プログラミング言語　7－1
（１９８６．７．１８）

ＰＲＯＬＯＧの視覚的計算モデルＢＰＭ

森下　真一　・　沼尾　雅之

日本アイ・ビー・エム株式会社　サイエンス・インスティチュート

Ｐｒｏｌｏｇのプログラムは、宣言的にも手続き的にも解釈できるが、プログラムの作成・デバッグ・分析というプログラミングの過程において困難の原因となるのは、複雑なバックトラック、カット・オペレーター、副作用等の手続き的側面である。本研究では、Ｐｒｏｌｏｇの手続き的側面を明確に把握するために、Ｐｒｏｌｏｇの実行過程を視覚的に表現する計算モデルＢＰＭ（Ｂｏｘ　ａｎｄ　Ｐｌａｎｅ　Ｍｏｄｅｌ）を提案する。ＢＰＭは、Ｐｒｏｌｏｇのプログラミング環境の１つとして開発されたデバッガーＰＲＯＥＤＩＴ２の基礎として使われ、プログラミング作業を効率化する上での有用性が確かめられている。

BPM: Visual Computation Model for Prolog

Shin-ichi Morishita ・ Masayuki Numao

Science Institute, IBM Japan, Ltd.
5-19 Sanban-cho, Chiyoda-ku,
Tokyo 102, JAPAN

Prolog program has declarative meaning and procedural meaning, however, procedural aspects of Prolog, such as backtrack, cut operator or side effects, mainly cause confusion in programming and debugging work. In this paper, in order to grasp procedureal features of Prolog, we propose a visual computation model BPM(Box and Plane Model) that clarifies the semantics of backtrack, cut operator and other execution control predicates in visual way. BPM is used as the basis of debugger PROEDIT2 that has been developed as a programming environment for Prolog. And it proved to increase efficiency of programming work.

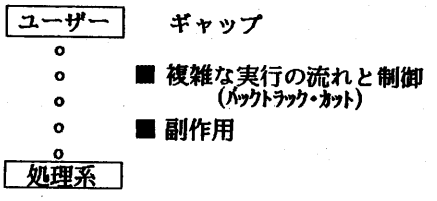
目次

１．　視覚的計算モデルＢＰＭの意図

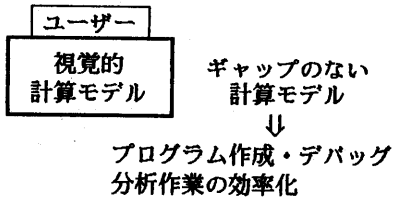
Ｐｒｏｌｏｇでプログラミングをする際の書き方のスタイルには大きく分けて、宣言的な解釈のできるプログラムを書く場合と、手続き的な解釈からプログラムを作成する場合とがある。宣言的な解釈ができるＰｒｏｌｏｇのプログラムは仕様に近い記述ができる点で重要ではあるが、大規模なプログラムを宣言的に記述することは難しい。また、近年のＰｒｏｌｏｇの普及に伴ない最近はＰｒｏｌｏｇで比較的大きなプログラムが作られるようになってきているが、このような実際のプログラミング現場では宣言的な解釈のできるプログラムを書くことよりも、ユーザーが手続き的な解釈を念頭において大規模なプログラムを作成するケースが多い。

ここで宣言的解釈と、手続き的解釈の違いを明らかにしておきたい。宣言的解釈の背景にはプログラムをエルブラン領域上の mapping とみなし、その最小不動点をモデルとする立場がある。この立場は理論的考察をするには欠かせないものだが、そこで仮定されている計算規則は breadth first search でありＰｒｏｌｏｇの計算規則である depth first search とは異なる。その差異が顕著に現れる例として breadth first search した場合にはモデル中に存在が保証されている値でも、depth first search で探す場合には節の順番や、節本体のゴールの順序に意味があるために見つけられないことがある。ここにＰｒｏｌｏｇのプログラムを depth first search を考慮して手続き的に解釈する必要性がある。また手続き的解釈のもとに、Ｐｒｏｌｏｇに組み込まれたカット・ｏｒ・ｉｆ－ｔｈｅｎ－ｅｌｓｅ等の述語の意味を最小不動点モデルで説明することはできない。例えばカットを含んだプログラムは最小不動点モデル上では存在しうる値を、枝がりにより捨てる可能性がある。（［Ｌｌｏｙｄ］）

このようにＰｒｏｌｏｇでプログラミングする際、ユーザーはプログラムを手続き的に解釈することが必要不可欠である。ここでいうユーザーがプログラムを手続き的に解釈するということは、具体的にはユーザーが処理系の動きを理解することにあたる。この場合Ｐｒｏｌｏｇ処理系のプログラム実行のメカニズムがユーザーにとって把握しやすいものであればよいのだが、実際には必ずしも理解しやすいものではない。特にバックトラック・カット等の実行過程や、それらがルールベースの書換えと絡んだ場合の動き等は複雑で、ユーザーの描いているプログラムの実行イメージと、処理系によるプログラムの実行には大きなギャップがある。そこで、そのギャップを吸収し処理系によるプログラムの実行を正確に反映し、ユーザーの意図したプログラムが処理系にいかに解釈されているかを示すことのできる計算モデルを設計し、ユーザーに提供することによりプログラムの作成・デバッグ・分析の作業における従来の負担を軽くすることが必要である。以上の問題意識の下、本研究ではユーザーのプログラミング作業を効率化する為の計算モデルとして、視覚的計算モデルＢＰＭ（Box and Plane Model）を提案する。



★ 従来のプログラミング環境



☆ 視覚的計算モデルＢＰＭ提案の意図

ＢＰＭはＰｒｏｌｏｇの実行過程を表現する為の記号として、ＤＥＣ１０ Ｐｒｏｌｏｇのデバッガーで用いられているボックスと矢印、およびＢＰＭで新しく導入された概念であるｐｌａｎｅを用いる。大事な点はこれらの記号にどのような意味づけをするかであるが、意味づけの対象としてＢＰＭはＰｒｏｌｏｇの様々な側面のうち、特にギャップを生む可能性のある問題点として次の項目に注目し、そのセマンティクスを与える。

（１）　Ｐｒｏｌｏｇプログラムの複雑なバックトラックの動き
（２）　バックトラックを制御するカットオペレーター
（３）　実行制御述語（ｏｒ，ｎｏｔ．．ｅｔｃ）
（４）　ルールベースを変更する述語

以下では、各々の項目についてその問題点とＢＰＭによる解決のアプローチを説明する。

まず（１）の問題点は複雑なバックトラックの動きである。従来のボックスモデルと呼ばれるものは、１つのゴールに注目しその実行過程をボックスに矢印を配置して表現したがバックトラックの動きを表現するにはボックス間の制御の流れを表わさなくてはならない。その際大事なことは、どの程度の範囲のボックスをひとまとまりの単位として、ボックス間の制御の流れを表現するかということである。ＢＰＭではＰｒｏｌｏｇの変数束縛が節単位であるという性質に注目し、節内のゴールをひとまとまりにし、各ゴールに対応するボックスを２次元的に配置し、ボックス間の制御の流れをボックス間に矢印を結び付けることで、バックトラックを自然に表現することに成功している。

次に（2）のカットオペレーターのセマンティクスであるが、従来は探索木の枝がりで説明されていた。しかしカットが多用される通常のプログラムの動きを理解する場合、各々のカットの有効範囲がどこにあるかを把握したり、カットによる失敗が実行された後にプログラムの実行制御がどこに移るか知りたいときなど、従来の方法では必ずしも分りやすい説明がなされているとは言い難い。これに対しＢＰＭでは、カットの有効範囲を明示し、カットによる失敗が実行されたときの制御の移り先を明確に表現することができる。

次に項目（3）を解説する。ｏｒ，ｎｏｔ 等の実行制御述語は、 高階述語を用いて定義することができるが、通常の処理系ではあらかじめシステム組込み述語として用意されている。ところで、このようなシステム組込み述語と高階述語とではセマンティクスが異なったものになるという問題点がある。特にこれらの述語がカットとともに用いられた場合にこの違いはカットの有効範囲の差異となって顕著に現れる。従って計算モデルでも、これらのシステム組込みの実行制御述語の表現ができることが必要になってくる。本稿ではｏｒ述語を例に、ＢＰＭによっていかに実行過程が表現されるかを示す。そしてｃｕｔとともに用いられた場合の問題点を考察し解決案を与える。

項目（4）については、ルールベースの詳細な変化や大域的な変化についても考察できるモデル化を行った。

以上、視覚的計算モデルＢＰＭの提案の意図と、モデルの扱おうとする問題点と解決のアプローチを示した。次章ではＢＰＭの定義とその表現能力について述べる。

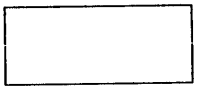
2.　BPM（Box and Plane Model）の定義と表現能力

2.1　シンタックス

- box

goal-box，head-box の2種類があり、各々点線と実線でかこまれた箱で表現される。

head－box



goal－box

- plane

plane は、内部にいくつかの box を含み、太線で囲まれた箱である。

plane

- arrow

arrow は box 間の制御の流れ及び goal-boxと plane 間の制御の流れを示す記号で、以下の種類がある。

→ : call/exit,　　← : redo/fail,

⇒ : in_call/out_exit,　　⇐ : in_redo/out_fail

２．２　セマンティクス

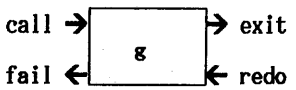
Ｐｒｏｌｏｇのプログラムの表記について、次の記号の約束をする。

a,b,c,...g, g1, g2,...	述語名	g1(T1) & g2(T2) & ・・・	節の本体
X, Y, ...	変数	<- g(T)	ゴール g(T) の実行
T, S, ...	項	θ, θ_1, θ_2,..	代入，ユニファイヤー
g(T), g1(T1), g2(T2), ...	ゴール	$\theta_1 \circ \theta_2$	代入の合成
g(T) <- g1(T1) & g2(T2) &・	節		

２．２．１　ゴールの実行の表現

対象とするゴールを g(T) とする。g(T) の実行過程は、 g(T) に対応して生成される goal-box へ、call, exit, redo, fail の arrow を配置して表現する。各々の arrow の意味は、

call　g(T) に対応する goal-box を生成と実行の要求
exit　g(T) の実行結果 g(T)θ を返す。
　　ユニファイヤーθを返すと考えてもよい
redo　g(T) を再実行することを要求する
fail　g(T) の実行が失敗したことを示す

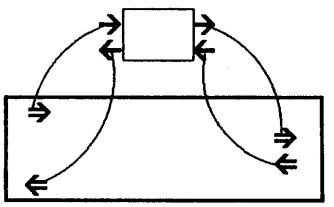


これらの arrow を、配置された goal-box に『属する』と呼ぶことにする。

２．２．２　goal-box 内部の表現

goal-box（対象のゴールを g(T) とする）の内部での実行過程を、boxとarrowを２次元的に配置して表現するものが、planeである。plane は head-box とそれに続く goal-box から構成される。そして box 間の制御の流れを表現するため arrow が box のまわりに配置される。

goal-box に属する call(➔), exit(➔), redo(➔), fail(➔)の各 arrow は、plane では各々、in-call(⇒), out-exit(⇒), in-redo(⇐), out-fail(⇐)によって対応づける。in-call, in-redo は、plane への入口を、out-exit, out-failは、plane からの出口を示す。



head-box ではゴール g(T) とユイファイ可能な節をルール・ベースから取り出す過程が表現される。head-box にも goal-box と同様に call, exit, redo, fail の４種類の arrow があるが、これらの意味は以下のようになる。

call　g(T) に対応する head-box を生成し、
　　g(T) にユニファイ可能な節を要求
exit　g(T) にユニファイ可能な節
　　g(T0) <- g1(T1) & ・・
　　および、T と T0 のユニファイヤーθを返す
redo　g(T) にユニファイ可能な節を新らたに要求
fail　g(T) にユニファイ可能な節が尽きたことを示す

これらの arrow を、配置された head-box に『属する』と呼ぶことにする。

次に goal-box に対応する plane の表現を次のプログラムを例に説明する。

```
descendant(X,Y) <- offspring(X,Y).
descendant(X,Z) <- offspring(X,Y) & descendant(Y,Z).
offspring(abraham,ishmael).
offspring(abraham,issac).
offspring(issac,esau).
offspring(issac,jacob).
```

図１は、ゴール　<-descendant(abraham,V) & fail.　の実行過程をＢＰＭで表現したものである。goal-box descendant は a→ で生成され初めの実行結果 →b (descendant (abraham,ishmael))を返す。続いて ←c,←e,←g の再実行の要求があり実行結果として、→d,→f,→h が返されたのがわかる。 最後の再実行の要求 ←i に対しては解の探索に失敗して ←j で終わる。

図２は、図１の goal-box descendant に対応する plane である。 goal-box の arrow a→ ～ j← は、各々 plane の arrow a⇒ ～ j⇐ へ置きかえられる。また節を取り出す過程は、head-box descendant に属するarrowで表現されている。→1 で descendant の初めの節が返され、→4 で２番目の節が返されたのがわかる。

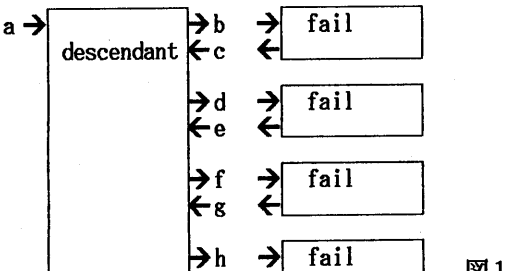


図１

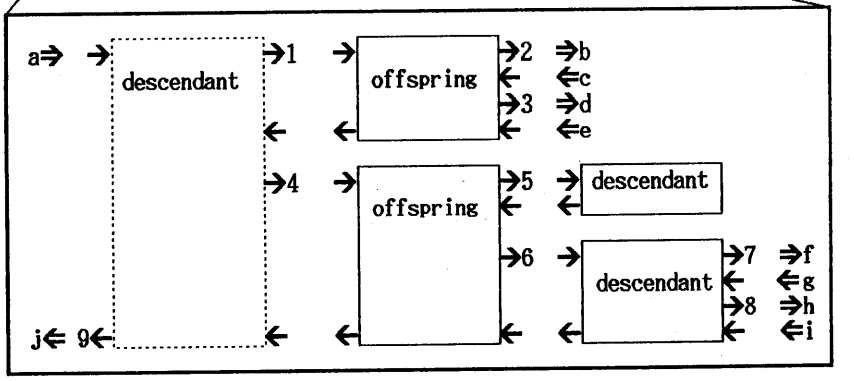


図２

ところで plane 中では exit arrowに返されるユニファイヤーを１つ１つ追うことで、ユニフィケーションの過程を調べる事ができる。➔1 でdescendant の初めの節

descendant(X,Y) <- offspring(X,Y).

とユニファイヤー θ_1 = { abraham/X, V/Y } が 返されると、次にユニファイ後の節の本体 offspring(abraham,V) の goal-box は➔1 の右側に配置され最初の実行結果 offspring(abraham,ishmael) とユニファイヤー θ_2 = { ishmael/V } が ➔2 に返される。そして ⇒b へと抜ける。⇒b には descendant(abraham,V)θ_b が返される。このユニファイヤーθ_b = { ishmael/V } は、$\theta_1 \circ \theta_2$ = { abraham/X, ishmael/Y, ishmael/V } を変数 V へ制限したものである。

次に ⇐c の再実行の要求に対しては、goal-box offspringから次の実行結果offspring(abraham,issac) とユニファイヤーθ_3 = { issac/V } が ➔3 に返され ⇒d へ抜ける。⇒d には descendant(abraham,V)θ_d が返される。ユニファイヤーθ_d = { issac/V } は、$\theta_1 \circ \theta_3$ = { abraham/X, issac/Y, issac/V } を変数 V へ制限したものである。 同様に ⇒f, ⇒h のユニファイヤー θ_f, θ_h は、➔4, ➔6, ➔7, ➔8 のユニファイヤー θ_4, θ_6, θ_7, θ_8 により合成された$\theta_4 \circ \theta_6 \circ \theta_7$と$\theta_4 \circ \theta_6 \circ \theta_8$ の、変数 V への制限である。

２．２．３ ＢＰＭによるカット・オペレーターのセマンティクス

plane から out-fail arrow (⇐) を出口として抜け出る場合には２通りある。１つは head-box がユニファイ可能な節を取り出すのに失敗して fail で抜け、out-fail arrow で plane から抜け出る場合である。例えば図２では、head-box descendant(abraham,V) はユニファイ可能な節を取り出すのに失敗し(9⇐)、j⇐ から plane を抜け出ている。他の１つは、カット・オペレーターの goal-box がfailして plane から抜ける場合である。

すなわち、goal-box cut のセマンティクスを、call(➔)に対しては exit(➔)を返すが、redo(⇐)に対しては fail(⇐) を返し、続いてout-fail(⇐) で plane から抜け出ると規定する(図３を参照)。また以後カットの有効範囲 (スコープ) は plane であるという言い方をする。例としてカットを使ってｎｏｔを定義した場合の実行過程を図４に挙げておく。

```
not (P)  <-  P  &  cut  &  fail.
not (P).
```

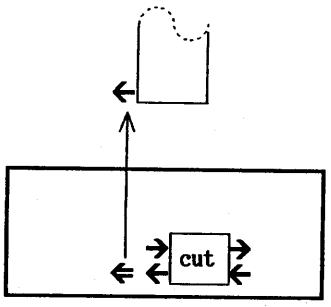
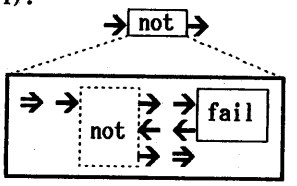


図3 カットのセマンティクス

● <- not(fail).



● <- not(true).

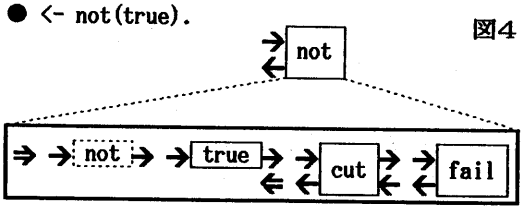


図4

２．２．４　ｏｒ述語のセマンティクス

Ｐｒｏｌｏｇではｏｒを表現するのに、同一述語名同一 arityを headにもつホーン節をつくることで、論理的には解決している。しかし ｏｒ を表現するのに、いちいち新しい述語名そしてホーン節を作るのは手間がかかる。そこで実際のプログラミング現場では使いよさが優先され、通常の処理系ではｏｒを表現する述語がホーン節本体に書くことができるようになっている。

例えばｏｒ述語を | としたとき、図５（ａ）は | を用いず述語 ｑ でｏｒを表現しているのに対し、図５（ｂ）は述語 ｑ のかわりに | を使用した場合のプログラムである。

```
p <- ･･･& q & ･･･
q <- r.
q <- s.
```

図５（ａ）ｏｒ述語使用前

```
p <- ･･･&(r|s)&･･･
```

図５（ｂ）ｏｒ述語使用後

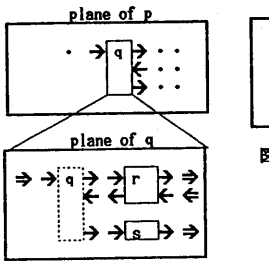


図６（ａ）ｏｒ述語使用前

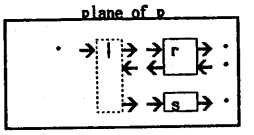


図６（ｂ）ｏｒ述語使用後

ＢＰＭでは | を含むプログラムの実行過程を表現する際、 | を用いないプログラムの実行過程の図から、自然に表現し直せるように配慮している。 例えば図６（ａ），（ｂ）は各々プログラム図５（ａ），（ｂ）の実行過程の一部を表現したものである。図６（ｂ）は図６（ａ）の plane ｑ を plane ｐ のなかの goal-box ｑ へ射影することにより得ると考える。この際プログラム図５（ａ）の述語 ｑ を図５（ｂ）では | へ書換たことに対応して、図６（ａ）の head-box ｑ は図６（ｂ）では、pseudo head-box | に置き換えられる。

この pseudo head-box | は新たな概念だがそのセマンティクスは head-box のセマンティクスに類似する。厳密には次のようになる。まず論理式 Ａ | Ｂ において ｏｒ述語 | の前の論理式Ａを第一論理式、Ｂを第二論理式とここでは呼ぶとする。pseudo head-box | に属するarrowの意味は次のようになる。

call	pseudo head-box を生成し、第一論理式を要求する
exit	論理式を返す
redo	次の論理式を要求する
fail	第一論理式、第二論理式が既に返され、pseudo head-box の終了を意味する

```
p <- ･･･& q & fail.
q <- cut.
q <- s.
```

図７（ａ）ｏｒ述語使用前

```
p <- ･･･ &(cut|s)& fail.
```

図７（ｂ）ｏｒ述語使用後

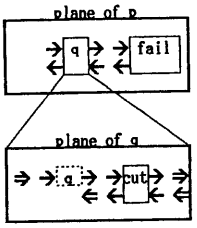


図８（ａ）ｏｒ述語使用前

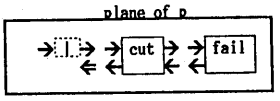


図８（ｂ）ｏｒ述語使用後

このようにＢＰＭではｏｒ述語のセマンティクスを plane の射影で説明するが、カット・オペレーターが第一論理式ないし第二論理式に現れるときは、plane の射影操作を基本にカット・オペレーターのスコープを考慮してセマンティクスを与える。

例として図７（ａ），（ｂ）のプログラムをあげる（このプログラムは実用上の意味はない）。この２つのプログラムはカットのスコープが異なるため同一の動きをしない。このことは、各々の実行過程を表わした図８（ａ），（ｂ）を見ることでより明らかになる。すなわち（ａ）のカットのスコープが plane ｑ にあるのに対し（ｂ）のカットのスコープはplane ｐ にある点が違う。

ここではＣ－ｐｒｏｌｏｇ，VMPROLOG(VM/ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ･ｲﾝ･ﾛｼﾞｯｸ)においてシステムに組み込まれているｏｒ述語のセマンティクスを与えた。しかしながら、ｏｒ述語を

or(P,Q) <- P,　or(P,Q) <- Q

のように高階述語で定義している処理系ではカットのスコープが変わり、スコープは一つレベルが下の plane へ移り、動きは図８(ｂ)と類似のものになる。

２．２．５　ルール・ベースの状態とＢＰＭ

ＢＰＭでは、ルール・ベースの状態を図形中に陽には表現せず、各 arrow にその時点でのルール・ベースの状態が対応していると考える。ここではルール・ベースにルール R を登録する述語を assert(R)、R を削除する述語を　retract(R) とする。図９はゴール assert(R) & retract(R) を実行したときのＢＰＭの表現である。

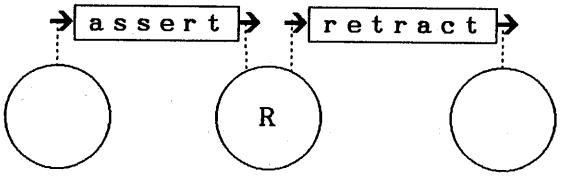


図９

ルール R は、1→ ではルール・ベースへ登録されておらず、goal-box assert(R) で登録され、→2 ではルール・ベースに存在しており、3→ でもルール・ベースの状態は変化せず、goal-box retract(R) で削除されるため、→4 では存在しない。

２．２．６　計算の全体像

ＢＰＭでは計算の全体像を、plane が多層に重なり合った木で表現する(図１０)。

木のトップレベルの plane は、図１のように head-box をもたない plane である。plane 中の各 goal-box に対応する plane はその下に描かれている。木の終端は、head-boxしかない planeである。head-box しかない plane は、対応するgoal-box の述語 g(T) とユニファイ可能な節がない場合及びユニファイ可能な節がファクトしかない場合につくられる。

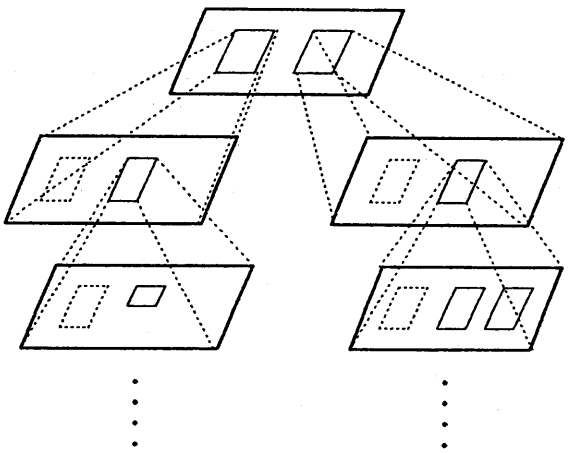


図１０　計算の全体像

２．２．７　ループするプログラムの表現

ループするプログラムのタイプには大きく分けて次の２種類がある。

- recursion loop
- backtrack loop

以下各々についてＢＰＭでの表現を例によって示す。

recursion loop の例として、プログラム

ｌｏｏｐ　<－　ｌｏｏｐ．

を考え、ゴール　<－ｌｏｏｐ．　を実行した場合の実行過程の表現は図１１のようになる。図からわかるように、goal-box loop からは exit arrow で抜けることなく無限に plane を生成してゆく様子がわかる。

また、backtrack loop の例として、プログラム

ｎ（０）．
ｎ（ｓ（Ｘ））　<－　ｎ（Ｘ）．
ｄ（ｓ（Ｘ））　<－　ｄ（Ｘ）．

を考え、ゴール　<－　ｎ（Ｘ）＆ｄ（Ｘ）．を実行した場合の実行過程の表現は図１２のようになる。この場合は無限に広がる plane が生成されてゆくことがわかる。

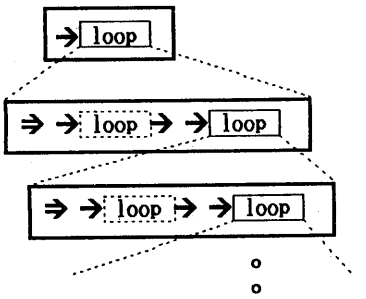


図１１　recursion loop

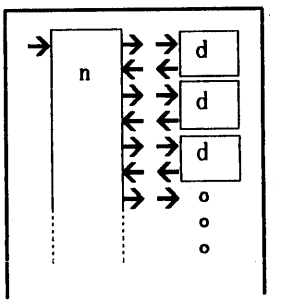


図１２　backtrack loop

３．　まとめ

本稿ではユーザーと処理系のギャップを埋める為の視覚的計算モデルＢＰＭを与えた。ＢＰＭは、プログラム作成・デバッグ・分析という一連のプログラミング作業の効率化を目指し導入されたモデルであり、我々は過去ＢＰＭに基づいたＰｒｏｌｏｇのデバッガーＰＲＯＥＤＩＴ２を開発し、実際にプログラム開発環境として使用している［森下・沼尾］。

参考文献

[Lloyd]　J.W.Lloyd, "Foundation of Logic Programming", Springer, 1984

[森下・沼尾]　森下 真一・沼尾 雅之,
『Ｐｒｏｌｏｇの視覚的計算モデルＢＰＭとそれに基づくデバッガーＰＲＯＥＤＩＴ２』
Proc. of Logic Programming Conf. in Tokyo, 1986