができる．制約は自由に変更できるから，上記のような工夫が何もなくても，変更のまったく自由な黒板が実現できる．制約は backtrack によって変更前のもの戻るので，黒板も同じように backtrack すればもとに戻ることができる．したがって，大域的な名前を持つ１つの制約付変数が１つの黒板を実現することになる．

６．　むすび

以上により，黒板は，Prolog による知識処理にとってなくてはならない機構であることは明らかになった．黒板の有効性はもちろんこれに留らず，たとえば，文脈処理などを実現する基礎的な枠組みとしても使える．文脈処理への応用の方法については別の機会に述べたい．

文　献

[1] 赤間清：ＰＡＬ：継承階層を扱う拡張PROLOG，情報処理学会論文誌　Vol.28 No.4 pp.27-34 (1987)

[2] 赤間清：継承階層 prolog の高速化機構，人工知能学会誌，Vol.2, No.4, pp.492-500 (1987)

[3] 赤間清：継承階層 prolog による自然言語処理，情報処理学会，自然言語処理研究会資料，62-7, pp.45-52 (1987)

[4] 赤間清：集合束縛変数に基づく意味表現とユニフィケーション，情報処理学会，自然言語処理研究会資料，62-8, pp.53-60 (1987)

[5] Sowa, J. F. : Conceptual Structures, Addison-Wesley P.C., p.481 (1984)

[6] Ait-kaci,H. and Nasr,R : LOGIN: A Logic Programming Language with Built-in Inheritance, the journal of logic programming, Vol.3, No.3, pp.185-215 (1986)

[7] Dincbas,M. : Domains in Logic Programming, Proceedings of AAAI '86, pp.759-765 (1986)