階層的プロダクションシステム-HIPS†

溝口理一郎†† 大上勝也†††* 冨田雅己††††
西山静男††† 角所収††

人工知能の分野において，プロダクションシステムは有望な知識表現法の1つと考えられており，現在いくつかの知識ベースシステムがプロダクションシステムを用いて開発され実働している．プロダクションシステムは基本的にはプロダクションルールによる単なる文字列の書き換えシステムであり，効率的な問題解決システムを構築するためには，問題領域の適切な分解と問題固有の知識及びヒューリスティックな知識の表現方法が大きな問題となっている．本論文では，PS の基本的特徴を保持しつつ，PS の能力を向上させるために設計された階層的プロダクションシステム-HIPS について述べる．HIPS は2方向の階層構造をもつ複数の基本 PS から構成されている．HIPS の主な特徴は人間が持っているような問題固有の階層的知識，及びそれらの知識の使い方に関するヒューリスティックな知識をすべてプロダクションルールの形で表現できるところにある．また HIPS は，与えられるルールに依存して，通常のプロダクションシステム，リスプシステム，パターンマッチングによる関数呼出しに基づくリスプシステム，あるいはそれらの組合せから成る複合的プログラミングシステムとなる．そのため，人間の持つ知識を人間の思考方法にかなり近い形で表現でき，プログラミングが容易になること等を，ブロック問題など2,3の例を用いて説明する．

1. まえがき

人工知能の分野において，知識の表現とその利用法がきわめて重要な問題として注目を集めている中で，Production System[1),10),13),16)]（以後 PS と略す）は DENDRAL[3)] や MYCIN[13)] など多くの実働しているシステムにも用いられ，有望な知識表現法と考えられている．

PS は，基本的には文字列に対する書き換えシステムとして考えられ，次の3つの基本的な構成要素から成っている．

（1） Working Memory (WM): 単なる string の集まり

（2） Production Memory (PM): $\alpha \rightarrow \beta$ という if-then clause の形式をとるルールの集合で，α, β ともに string である．left-hand-side (LHS) すなわち α を条件部，right-hand-side (RHS) すなわち β を実行部と呼び，WM 内に α が現われたら β に書き換えることを意味する．

（3） Production System Interpreter (PSI): PM のルールの中から WM に対して条件部がマッチするものを探し出し，WM を実行部の指定通りに書き換える働きをする．

PS の基本動作はある WM に対して，その条件部がマッチするルールを探し，それを実行することの繰り返し（recognize-act cycle という）であるが，一般に起動可能なルールは複数個存在するので，実際にはこれらのうちどのルールを起動するかを決定する必要がある．このようにある時点で同時に起動可能なルールの集合を conflict set，またこの中からどれか1つのルールをある種の戦略に従って選択することをルール競合の解消 (conflict resolution) と呼び，これらはすべて PSI が行う．

基本的な PS には次に挙げる3つの特徴がある[1),15)]．

（1） Modularity: PM 中の各々のルール間の相互作用は WM の変更（挿入，削除など）を通じてのみ関接的になされるので，ルールの記述に際して他のルールとの関係をあまり考慮する必要がない．そのためルールの追加，削除，変更が容易にでき，システムの能力を段階的に増大させていくことが可能である．

（2） Readability: 従来のプログラム手法では使用される知識の内容が制御の流れの中に含まれているため，どのような知識がいかに使われたかを知るためには，そのプログラムの処理の流れを順次見ていかなければならない．ところが，PM 中の各々のルールはそれぞれが一片の完結した知識であるため，その意味の理解が容易である．

† Hierarchical Production System-HIPS by RIICHIRO MIZOGUCHI (I.S.I.R., Osaka University), KATSUYA OGAMI (Faculty of Engineering, Kansai University), MASAMI TOMITA (Faculty of Engineering, Osaka University), SHIZUO NISHIYAMA (Faculty of Engineering, Kasai University) and OSAMU KAKUSHO (I.S.I.R., Osaka University).

†† 大阪大学産業科学研究所
††† 関西大学工学部電気工学科
†††† 大阪大学工学部電子工学科
* 現在，シャープ株式会社

(3) Self-Explanatory: 処理の全過程が WM の内容を順次変化させたルールの系列としてわかるので，いかにして結論が得られたかが，それを見れば容易に理解できる．

以上の3つの利点は知識工学の分野における情報処理のように，全体の制御の構造が不明確な分野，あるいはそれを事前に決定することがきわめて困難な分野には望ましい特徴である．

それにもかかわらず，PS には効率の悪さ，制御構造の一様性に起因するプログラミングの困難さ等の欠点が指摘されてきた[10]．これらの欠点を克服するためにいくつかの研究[6),9),10)]がなされているが，ほとんどのものは，タグ・マーカ等の特殊記号の導入に代表される ad-hoc な改良であり，PS の優れた特性を犠牲にすることによりなされていた．

本論文では，PS の前述の3つの特徴を保存しつつ，さらに PS の能力を向上させるために基本 PS を2方向の階層構造をもたせて配列することにより，新しく階層的プロダクションシステム-HIPS (HIerarchical Production System) を設計し，インプリメントして良好な結果が得られたので報告する．Davis も述べているように[2),3)]，通常の PS ではルールの数が 100 を超えるような現実レベルの問題になると，競合するルールの数が多くなり，ほとんど実行不能な状態（バックトラックの回数が指数関数的に増大する）となる場合がある．HIPS は，2つの階層性を活用することによって，各時点において最も適したルールが発火するように利用者が自由にコントロールすることができるため，バックトラックを最小限に押え，現実レベルの大きなプログラムを効率よく実行することができる．さらに，HIPS では，各ルールに任意のリスプ関数が書けるように拡張されており，これらの拡張の結果，ユーザが書くプログラムに依存して次の6つの異なったシステムと見ることができる．

1) 階層的プロダクションシステム
2) 通常のプロダクションシステム
3) 通常のリスプシステム
4) 非決定的アルゴリズム用リスプシステム
5) パターンマッチングによる関数呼び出しに基づくリスプシステム
6) 以上の5つのすべての可能な組合せから成る複合システム

すなわち，HIPS は枠組としては階層的プロダクションシステムでありながら，リスプシステムをその一部として取り込んだ複合的プログラミングシステムとなっているが，以下では主に階層的プロダクションシステムとしての側面に重点を置いて述べる．

次章では，HIPS 設計の基本思想及びシステムの概略について述べる．3章では基本 PS の記述及びバックトラック機構について，4章ではインプリメンテーションについて述べ，5章ではロボットハンドによるブロック問題のプランニングといくつかのプログラミング例を示す．

2. システムの概観

2.1 概念的説明

PS を用いた知識表現に基づく問題解決システムを設計する際には，問題領域 (problem domain) をほとんど独立した状態に分解し，問題固有の知識をその使われ方とは無関係にシステムに組み込まなければならず，通常システムの能力はその分解の結果に大きく依存する．したがって，問題領域を適切に分解することは効率的な問題解決システムの構築にとって重要な問題となる．

音声理解システム Hearsay-II[4),7)] では問題領域を，句，単語列，単語，音節，セグメント，パラメータ・レベルの6つのレベルに分けられている．また，Kanade[8)] は画像理解システムに複数レベル表現（対象，副画像，領域，区画，画素レベル）を導入している．Soloway と Riseman[14)] の学習システムには9レベルのパターン記述が導入されている．彼らのシステムでは学習は莫大な量の知識を必要とする外部世界のことを取り扱うための知識駆動型の解釈と考えられる．問題領域のこれらの複数レベルの表現は人間の直観にとって自然であり，そしてそれらはシステム設計者が問題をトップダウンに考えるのに役立っている．

GPS (General Problem Solber) の例では，抽象空間の概念が Sacerdoti[11)] によって用いられている．彼の ABSTRIPS システムにおいてプラン作成は次のようになされている．最初それは詳細な状況を無視して大まかなプランをつくる．この場合，最初のプランは最も抽象度の高い空間における一連の動作からなる．それは直接問題領域に適用できないかもしれないが，主要な因果関係の連鎖を損なってはいない．次に一段低いレベルでの知識を使ってプランを詳細化する．この過程は実際に実行可能な動作の系列が得られるまで続けられる．

以上のように，問題領域及び知識の階層的な（複数

レベルの）表現は複雑な問題を取り扱う際には有益であることがわかる．

PS 設計に関する諸問題の中で，ルール競合の解消は主要な課題である．Davis[1)~3)] は，メタ・ルールの導入によってルール競合の解消問題を論じている．メタ・ルールは conflict set 内から次に実行すべきルールを選択するためのヒューリスティックなルールで，下位レベルの知識をいかにして使うかに関する知識と考えられる．さらに，メタ・メタ・ルールも考えられる．このルールの階層性は表現の一様性を保存しながら，ヒューリスティックな知識をシステムに組み込むことを可能にする．Goldstein と Grimson[6)] そして Hayes-Roth と Lesser[7)] は同様の問題を考察している．それぞれ表現は異なっているが各 recognize-act cycle（以下，繰り返しサイクルという）において，付随したヒューリスティックな知識に基づき，より適したルールを発火させることによってシステムの能率を上げることを意図している点は同じである．

知的システムの構築に必要な以上の2種類の階層的知識を考慮して我々は**図1**にブロック・ダイヤグラムで示す階層的プロダクションシステムの一般的概念を得る．

HIPS は次の4つの特徴を持っている．

（1） 縦の次元はプラン作成のための抽象レベルにあたる，問題に固有な知識の階層性に相当する．

（2） 横の次元はルール競合の解消のための戦略としてのメタ・ルールの階層性に相当する．

（3） 各構成要素は通常の PS 構造を持っている．

（4） 構成要素間の相互作用は隣同士に限定される．

HIPS はルール（知識）の階層構造を反映することによって既存の PS を自然な方向に拡張するように設計されており，そこではルール競合の解消のための ad-hoc な手法は何も使われていない．我々のシステムではユーザが定義したヒューリスティックスを他の知識と同様にルールの形式で組み込むことができる．

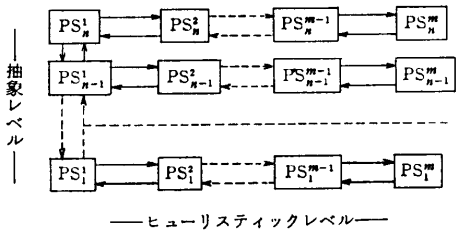


図 1 階層的プロダクションシステム HIPS$^{m}_{n}$
Fig. 1 Two-directional hierarchical structure of HIPS.

さらに問題固有の知識の階層性はシステムが各状態に適したルールの呼び出しをするのに役立つ．なぜならあるレベルでの PS はどのルールを発火すべきかを決定する時には抽象レベルにおいて次の一段高いレベルで設定されたサブゴールを参照することができるからである．このように各レベルでのルールの呼び出しは，次の高いレベルのプランによって導びかれる．

概念的には，HIPS は階層的プランニングという点で ABSTRIPS[11)] と共通点を持つ．しかし，動作の基本としてデータ駆動型とゴール指向型という点，また HIPS にはヒューリスティックな知識を埋め込めるとともに，任意のリスプ関数表現が可能な汎用プログラミングシステムとしても使えるという点等，いくつかの本質的な相異点を持っている．

2.2 システムの動作と概略

ここでは HIPS の縦と横のそれぞれのレベルにおける階層的な処理の概略を述べる．最初ユーザはシステムへ問題記述として初期状態（WM となる）と最終（ゴール）状態，及び各階層に対応したルールすなわち階層的な問題固有の知識とヒューリスティックな知識のリスト，PM を与える．なお，HIPS の縦，横の各レベルはユーザが作成するこの PM 内のルールの階層性の数に従って実行時に自動的に生成される．

図1に基づいて説明する．まず最初 HIPS は最も高い抽象レベルのルールを使って基本 PS である PS^{1}_{n} を働かせ抽象度の高い解（初期状態と最終状態を結ぶ状態の系列）*を得る．しかし，その途中の各繰り返しサイクルにおいて conflict が生じれば，ヒューリスティックレベルで1つ上のレベルの PS^{2}_{n} を起動し，ルール競合の解消を行う**．その際にヒューリスティックス用のルールがまた conflict を起こせば，さらに上位の PS^{3}_{n}, PS^{4}_{n}, …, PS^{m}_{n} と conflict が完全に解消されるまであるいはヒューリスティックレベルのルールがなくなるまで同様の操作が繰り返され，その後もとの PS^{1}_{n} での繰り返しサイクルに戻り，処理を再開する．そうして最終的に解の系列が求まれば，それをもって1つ抽象レベルの低い PS^{1}_{n-1} に降り，そのレベルでのルールを使って PS^{1}_{n} で得られた解系列をサブゴール系列とみなし各サブゴール間を埋めていく．同様に横のヒューリスティックレベルの PS をも必要な際には，それを起動しながら抽象レベルを，

* 通常の意味での解は初期状態を最終状態へ変換するルールの系列として表わされるが，ここでは便宜上状態の系列として表わすことにする．

** ヒューリスティックレベルで上位の PS における動作は主に WM（下位での conflict set）の順序を入れ替える操作から成る．

PS^1_{n-2}, PS^1_{n-3}, …, PS^1_1 へと徐々に降りていき，プランをより詳細なものへと具体化していく．

以上がHIPSの全体の動作の概略であるが，この間にWMとルールのLHSとのマッチングの際のバックトラック，ある繰り返しサイクルにおいて処理を進めることができなくなった際のバックトラック，及びある抽象レベルで得た解の中のサブゴールが適当でないことが下位のレベルに降りて初めてわかった場合*上位レベルにバックトラックしてそのサブゴールを修正し直すレベル間のバックトラックという3種類のバックトラック機構があるが，これに関する詳細な説明は**3.3**に述べる．

3. システムの記述

本章ではHIPSの核となっている基本PSについて述べる．

3.1 ルールの構文と意味

この節では，ルールとWMの形式及びその意味を述べる．HIPSのルールは，

(RULE-ID (LHS)→(RHS))

なる形式をとっており，一般にLHSは，

((B =X) (function-name Arg))
((C #X) (=X *))
(−(=X) (D =Y (=X #Y)))

などのように定数，“=”または“#”で始まる文字アトムである変数，及び関数のリストのリストである．“=”で始まる変数は1個の，“#”で始まる変数は1個以上の任意の文字アトムにマッチし，各々マッチしたアトム及びアトムのリストと対応づけられる．1つのルール内の同じ変数には同じ値がマッチするようになっている．また，“*”はdon't careマークで何にでもマッチするが，値は不定である．そのため同一ルール内に“*”が複数回現われても，そのたびに同じ値であるとは限らない．また，リストの前に“−”があるものは他のリストとは逆にこのリストがWM中に存在しない時のみマッチする．なお，変数と値との結合はLHSをWMとマッチングする際に作ったA-LISTを介して，そのルールのRHSが実行される時に行われる．

以上がLHSの一般形式であるが，以下の例で示す関数 *ODD のようにユーザ定義のLHS関数もあり，これは頭に“*”を付けてユーザが場合に応じて用意する．なお，LHS関数の実引数に表われる =X, #Y 等の変数は各々の結合する値に置き換えられる．

```
(DE *ODD (X)
  (COND ((NULL X) T)
        ((EQUAL (REMAINDER (CAR X)
                           2) 0) NIL)
        (T (*ODD (CDR X)))))
```

一方，RHSは以下に示すような，WM中の要素を除去したり付け加えたり，WMの現在の状況をユーザに知らせたりするシステム関数から成る．

RHS system functions:

(DELETE arg)…WMから argを除去する．
(DEPOSIT arg)…WMに argを付け加える．
(SAY)…WMの現在の内容を知らせる．
(STOP)…処理を終了する．etc.

ここで argは (arg 1, arg 2, …arg n) のように複数個の引数を意味する．ユーザはLHSと同じようにRHS関数を用意できる．RHS関数名は，システム関数，ユーザ定義関数ともに =X, #X 等の変数を評価する場合は“#”で始まり，評価しない場合は単に英文字で始まる名前をつけるように定める．したがって，=XがAと結合されている時，(#DEPOSIT (CLEAR =X)) は (CLEAR A) を，(DEPOSIT (CLEAR=X)) は (CLEAR =X) をそれぞれWMに追加する．また，“*”で始まるシステムRHS関数は，LEXPR関数として定義されており，これは実引数に表われる =X, #Y 等の変数が評価される上に，さらに通常のリスプの意味でも評価される．たとえば，(*DEPOSIT (LIST 'CLEAR =X)) は，引数の評価値 (CLEAR A) をWMに追加する．

以上述べたようにルールに現われる変数はルールに固有の局所変数であり，右辺には任意の関数を書くことができることから，各ルールは1つの独立したプログラムとみることもできる．また，各プログラムは直接名前を呼ばれることはなく，(実質的に名前を呼ばれるのと等価なプログラミングも可能) その起動条件が満足された時に自動的に起動されることから，パターン照合による呼び出しに基づくプログラミングが可能となる．

最後に，WMは文字アトムからなる任意の構造を持つリストのリストである．たとえば，

((A X C) (B X))
((1 5 (2 4)) (6) (9 8 2))

というような形式をとる．

---

* 下位レベルでPSが与えられたサブゴールへの解を見出せなかった場合 (**3.3.2** 参照)．

3.2 Conflict Set

3.1に示した取り決めに従ってPSIはルールのLHSをWMの状況と照合し，起動条件が満足されているルールの集合 conflict set をつくる．

次のようなWM及びPMが与えられた場合を考える．

WM:

((A B C) (A C D) (B X) (E D A) (A) (A E C))

PM:

((P1 ((A =X C) (B X)−(#B E))⇒((DELETE (B X))))
(P2 ((B X) (A =X *))⇒((#DEPOSIT (A =X B))))
(P3 ((=X) (#Y =X)−(C =X))⇒((SAY))))

この時，conflict set は下のごとくになる．

Conflict Set:

((P1 ((DELETE (B X))) ((=X. B)))
(P1 ((DELETE (B X))) ((=X. E)))
(P2 ((#DEPOSIT (A =X B))) ((=X. B)))
(P2 ((#DEPOSIT (A =X B))) ((=X. C)))
(P2 ((#DEPOSIT (A =X B))) ((=X. E)))
(P3 ((SAY)) ((=X. A) (#Y. (E D)))))

たとえば，ルールP1に対しては変数 =X に値B，または値Eを対応づけることによってWMとマッチする．そしてあらかじめユーザにより用意された戦略に従って横のヒューリスティックレベルでこの6つを適当な優先順位に従って並べ換えて，その先頭のルールを選択し，実際にそのルールの RHS に示されたシステム関数を WM に対して解釈実行していく．なお，ヒューリスティックレベルでのルールがユーザによって用意されていない場合には，単にシステムは conflict set 内の先頭のルールから順に選択するようになっている．

3.3 バックトラック機構

HIPS の全処理中には前述のごとく3種類のバックトラックが必要に応じて行われる．以下に，繰り返しサイクルのバックトラックとレベル間のバックトラックについて述べる．

3.3.1 繰り返しサイクルのバックトラック

このバックトラックはある繰り返しサイクルにおいて，その時の WM に対して適用するルールが存在しなかったり，過去に幾度か繰り返してきた繰り返しサイクルにおける WM と同じものになった際，ループを防ぐため1つ前の繰り返しサイクルに戻り，その WM に対してその際の conflict set 内の次のルールを実行することによって行われる．しかし，この繰り返しサイクルのバックトラックが度々行われると非常に効率が悪くなる．したがって，そのためにHIPSには，ヒューリスティックレベルの階層性が用意されており，そこにユーザがヒューリスティックルールを書くことができる．したがって，そのルールを使ってconflict set 内のルールを優先順位の高いものから低いものへと再配列してルールの適切な選択が行えるので実際にはこのバックトラックの回数を少なくすることができ，効率の良い処理が可能となる．

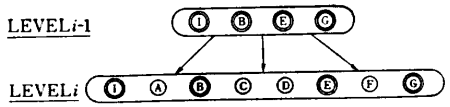


図2 抽象レベルにおけるプランの具体化
Fig. 2 Step-wise refinement of a plan on abstract level.

3.3.2 レベル間バックトラック

このバックトラックはある抽象レベルでの解であるサブゴールの系列の中にあるサブゴールが次の1つ下位のレベルにおいて不適当であるとわかった場合，元の上位レベルにバックトラックして，その不適当なサブゴールを修正して再び下位レベルの処理を続けるというものである．**図2**を使って簡単に説明する．

今，レベル i-1で得た解の系列①→Ⓑ→Ⓔ→Ⓖをレベル i でもう少し具体化して，解の系列が①→Ⓐ→Ⓑ→Ⓒ→Ⓓ→Ⓔ→Ⓕ→Ⓖとなったとする．ただし，この場合 PS は①—Ⓑ間，Ⓑ—Ⓔ間，Ⓔ—Ⓒ間の3回働いている．そして次にレベル i+1 に降りて，さらに具体的な解を得るため PS を①—Ⓐ間，Ⓐ—Ⓑ間，Ⓑ—Ⓒ間と働かせた後にⒸ—Ⓓ間をつなぐルールの系列が1つも存在しなかったとする．この場合にはレベル i にバックトラックし，Ⓒ—Ⓔ間をつなぐサブゴールの系列が他にないか探索し，再びレベル i+1 でサブゴールⒸからの処理を続ける*．またⒸと新しいゴールを結びつけることができなければ再びレベル i にバックトラックし他のサブゴールを探すがⒸ—Ⓔ間をつなぐ経路が他になくなると今度はⒷ—Ⓔ間をつなぐ他の経路を探しレベル i+1 で処理を再開するが，それも失敗すれば最後にはさらにレベル i−1 にバックトラックしレベル i−1 で見つけたサブゴールⒺをも変更

* レベル i にバックトラックする際，レベル i での①〜Ⓒへの解の系列については，レベル i+1 でもより詳細な解が見つかっており，レベル i においてこの部分の処理を再びやり直す必要はない．よってレベル i のこの①〜Ⓒの解の探索の重複を避け，レベル i+1 でバックトラックの原因となったサブゴールの再設定の部分のみを行い，さらにレベル i+1 での処理の重複も避けるように注意が払われている．

してみる．同様にして，このバックトラックはさらに上位のレベルに伝搬し得るが，レベル $i-1$ の上位がもう①と©だけの初期状態の場合には解なしということで処理は終了する．

4. インプリメンテーション

HIPS は Lisp 1.9 を用いてインプリメントされ，作業領域を含めて 80 kB ほどのシステムである．我々の主な目的は人工知能用のプログラミング言語を開発し，その使用経験を通して PS 自体の有効性を確認することにあるので，インタプリタの効率については現在のところ特別な考慮を払っていない．

4.1 基本方針

HIPS のインプリメンテーションの方針は，知識の各抽象レベル上での動作をすべて基本 PS によって統一して表現しようとする点に，その基礎を置いている．したがって個々の PS が全く同一の制御構造を持っており，全体の制御の流れが常に一様であることから，**図3**のように1つの基本 PS をまず作り，再帰的に働かせることによって，ヒューリスティックレベル全体の PS の階層性をインプリメントした．また，縦の抽象レベルについても横のヒューリスティックレベル全体を再帰的に呼び出すことによって全体として図3のような階層性を持った PS に拡張した．このようにレベル間バックトラックの内部の詳細な処理以外ではすべて再帰構造を用いてバックトラック機構をインプリメントしたので，プログラムはかなりコンパクトでかつ見やすいものになった．

4.2 概　略

インプリメンテーションの概略をパスカル風に記述すると**図4**のようになる．

5. HIPS の実行例

5.1 階層的プロダクションルールの例

本節ではロボットハンドによるブロックの問題に関して作成したルール及びその実行結果を用いて HIPS の動作について述べる．

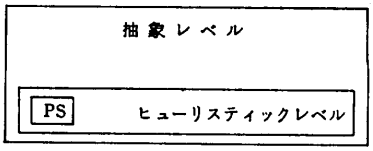


図3 HIPS の再帰的な構造
Fig. 3 Recursive structure of HIPS.

我々の問題の初期状態と最終状態を**図5**に示す．

WM におけるブロック及びロボットハンドの状態の表現方法を以下のごとく定めた．

(ONTABLE X)……ブロック Xがテーブル上にある状態．

(CLEAR X)……ブロック X 上に他のブロックがない状態．

(ON (X Y))……ブロック X がブロック Y 上にある状態．

(HOLDING X)……ロボットハンドがブロック X をつかんでいる状態．

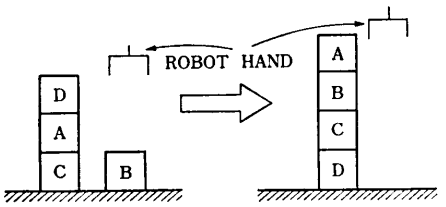
(HANDEMPTY)……ロボットハンドが何もつかんでいない状態．

```
PROCEDURE <HIPS>;
BEGIN
   READ (initial-state, goal-state, PM);
   DATA:=(LIST initial-state  goal-state);
   <abstract level>[DATA PM];
END

FUNCTION <abstract level>[SUBGOALS PM]:BOOLEAN;
BEGIN
   IF PM is empty THEN RETURN with success;
   BEGIN
  LOOP:CASE <one-level loop>[SUBGOALS (CAR PM)] OF
         success:
            set SUBGOALS to the result of <one-level loop>;
            CASE <abstract level>[SUBGOALS (CDR PM)] OF
              success:RETURN with success;
              failure:BEGIN
                        search the inappropriate subgoal;
                        GOTO LOOP ( *search for another possibility*)
                      END
            END;
         failure:RETURN with failure (*level backtrack*)
       END
   END
END

FUNCTION <one-level loop>[SUBGOALS PM]:BOOLEAN;
BEGIN
LOOP:IF there is only one subgoal
      THEN RETURN with success (*the result is the sequence of states
                                 from the initial state to the goal
                                 state*)
      ELSE BEGIN
             WM:=(CAR SUBGOALS);
             goal:=(CADR SUBGOALS);
             CASE <recognize>[WM PM] OF
               success:BEGIN
                         SUBGOALS:=(CDR SUBGOALS);
                         GOTO LOOP
                       END;
               failure:RETURN with failure
             END
           END
END
```

```
FUNCTION <recognize>[WM PM]:BOOLEAN;
BEGIN
   IF WM matches the goal state THEN RETURN with success;
                    (*the result is the sequence of states
                      from the initial state to the goal state*)
   IF WM matches one of the old states
     THEN BEGIN
            erase the current state;
            RETURN with failure (*backtrack*)
          END;
   REPEAT (*make conflict set*)
      IF LHS of a rule matches WM
        THEN add the rule to CONFLICT-SET
   UNTIL PM is exhausted;
   <conflict resolution>[CONFLICT-SET (CDR PM)];
                          (*rearrange the rules of conflict set
                            according to a certain strategy*)
   RETURN <act>[CONFLICT-SET]
END

FUNCTION <act>[CONFLICT-SET]:BOOLEAN;
BEGIN
LOOP:IF CONFLICT-SET is empty
       THEN RETURN with failure (*backtrack*)
       ELSE BEGIN
              execute RHS of the first rule of CONFLICT-SET;
                                        (*WM is rewrited*)
              delete the first rule from CONFLICT-SET;
              CASE <recognize>[WM PM]
                success:RETURN with success;
                failure:BEGIN
                          set WM to the last one;
                          GOTO LOOP
                        END
              END
            END
END

PROCEDURE <conflict resolution>[CONFLICT-SET PM];
BEGIN
   IF conflict set contains more than one rule
     THEN <recognize>[CONFLICT-SET PM] (*using rules for conflict
                                         resolution*)
END
```

図 4 HIPS インタプリタの概略

Fig. 4 A rough sketch of HIPS interpreter.



((CLEAR B) (CLEAR D) ((CLEAR A) (ONTABLE D)
(ONTABLE B) (ONTABLE C) (ON (A B)) (ON (B C))
(ON (D A)) (ON (A C)) (ON (C D))
(HANDEMPTY)) (HANDEMPTY))

initial state goal state

図 5 ロボットハンドによるブロック問題

Fig. 5 Block problem with a robot-hand.

したがって，図5の初期状態及び最終状態の場合には WM の内容はそれぞれ図の下に示されたようなリスト構造で表現されることになる．

図6に示すルールのリストPMを作成した．ここで N1～N2, P1～P2, R1～R4 はそれぞれ抽象レベル 1，抽象レベル 2，抽象レベル3におけるルールである．また，Q1 及び S1 はそれぞれ抽象レベル2及び3でのヒューリスティックレベル用のルールである．抽象レベル1でのヒューリスティックレベル用のルールは用意されていない．レベル3における4つのルールは以下のようなロボットハンドの動作に対応する．

- pick up (X)……テーブル上にあるブロックXを持ち上げる動作（R1).
- putdown (X)……ブロックXをテーブル上に降ろす動作（R2).
- stack (X, Y)……ブロックXをブロックY上に置く動作（R3).
- unstack (X, Y)……ブロックY上にあるブロックXを持ち上げる動作（R4).

レベル2における3つのルールはどのブロックを移動させるかをロボットハンドのことは気にせずに決定するための知識に対応する．レベル1での2つのルールは，そこを通れば早く最終状態に到達できるとユーザが考える状態を SUBGOALという RHS 関数（以前の WM の内容を消し，その引数を新しい WM とする）を用いてサブゴールとして設定する働きをする．また，レベル2と3におけるヒューリスティックレベル用のルールの LHS である ＊DIFFERENCE は conflict set 内のルールをすべて実行した場合に，それぞれの結果とサブゴールとが共有する状態記述の数（利用者が各状態に重みをつけることもできる）を両者の近さとして評価する関数であり，RHS の REPLACE はその評価値の大きいもの順に conflict set 内のルールを再配列する働きをする．レベル1におけるルールは，抽象化されたルールというよりはむしろヒューリスティックスに近い意味を持っていることがわかる．

前述の PM 及び初期状態，最終状態を入力した結果，図7のような10ステップからなる最適な解の系列が1回のバックトラックもなく求まった．ここではブロックの問題は下から積み上げていくべきであるという常識を表わすために，(ONTABLE D) に4点，(ON (C D)) に3点，(ON (B C)) に2点の重みを定

```
(((N1 ((CLEAR A)(CLEAR B)(CLEAR C)(CLEAR D)(ONTABLE A)(ONTABLE B)
       (ONTABLE C)(ONTABLE D)(HANDEMPTY))--->
      ((SUBGOAL ((CLEAR A)(ONTABLE D)(ON (A B))(ON (B C))(ON (C D))
                 (HANDEMPTY)))(SAY)))
  (N2 ((*))--->((SUBGOAL ((CLEAR A)(CLEAR B)(CLEAR C)(CLEAR D)
                          (ONTABLE A)(ONTABLE B)(ONTABLE C)(ONTABLE D)
                          (HANDEMPTY)))(SAY)))))
(((P1 ((ONTABLE =X)(CLEAR =X)(CLEAR =Y))--->
       ((#DELETE (ONTABLE =X)(CLEAR =Y))
        (#DEPOSIT (ON (=X =Y)))(SAY)))
  (P2 ((ON (=X =Y))(CLEAR =X)(CLEAR =Z))--->
       ((#DELETE (ON (=X =Y))(CLEAR =Z))
        (#DEPOSIT (ON (=X =Z))(CLEAR =Y))(SAY)))
  (P3 ((ON (=X =Y))(CLEAR =X))--->
       ((#DELETE (ON (=X =Y)))
        (#DEPOSIT (ONTABLE =X)(CLEAR =Y))(SAY))))
 ((Q1 ((*DIFFERENCE))--->((REPLACE)(STOP)))))
(((R1 ((ONTABLE =X)(CLEAR =X)(HANDEMPTY))--->
       ((#DELETE (ONTABLE =X)(CLEAR =X)(HANDEMPTY))
        (#DEPOSIT (HOLDING =X))(SAY)))
  (R2 ((HOLDING =X))--->
       ((#DELETE (HOLDING =X))
        (#DEPOSIT (ONTABLE =X)(CLEAR =X)(HANDEMPTY))
        (SAY)))
  (R3 ((HOLDING =X)(CLEAR =Y))--->
       ((#DELETE (HOLDING =X)(CLEAR =Y))
        (#DEPOSIT (HANDEMPTY)(ON (=X =Y))(CLEAR =X))
        (SAY)))
  (R4 ((HANDEMPTY)(CLEAR =X)(ON (=X =Y)))--->
       ((#DELETE (HANDEMPTY)(CLEAR =X)(ON (=X =Y)))
        (#DEPOSIT (HOLDING =X)(CLEAR =Y))(SAY))))
 ((S1 ((*DIFFERENCE))--->((REPLACE)(STOP)))))
```

図 6 ブロック問題のルールのリスト (PM)

Fig. 6 List of the rules for the block problem.

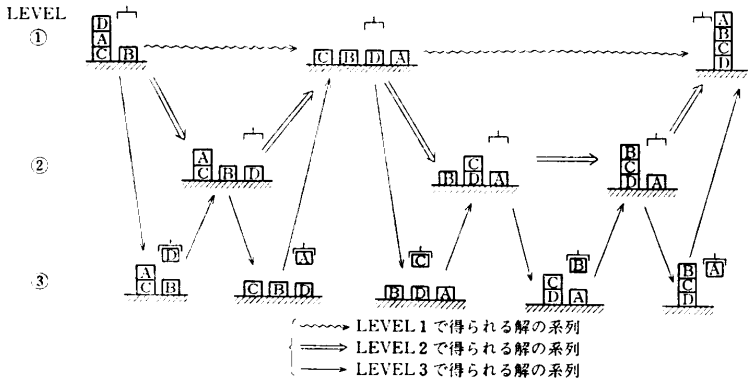


図 7 ブロック問題の解の系列

Fig. 7 Sequences of solution to the block problem.

めた（他は等しく 1 点）．なお，同様の問題を図 6 における R1～R4 のみのルールから成る通常の PS に等価な HIPS で解いた場合，107 回のバックトラックを要し，かつ合計 46 ステップからなる冗長な解系列を得た．このことは，HIPS の抽象レベルによるサブゴールの設定，及びヒューリスティックレベルによる各サブゴールを参照してのルール競合の解消機能の有効性を示している．

ところでレベル 1 でのサブゴールの設定をしなかった場合には，レベル 2 において図 8 のような遠まわりの解の系列が求まった．このように抽象レベルでの階層性を高くしてサブゴールの与え方を適切に行うことは優れた解を得るためには重要であることがわかる．

最後に，ブロック問題の繰り返しサイクルの一例を図 9 に示す*.

5.2 他のプログラミング例

5.2.1 Lisp インタプリタ

いくつかの関数をユーザ定義 RHS 関数として定義しておき WM, GOAL とも任意で以下のようなルールを書けば HIPS はリスプインタプリタそのものとして動作する．

((((P1 ((*)) →((Function-Name Arg) (STOP))))))

すなわち，ルール P1 は任意の WM にマッチし，右

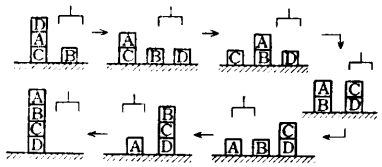


図 8 LEVEL 1 で subgoal の設定をしなかった場合に得られる subgoal の系列

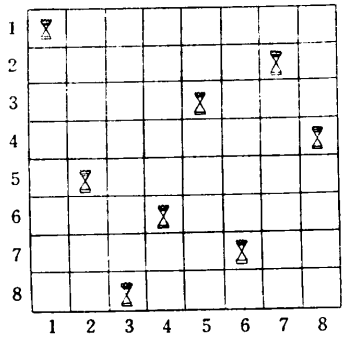
Fig. 8 A solution sequence of the block problem in the case of no subgoals on level 1.

* バックトラックの例を示すために各状態記述の重みとして (ON-TABLE D) に 3 点，(ON (C D)) に 2 点，(ON (B C)) に 2 点を与えて動作させた．

```
>>>THE-CURRENT-STATE-DESCRIPTION<<<
((CLEAR B) (ON (A C)) (ON (C D)) (ONTABLE B) (CLEAR A) (ONTABLE D) (HAND
EMPTY))

++++++++++++++++++++++++++++++++++++++
(8. -TH RECOGNIZE-ACT-CYCLE)

CONFLICT-SET:
((P1) ((#DELETE (ONTABLE =X) (CLEAR =Y)) (#DEPOSIT (ON (=X =Y))) (SAY))
((=X . B) (=Y . A)))
((P2) ((#DELETE (ON (=X =Y)) (CLEAR =Z)) (#DEPOSIT (ON (=X =Z)) (CLEAR =
Y)) (SAY)) ((=Y . C) (=X . A) (=Z . B)))
((P3) ((#DELETE (ON (=X =Y))) (#DEPOSIT (ONTABLE =X) (CLEAR =Y)) (SAY))
((=Y . C) (=X . A)))

###CONFLICT-RESOLUTION-START###

(ONE-RULE-IS-FIRED)
((01) ((REPLACE) (STOP)) NIL)

(THE RULE Q1 IS SELECTED)

CRITICALITY-VALUES:
(8. 10. 9.)

###CONFLICT-RESOLUTION-IS-DONE###

(NOW-THE-CONFLICT-SET-IS-ARRANGED-AS-FOLLOWS)
((P2) ((#DELETE (ON (=X =Y)) (CLEAR =Z)) (#DEPOSIT (ON (=X =Z)) (CLEAR =
Y)) (SAY)) ((=Y . C) (=X . A) (=Z . B)))
((P3) ((#DELETE (ON (=X =Y))) (#DEPOSIT (ONTABLE =X) (CLEAR =Y)) (SAY))
((=Y . C) (=X . A)))
((P1) ((#DELETE (ONTABLE =X) (CLEAR =Y)) (#DEPOSIT (ON (=X =Y))) (SAY))
((=X . B) (=Y . A)))

(THE RULE P2 IS SELECTED)

>>>THE-CURRENT-STATE-DESCRIPTION<<<
((CLEAR C) (ON (A B)) (ON (C D)) (ONTABLE B) (CLEAR A) (ONTABLE D) (HAND
EMPTY))

THIS-STATE-OCCURS-ON-THE-PATH-BACK-TO-THE-INITIAL-STATE?!

(THE RULE P3 IS SELECTED)

>>>THE-CURRENT-STATE-DESCRIPTION<<<
((CLEAR C) (ONTABLE A) (CLEAR B) (ON (C D)) (ONTABLE B) (CLEAR A) (ONTAB
LE D) (HANDEMPTY))

++++++++++++++++++++++++++++++++++++++
```

図 9 ブロック問題の繰り返しサイクルの例

Fig. 9 An example of the recognize-act cycle with backtrack of the block problem.



WM=((1 1) (5 2) (8 3) (6 4)
(3 5) (7 6) (2 7) (4 8))

図 10 8-クイーン問題の1つの解

Fig. 10 A solution to the eight queens problem.

辺の関数が評価され，その後停止する．

5.2.2 8-クイーン

HIPS の複合プログラミングシステムの例として，8-クイーンについて述べる．HIPS の階層性は用いられていないが，従来のプログラミング言語と対比して興味あるプログラム例となっている．問題は，**図 10** のようにチェスの 8×8 盤上に飛車と角の両方の動きが可能なクイーンを8つ，互いに取りあうことのできない位置に置くことである．WM として与えるチェス盤上の状態表現を次のように定めた．

(COL X)：第 X 列にこれからクイーンを置こうとしていることを表わす．

(X Y)：第X行，第Y列の位置にすでにクイーンが置かれていることを示す．また PM として**図 11** のような P1～P9 のルールを用意した．

ここで，P 1～P8は実際に解を見つけるためのルールで，P9 は解が求まったときに処理を終えるためのルールである．また，ルールの LHS における関数 *OK はユーザ定義のマッチ関数であり，注目する位置にクイーンを置くことができる時にT，できない時 NIL をとる．各ルールは，WM とのパターンマッチングが成功し，かつ関数 *OK の値が T になった時のみ起動可能となる．ルール P1～P8は，行番号が異なるだけであとは全く同じであり，たとえば P1は，第1行第 =X 列にクイーンを置けるなら (COL =X) を消し，(1 =X) と (COL =X+1) を WM に置け，と読むことができる．このように図 11 のルール表現は，概念的にきわめてコンパクトであり，バックトラック等のコントロールは自動的に行われるため，従来のプログラミング言語によるものと比較して理解しやすい表現になっている．図 10 は，初期状態として ((COL 1)) を与えた場合の結果である．たとえば，最初は，ルール P1 が発火され，RHS において WM から (COL 1) が除かれ，(1 1) 及び (COL 2) が WM に加えられる．

また，SORTQ は，WM の要素を列の順番に並べ換えるための RHS 関数である．

6. むすび

階層的プロダクションシステム HIPS を紹介し，その実行例を示した．HIPS は2次元の階層構造をもつ複数の基本 PS から構成され，人間が持っているよう

```
((((P9 ((COL 9))--->((DELETE (COL 9))(STOP)))
   (P1 ((COL =X) (*OK 1 =X))--->((#DELETE (COL =X))(#DEPOSIT (1 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY)))
   (P2 ((COL =X) (*OK 2 =X))--->((#DELETE (COL =X))(#DEPOSIT (2 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY)))
   (P3 ((COL =X) (*OK 3 =X))--->((#DELETE (COL =X))(#DEPOSIT (3 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY)))
   (P4 ((COL =X) (*OK 4 =X))--->((#DELETE (COL =X))(#DEPOSIT (4 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY)))
   (P5 ((COL =X) (*OK 5 =X))--->((#DELETE (COL =X))(#DEPOSIT (5 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY)))
   (P6 ((COL =X) (*OK 6 =X))--->((#DELETE (COL =X))(#DEPOSIT (6 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY)))
   (P7 ((COL =X) (*OK 7 =X))--->((#DELETE (COL =X))(#DEPOSIT (7 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY)))
   (P8 ((COL =X) (*OK 8 =X))--->((#DELETE (COL =X))(#DEPOSIT (8 =X))
                                 (*DEPOSIT (LIST 'COL (ADD1 =X)))(SORTQ)(SAY))))))
```

図 11 8-クイーン問題のルールのリスト (PM)

Fig. 11 List of the rules for the eight queens problem.

な問題固有の階層的知識，及びそれらの知識り使い方に関するヒューリスティックな知識をすべてプロダクションルールの形で表現することを可能にしている．さらに，ルールに任意のリスプ関数を書くことができることから，通常のリスプシステムを完全に包含したプログラミング言語となっている．このことよりHIPS のプログラミング言語としての能力は高められたが，その反面 PS としての特長が減少するのではないかとの危具がある．しかしながら，現実の問題には，PS 流のルールの形式では表現が困難な部分が少なくなく，それを無理してルール表現するよりもむしろリスプ流の関数表現で補う方がより自然なプログラミングが可能となり，ユーザが少し注意さえすれば (**5.1** の例のように，リスプ関数を多用せず，概念的にコンパクトなものに限定すれば) PS の特長を損なうことはないと考えられる．このように HIPS は既存の PS の自然な拡張であるとともに，リスプを含む複合的プログラミング言語となっており，人工知能におけるプログラミング (ユーザが持つ知識の埋込み等) がかなり容易になるものと期待される．

参 考 文 献

1) Davis, R. and King, J.: An overview of production systems, Machine Intelligence, Vol. 8, pp. 300-332 (1977).
2) Davis, R.: Meta-Rules: Reasoning about Control, Artificial Intelligence, Vol. 15, pp. 179-222 (1980).
3) Davis, R.: Content Reference: Reasoning about Rules, Artificial Intelligence, Vol. 15, pp. 223-239 (1980).
4) Erman, L. D. and Lesser, V. R.: A multilevel organization for problem solving using many, diverse, cooperating sources of knowledge, Proc. of the 4th IJCAI, pp. 483-490 (1975).
5) Feigenbaum, E. A., Buchanan, B. G. and Lederberg, J.: On generality and problem solving ——A case study involving the DENDRAL program, Machine Intelligence, Vol. 6, pp. 165-190 (1971).
6) Goldstein, I. P. and Grimson, E.: Annotated production systems ——A model for skill acquisition, Proc. of the 5th IJCAI, pp. 311-320 (1977).
7) Hayes-Roth, F. and Lesser, V. R.: Focus of attention in the Hearsay-II, Proc. of the 5th IJCAI, pp. 27-35 (1977).
8) Kanade, T.: Model representations and control structures in image understanding, Proc. of the 5th IJCAI, pp. 1074-1082 (1977).
9) Moran, T. P.: The symbolic imagenary hypothesis: A production system model, Computer Science Department, Carnegie-Mellon Univ. (1973).
10) Rychener, M. D.: Control requirements for the design of production system architectures, Proc. of the Symposium on Artificial Intelligence and Programming Languages, pp. 37-44 (1977).
11) Sacerdoti, E. D.: Planning in a hierarchy of abstraction spaces, Artificial Intelligence, Vol. 5, pp. 115-135 (1974).
12) 佐藤泰介：プロダクションシステムの試作とその使用経験，情報処理学会，人工知能と対話技法研資 3-1 (1978).
13) Shortliffe, E. H.: MYCIN: A rule-based computer program for advising physicians regarding antimicrobial therapy selection, Stanford Univ. Computer Science Report, CS-74-465 (1974).
14) Soloway, E. A. and Riseman, E. M.: Levels of pattern description in learning, Proc. of the 5th IJCAI, pp. 801-811 (1977).
15) 辻井潤一：プロダクションシステムとその応用，情報処理，Vol. 20, No. 8, pp. 735-743 (1979).
16) Waterman and Hayes-roth: Pattern-Directed Inference Systems, Academic Press, New York (1978).

(昭和 56 年 6 月 11 日受付)
(昭和 56 年 10 月 7 日採録)