資　料

第５世代のコンピュータ

研究開発計画・付属資料

昭和５７年３月

JIPDEC

財団法人　日本情報処理開発協会

この報告書は，日本自転車振興会から競輪収益の一部である機械工業振興資金の補助を受けて昭和５６年度に実施した「第５世代電子計算機に関する内外技術動向調査」の成果をとりまとめたものであります。

第５世代のコンピュータ
研究開発計画・付属資料

この付属資料は，第５世代コンピュータ開発の開始にあたって，基礎技術の選択に対する内外からの基本的批判 ― すなわち５Ｇ核言語のモデルとして何故PROLOGを採用したのか，日本は人工知能および知識工学の分野での研究・経験の蓄積が少ないにもかかわらず，知識ベース・システムの開発が行えるのか，等 ― に対する，回答への試みとなる３つの論文より成るものである。

各論文の著者は，本調査研究に御協力いただいた方々であり，それぞれの観点・立場から上にあげた批判に対する回答を試みている。

1.では，推論機能およびLISPとの比較の観点から，PROLOGの核言語モデルとしての妥当性を論じている。

2.では，核言語としてPROLOGの拡張機能を探るために，いくつかの典型的な既存言語を取り上げ，PROLOGとの比較を行っている。

3.は，従来のアルゴリズムを主体とするシステム設計とはかなり異なったアプローチが必要な知識ベース・システムの開発方法について，その方略を整理し，今後の知識ベース・システム構築に資することをねらいとしている。

目　　次

1. PROLOGの核言語モデルとしての妥当性 —WHY PROLOG？—

1. PROLOGの核言語モデルとしての妥当性

—— Why PROLOG ? ——

1.1 はじめに

第5世代コンピュータ・プロジェクトは，知識情報処理を指向している。そこでの一番もとになる処理は，推論である。推論は，第5世代コンピュータでは，今のコンピュータでの加減乗除算などの基本演算と同様の役割を果たすものと考えられる。

コンピュータに推論能力を持たせるというと，いかにも人間と同じように物事を理解し，自分でいろいろと考えることができなければならないと思いがちであるが，実はもう少しレベルの低い，単純な能力を考えている。コンピュータは，人間があらかじめ作ったプログラムに従ってしか動かないことは，推論処理の場合でも同じである。通常の数値計算などと異なるのは，推論を行うときの基本操作を抽出してプログラムの形で与えておく点である。そうすれば，一見，複雑な論理的思考を行わせることが可能となる。

推論の中でも，演繹と呼ばれる推論は，数値計算と同様，その処理を一定の規則に基づいた機械的手続きによって実現できる。ところが，帰納，類推，日常的推論などは，あいまい性を含み，単純な機械化は困難である。その場合には，コンピュータを使っている人間をシステムの一部として考えることも必要となる。

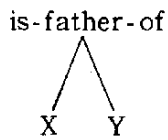
本稿では，以下，1.2で，演繹的推論のメカニズムを示し，1.3でより高度な推論のメカニズムについて述べることにする。そして，1.4でそれらの機能を実現する上でなぜPROLOGをモデルとした新言語が必要となるかについて述べる。

1.2 演繹的推論のメカニズム

コンピュータによる推論のメカニズムを，数値計算のメカニズムと対比させて考えてみよう。まずコンピュータに表現の規則を与えることが必要である。いま，XはYの父親であることを

is-father-of （X，Y）

と表すとする。この表現は，"is-father-of "，"（"，"X"，","，"Y"，"）" から成る記号列と考えてもよいが，つぎのような木で表すと，もっと分かりやすい。それは，is-father-of を根とする2進木



である。たとえば，" 家康は秀康の父である "，" 家康は秀忠の父である "，" 秀忠は家光の父である " は，それぞれ，図1－1のa，b，cのような木構造で表される。

いま，" 祖父 " という概念を父子の関係を使って定義すると，XがZの父でZがYの父であれ

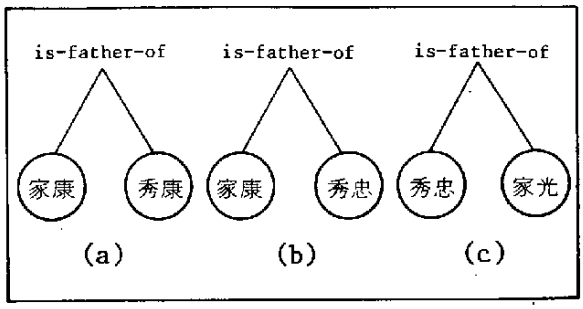


図１－１　is-father-of 構造の例

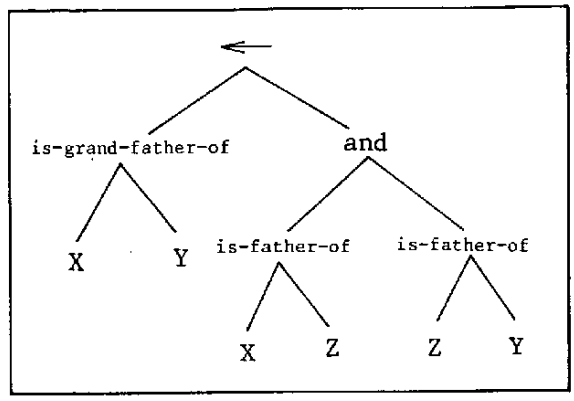


図１－２　概念“祖父”の定義の構造

ばＸはＹの祖父であるから，祖父－孫の関係を表す述語を is-grand-father-of とすると，

is-grand-father-of （Ｘ，Ｙ）
←is-father-of （Ｘ，Ｚ）and　　(1)
is-father-of （Ｚ，Ｙ）

と表すことができる。ここで，Ｘ，Ｙ，Ｚは，変数を表す。

この定義自身，推論操作にとっての一つのデータである。この定義の構造は前の is-father-of の構造より複雑であるが，同様に図１－２のような木構造によって表される。

いま，図１－１の三つの関係から

is-grand-father-of（家康，家光）　　(2)

が成り立つかどうかを調べてみることを考える。それを調べるには，(1)式から

is-father-of（家康，Ｚ）and
is-father-of （Ｚ，家光）　　(3)

を満足するＺなる人物を求められればよいことが分かる。(3)式を新たな問題とすべき根拠は，論理学でよく知られた三段論法である。三段論法は，「ＡならばＢである」と，「Ａである」ことから「Ｂが成り立つ」ことを保証する推論規則であるが，この規則を用いると，「Ｂを示したい」ときに「ＡならばＢである」ことが分かっていれば「Ａを示せばよい」ことが導き出せる。この論理操作のことを resolution と呼ぶが，resolution 操作をよくながめてみると，それは，構造(1)と構造(2)から新たな構造(3)を作り出していることが分かる。推論操作としてコンピュータが実際に行っているのは，ここで示した構造の操作である。この構造の操作は，数値計算における加減乗除算のように一定の規則によって機械的に行われる。そして，その操作が施された後で得られた構造を，表現の規則に従って解釈してみると，まさしく意味的にも推論を行った結果と一致するわけである。これは，推論操作をそのように定めたから当然のように思えるが，重要なことは，一階の述語論理で表現できる問題であれば，いかなる問題でもまったく機械的な処理によって解くことができる点である。

さて，推論操作を先に進めてみよう。新たに作られた問題(3)の前半部分 is-father-of(家康，Z)は，図１－１の(a)から，Z＝秀康とすれば満足される。すると，後半部分は，is-father-of（秀康，家光）となるが，これは事実に反し，実際にそのようなデータは存在しない。すなわち，ここに至って推論操作は失敗してしまったわけである。この失敗は，回復できるであろうか。失敗の原因を考えてみると，問題(3)の前半部分is-father-of（家康，Z)で，Zに対する選択が誤っていたことが分かる。推論操作を行うコンピュータ・プログラムは，もし途中で推論に失敗したら，それ以前の操作で他に選択すべき可能性があるかどうかを順次実行を遡って調べていき，新たな選択を行うようになっている。この操作は，自動後戻り制御と呼ばれ，人間の行う試行錯誤機能の機械化となっている。

さて，推論操作は，失敗によって自動後戻りが起こり，そして，is-father-of（家康，Z）のZに対して，もう一つの可能性であるZ＝秀忠が選ばれる。そして，引き続いてis-father-of(秀忠，家光）が調べられる。今度は，図１－１(c)にある通り，この式が満足され，すべての部分問題が解けて，is-grand-father-of（家康，家光）の正しいことが示されたことになる。

ここで，推論操作は全く機械的に実行されることに改めて注目したい。解くべき問題が与えられたとき，それをどう解いていくか，問題解決の過程で新たに問題を作り直すが，それをどのように作るか，試行錯誤をするときにどこまで実行を遡るか，これらはすべて決められた規則に従って，すなわち，あらかじめ用意されたプログラム通りに動く。すなわち，その規則の基になっているのが，一階述語論理の証明手続きについての理論なのである。われわれが問題を解くときには，その問題が表している意味内容をも考えて考察を進めていくが，機械的処理では，構造のみを操作している。

1.3　あいまい性を含む推論のメカニズム

人間が日常行っている推論は，必ずしも厳密なものでなく，あいまい性を含んでいる。診断などでは，データの不足から絶対に正しい判断は下せないことが多く，その場合，判断にはあいまいさがつきまとう。このような推論は，演繹的推論のように問題に対する答だけを求めようとしたのでは機械化が不十分である。それではどうすればよいか。推論の結果得られた結論は，当座のもの，仮の結論であって，後でより正しそうな結果が得られれば，いつでも判断を翻せるようにしておけばよい。そのためには，仮の結論を導いた理由あるいは根拠を結論と共にコンピュータで把握していることが必要となる。そうしないと，誤りの原因を的確に見出すことができないからである。

例として，つぎのような推論を考える。

①通常，日中晴れていれば，太陽は輝いて見える。

②今は日中で，かつ晴れている。

③今は，太陽は輝いて見える。

ところが，たまたま，今の時点で，皆既日食が起こったとする。そのとき結論③はもはや正し

くない。それは,

④今，皆既日食が起こっていれば，太陽は輝いて見えない。

からである。結論③をくつがえすためには，結論③を導き出すに至った根拠，すなわち①および②について，誤りの原因を調べることが必要となる。ここで，①の命題は，常に正しい命題ではないことが分かる。①が成り立つのは，あくまで"通常"であり，例外的に成り立たないことがあり得る。そして，たまたま今の時点がその例外であったわけである。日常的推論では，ありとあらゆる可能性を厳密に調べる手間を省くために，はじめは例外的に起こる事柄を考慮に入れずに推論を進めていく。そして，後で矛盾に気がついた時，その結論を翻すわけである。

ここで行われている推論の仕組みを論理的に見ると，問題の中身について考えている部分と，問題を解いていく過程について考えている部分の二つが含まれていることが分かる。すなわち，結論③を導き出す推論は問題の中身を扱っており，誤りの原因をつき止める推論は問題解決の過程を扱っている。すなわち，後者の推論は，前者に比べて一段レベルの高い対象を扱っているので，このような推論をメタ・レベルの推論という。

あいまい性を含む推論の例としては，このほかに，病気の診断システムなどがある。MYCINと呼ばれるシステムでは，専門医の診断に用いる知識を，あいまい性をも含めてルール化している。その結果，得られる結論もあいまいなものとなるが，MYCIN では診断の理由を，このシステムを利用する医師に示す機能を持っている。その診断理由は，専門医によって作られたルール(の集まり)である。このような説明機能により，人間はコンピュータの下した診断を評価できるわけである。そして，この説明機能を実現する場合にも，推論過程についての知識，すなわちメタ・レベルの知識を必要としている。

1.4　何故 PROLOGか

これまでは，コンピュータがいかにして推論を行うかを，二つのレベルの推論，すなわち演繹的推論と，あいまい性を含んだより高度な推論について述べてきた。

本節では，これらの推論機能を実現する上で，何故 PROLOG が適しているかを明らかにしたい。それによって，第5世代コンピュータの核言語のモデルとして PROLOG を選んだ根拠としたい。

議論の土台として，PROLOGの対抗馬を想定すると，現在の知識工学応用システムを考えれば，LISP が最も自然である。ゆえに，ここでは，何故 LISPでなく PROLOGか，に焦点を当てて論じることにする。

第1の観点は，知識情報処理に対する適性である。LISPは，この点に関しては，大きな実績をもっている。その実績は，主として，米国の人工知能研究により作られてきた。何故，LISPが人工知能研究で主役を演じてきたのか。それは，人工知能，とくに知識情報処理が，数値でなくリスト構造の処理が中心であり，LISP程自由にリスト構造を操作できる言語が他になかったからで

ある。

しかしながら，リスト構造の操作に関しては，PROLOGの能力はLISPとほとんど変らない。PROLOGは，その上に，前節でも述べたように，問題解決・推論のための基本機能を有している。LISP上の問題解決・推論システムは，まず，この基本機能をプログラムで作ることから始まる。PROLOGでは，それが不要であるばかりか，その基本機能は十分エレガントに作られており，機能が優れている割には，プログラムの効率は落ちていない。ゆえにLISP上の問題解決システムに比べて，大変能率のよいシステムがPROLOG上で実現可能である。

第2の観点はプログラミング言語としての比較である。LISPは，リスト処理が得意であるが，リストの作成，要素のアクセスは，基本命令 cons，car，cdr を用いて組立てていく。たとえば，リスト X＝(A (B C)) からCをアクセスするには，

car〔cdr〔car〔cdr〔x〕〕〕〕

としなければならない。この意味でLISPはリスト処理の「アセンブラ言語」と言うことができるであろう。

一方，PROLOGでは，リスト処理はすべてパターン照合時になされる。リスト処理の記述は，リスト構造のパターンがどのように変化するかを書けばよい。たとえば，前の例でリスト(A(B C))からCを取り出すには，このリストとパターン(*x (*y *z))を照合させて，*z を見ればよい。

もう1つの例として，関数あるいは手続きの呼出しと引数の評価の仕組について考えてみよう。LISPでは，関数への実引数の受け渡しは λ-binding によって行われる。すなわち，

f＝ λx. car〔cdr〔car〔cdr〔x〕〕〕〕

によって，関数 f の実引数は，仮引数 x に代入されることが示される。ゆえに，

f〔(A(BC))〕
＝car〔cdr〔car〔cdr〔(A (BC))〕〕〕〕
＝C

となる。

一方，PROLOGでは，手続きの呼出しは，パターン照合によって行われる。PROLOGにおけるパターン照合は，resolution での単一化（または統一化：unification）である。この単一化によって，呼出し側のパターンと，呼出される手続きにつけられたパターンが照合され，変数引数に対する代入が行われる。ところが，この代入は，LISPのように必ず呼出される側の変数になされるとは限らない。場合によっては，呼出し側が変数で，呼出される側が定数かも知れない。あるいは，もっと特別な場合として，一部分のみが変数であるようなリスト構造も引数となり得る。このように引数の評価はすべて単一化でなされるが，その機能は豊富である。手続き呼出し時に，定数が渡される場合には call by value のように働き，逆に変数が渡されて，手続き呼出し時，あるいは手続き実行時にその変数の値が決まる場合には call by reference のように働く。

大規模なシステムの保守管理の点から考えても，PROLOGはLISPに比べて有利である。PROLOGは，LISPに比べて，可読性がずっと良い。PROLOGプログラムでは，各手続きは文章になっており，その意味は，それを構成する各述語の意味から容易に推察できる。たとえば，つぎのような単純なappendプログラムについて考えてみよう。

```
1.  append (nil, X, X).
2.  append (U.X, Y, U.Z) ← append (X, Y, Z).
```

ここで，大文字は変数を表すものとする。述語append（P，Q，R）の意味は，リストPとリストQをつなげたリストはRであることを表す。このappendの意味から，手続き1.は「nilとXをつなげるとXになる」ことを表し，手続き2.は，「もしXとYをつなげてZになれば，U.XとYをつなげればU.Zになる」ことを表していることが分かる。

プログラミング言語としてのPROLOGの最大のメリットは，ソフトウェアの生産性が高い点である。ソフトウェアの生産性は，定量化も明確でないが，その向上にはソフトウェアの設計，コード化，保守・管理のすべての面でのコスト・パフォーマンスの向上が必要不可欠である。PROLOGは，この条件を満していると言えよう。コード化，保守・管理の面での利点は，いくつかすでに述べた。ソフトウェアの設計については，他の設計技法の手助けが必要となるであろうが，言語が高水準であること，概念レベルの関係が，そのまゝプログラム化し得る点など，潜在的な利点は見逃せない。

ソフトウェアの品質の点でも，PROLOGプログラムは，大きな利点をもっている。論理的に整理されているので，数学的扱いが容易で，その分プログラムの検証も楽になっている。さらに，最近，効率の良い虫取り法の開発が報告された。それは帰納的な推論の研究から派生したもので，ユーザとの対話を通して，実行の道筋を遡上り，どこに誤りの源があるかを確かめる方法である。

第3の観点としては，ハードウェア化を考えてみよう。LISPマシンは，高級言語マシンとして成功した数少ない例の1つである。その成功の原因は，専用ハードウェア化によるコスト・パフォーマンスの大幅な向上である。何故コスト・パフォーマンスが向上したかと言えば，記憶管理や，基本命令，インタプリタなど，多くの部分がハードウェア化可能であったからである。

PROLOGマシンも，同様の理由で，高級言語マシンとして成功することが有望視されている。

PROLOGの場合は，LISPより一層高度で時間のかかる処理部分をハードウェア化することが可能であり，その効果は一層大きいであろう。

第5世代コンピュータ・プロジェクトでは，専用の並列マシンの開発をねらっている。並列マシン開発でのLISPとPROLOGの比較は，もっと研究が進んだ段階でないと，正確な答は得られないであろう。現段階で言えることは，純粋に関数的な部分は，データフロー方式のような並列アーキテクチャに基づく並列マシンの実現可能性が見えている点である。PROLOGについては，

手続きを格納しているデータベースの更新を除く部分は副作用を伴わない計数になっているので，同様の可能性がある。

並列マシンが開発されると，それを利用するためには，これまでのソフトウェアの蓄積の利用はあきらめなければならない。とすれば，ＬＩＳＰのささえているソフトウェア遺産は，何の意味もなくなってしまう。

1.5　おわりに

本稿では，第５世代コンピュータにおける問題解決・推論機能として，どのようなものを考えているのか，それらを実現するためには，どんなコンピュータを作ればよいのか，とくに，その核言語としてどのようなものを考えればよいのか，について論じてきた。

核言語のモデルとしては，PROLOGを採用したが，その決定については，賛否両論があることも確かである。本稿の1.4では，特にこの点に焦点を当てて，「何故 PROLOGなのか」の問いに答える努力を試みた。確かにPROLOG は未完成な言語であり，ＯＳの記述が困難である等，その記述能力に対して不安をもつ声も聞えるが，経験の蓄積やプログラミング技法の開発とともに，言語の適切な拡張の可能性を予測して，これらの困難性に打ち勝っていけるであろうと考えている。

（電子技術総合研究所　古　川　康　一）

2. PROLOGと他言語との比較

2. PROLOGと他言語との比較

2.1 目的と背景

FGCS（Fifth Generation Computer System）はKIPS（Knowledge Information Processing System)を目指しているが，その基本ソフトウェアと応用ソフトウェアを記述する中核となる言語（以降この言語をFGCSの核言語と呼ぶ）として，PROLOG＋αを想定している。本稿では，核言語のベースとしてPROLOGを採用することの妥当性を検討し，「α」として付け加えるべき機能を探るために，いくつかの典型的な既存言語とPROLOGの比較を行う。

上述の目的に照して比較を行う項目と比較対象とする言語を選ぶために，現世代のコンピュータ・システムをとりまく状況と動向を概観する。

第1に，ソフトウェア危機と言われている状況が現実としてある。歴史をひもとけば判るように，コンピュータ言語の誕生は輝かしいものであった。初期のコンピュータにおけるプログラミングの多大な労力を，オートコーダ（初期のコンパイラはこう呼ばれた）は劇的に軽減した。そのため，当時開発されていたコンピュータ言語の枠組に疑問を持った人はいなかったように見える。それらの言語は，止むを得ないことであるが，コンピュータ自身の枠組と同様にチューリング・モデルに基づいたもの即ちノイマン型であり，人間の思考の表現し易さを第1に考えたものではなかった（勿論現在でもこの傾向は変ってはいない)。計算モデルとしては，チューリング・モデルの他に，それと同値なものとしてポストのモデルが知られていたが，ハードウェア実現上の容易さから，当時の技術ではチューリング・モデルを採らざるを得なかったようである。ノイマン型言語とソフトウェア危機の間に一義的な因果関係を立証することはできないかも知れないが，数十年後の現在，コンピュータ産業が，大規模ソフトウェアの開発/保守が極めて困難であることを認識しながらも，その有効な解決策を見出していないのは事実である。

第2に，コンピュータの応用分野は，上述したソフトウェア危機にもかかわらず，拡大の一途をたどっている。コンピュータが最初に利用された科学技術計算分野でも，宇宙開発や原子力開発の急速な進歩に伴って，必要とされる計算量はとどまるところを知らない勢いで伸びている。同時に，科学技術計算以外の分野でも，単なる事務計算をはるかに超えて，扱う情報は大量化しており，大規模データベース・システムなどが必要となっている。さらに，今後は単に大量化するだけでなく情報の質が高度化し，それらに相反するものであるが，省力化の要求も加わると予想される。即ち，高度知識情報処理システムが必要とされるのである。最近では，人工知能研究の応用として知識工学が発展しつつあり，典型的な例として機械翻訳システムや医療用コンサルテーション・システムなどが試作されている。欧米諸国と密接な経済関係を持ちながら，日本語という特殊な言葉に悩む我が国にとって，性能の良い日英機械翻訳システム開発は急務であろう。また，大量の専門家を必要としながら専門家の養成に金と時間がかかるため慢性的な人材不足に悩む分野では，専門家の援助・代用を務める実用的コンサルテーション・システムの出現を待ち

望んでいるであろう。

第3に，コンピュータの利用形態が，中央集中型から分散型に移行する動向が見られる。これは，上述した情報の大量化・高度化と無関係ではない。大量化・高度化した情報は，ある限度を超えると分散せざるを得なくなる。すべてを中央システムで処理するには，負荷が大きくなり過ぎるからである。今後のコンピュータ・システムの方向として，全体を制御するコンピュータと業務（応用分野）別の専用コンピュータがネットワークで結ばれたものを予想する人もいる。

最後に，ソフトウェア工学の発展が挙げられる。1970年前後からダイクストラの構造化プログラミング，ヴィルトの段階的詳細化，データとそれに対する手続きをまとめるカプセル化の概念とそのまとまりを対象物として扱うクラスの概念をもつ言語SIMULA67（ダール他）などが現れて来た。これらは，大規模なプログラムを明確な方法論に基づいて人間の思考に沿った形で構成して行こうとする欧州の思索家的コンピュータ学者の努力の結果である。これらを背景にして構造化プログラミングに基づいて問題を段階的に詳細化するときの各段階は，抽象化のレベルと呼ばれるようになった。さらに，複数のデータと複数の手続きのカプセル化機能を持つモジュール構造に抽象化のレベルを導入することによって，データの抽象化，即ち抽象データ型の概念が作られた。

2.2 比較対象とする言語

前項で述べた現在のコンピュータ・システムをとりまく状況と動向の概観から，以下の言語を比較対象に取り上げる。

(1) Ada

米国・国防省が組み込みシステム記述用に開発した言語であり，70年代のソフトウェア工学の成果を実用性とうまくバランスをとって取り入れていると言われている。ノイマン型の最後の実用言語とも言われている。

(2) LISP

マッカーシーによって開発されたリスト処理用言語であり，知識工学の基礎である人工知能研究では当初から使われている。種々の機能を加えた多様なLISPが作られている。コンサルテーション・システムでは記述言語はLISPを用いている例も多い。

(3) Small Talk

XEROXで開発された。新しい型のパーソナル・コンピュータ用言語として注目されているものである。SIMULA67，CLUなどと同様な対象指向型言語であり，ACTORモデルを言語仕様に取り入れている。ACTORモデルは，プログラミング言語のセマンティックス，並列処理，分散処理

などに対して，強力な理論的・概念的枠組を提供すると言われているものである。

(4) Iota

京都大学で開発されたものであり，プログラムを抽象化のレベルに従って階層的に記述し検証することを目的として設計されている。仕様記述・検証には，多ソート1階述語論理が用いられている。

2.3 比較項目

前項と同様な観点から，以下を比較項目として取り上げる。

(1) 駆動原理

2.1の「目的と背景」で触れたように，歴史的経緯から現存する殆んどの言語は駆動原理としてノイマン型コンピュータの枠組をそのまま採用しているが，それは必ずしも人間の思考の表現に適したものではない。最近では，非ノイマン型の駆動原理を持つ言語が出て来ている。PROLOGやSmall Talkがその例である。

ノイマン型言語と非ノイマン型言語の最も本質的な相違は，駆動原理にある。ノイマン型言語が現在のコンピュータと同様に文あるいは命令の上を「制御が走って」逐次に実行して行くのに対して，非ノイマン型言語は「制御が走る」という概念を持たない。種々のバリエーションはあるが，いずれにしても「制御が走る」という物理的・機械的な概念ではなく，より論理的な，より抽象度の高い駆動原理に基づいている。

(2) モジュラー・プログラミングのし易さ

モジュラー・プログラミングには2つの種類がある。1つは，抽象化のレベルに沿った階層的なモジュール化である。これは垂直型のモジュール化と言える。他の1つは，水平型のモジュール化とも言うべきもので，機能モジュール即ち部品を横方向にまとめるモジュール化である。いずれにしても，モジュラー・プログラミングは，2.1項でも述べたように，70年代のソフトウェア工学の発展から生まれた重要なプログラミング上の思想である。

(3) 推論アルゴリズムの記述のし易さ

KIPSの対象とする高度知識情報処理では，例として機械翻訳システムやコンサルテーション・システムを考えて見れば容易に推測されるように，人間が行っている高度な判断をプログラムでかなりの程度まで行わなければならない。従って核言語は，何よりも先ず推論アルゴリズムの書き易い言語でなければならない。これは，核言語が絶対に満さなければならない最も重要な性質であると言える。

(4) 並列処理のし易さ

この性質も前項目と同様に機械翻訳システムやコンサルテーション・システムで現われて来る処理を考えてみれば，その必要性が理解できる。そのようなシステムでは，多くの可能性を調べ尽すことが処理時間の多くを占めると考えられる。小規模な場合には，可能性を逐次順番に調べていても事足りるが，大規模になると可能性の数が飛躍的に増えるため，個々の可能性探索を並列に行うことが実用的性能を得るためには不可欠になる。従って，核言語も並列処理の記述力を十分に持っていなければならない。

(5) データベース・アクセスのし易さ

高度知識情報処理では，人間の知識をデータベース（知識ベース）に蓄積する。実用的システムでは，その知識ベースはかなりの大きさになる。核言語は，知識ベースを容易に高速にアクセスし得るものでなければならない。

(6) 分散処理のし易さ

2.1でも述べたように大規模な知識ベースはある限度を越すと必然的に分散する傾向を持つ。また，異なる機関・組織が蓄積したデータ（知識）を，お互いに重複して持つことなく利用し合うためにも分散処理は避けて通れない。知識ベース・システムは，大規模化するにしたがって，必然的に分散型知識ベース・システムへと移行せざるを得ない。従って，核言語には分散処理記述能力も必要となって来る。

(7) 検証のし易さ

ソフトウェア危機の唯一最大の原因は，ソフトウェアの信頼性を保証することが極めて困難であることにある。その困難さは，ソフトウェアの規模が大きくなるに従って，指数関数的に増大する。70年代のソフトウェア工学の中から生まれて来た考え方の中に，(2)で挙げたモジュラー・プログラミングの他に，検証しやすい言語を用いるというものがある。

検証のためにはプログラムの意味論が不可決であるが，それらに関してはプログラムの実行前後で成り立つ命題(predicate)に着目するダイクストラのWP（Weakest Precondition）や，プログラムの正しさの証明体系を1階述語論理の自然演繹体系と類似の公理的推論規則として与えるホア論理，λ論理（λカルキュラス）を端緒として考えられたスコットの表示的意味論（denotational semantics）などの顕著な成果がある。

以上のような検証のための道具は現在まだ実験的な段階にあり，実用的プログラムに有効性を発揮しているとは言い難いが，いずれにしても核言語は大規模な高度知識情報処理システムの開発に用いられる訳であるから，前項までに述べた種々の記述能力の他に，検証し易いという性質も併せ持たなければならない。さらに，プログラムの設計・製造・検証を一貫した過程としてそ

の中に含むプログラム開発支援システムをその上に構築できるものでなくてはならない。

2.4 駆動原理の比較

(1) Ada

ノイマン型言語であるから，駆動原理は文の逐次実行である。言うまでもなく，これは現在のハードウェアが行っている機械命令の逐次実行の概念に基づくものであり，ただ実行の単位が機械命令でなく言語で定める文になっているだけである。2.1でも述べたように，現在のハードウェアがノイマン型になっているのは効率よいコンピュータが容易に実現できるためである。文の逐次実行というノイマン型言語の駆動原理は，ノイマン型コンピュータの上で効率よく稼動する言語を考える限り自然な発想と言える。しかし，それが人間の思考に適合したものであるかどうかは別問題である。

しかしながらAdaは，他のいくつかのノイマン型言語と同様に，非常に洗練された制御構造記述機能を持っている。

- 条件文

  if～ then～ else～ end if;

  if～ then～ else if…… else～ end if;

  case～ of when～ ; when～ ;…… end case;

- 繰返し文

  loop～ end loop;

  for～ loop～ end loop;

  while～ loop～ end loop;

- 分岐文

  exit when～ ;

  goto～ ;

そのために，制御の流れのパターンを人間の言葉に近い形で書くことができる。しかし，これらの制御構造記述文を用いても，制御の流れという機械的概念を消せる訳ではない。

図2－1にAdaのプログラム例を示す。この例は，N要素から成る配列Aの中から変数Vと同じ値を持つ要素を見つけるものである。

```
K : = 0 ;
for I in 1..N loop
   if A ( I ) = V then
         K : = I ;
         exit ;
   end if ;
end loop ;
if K = 0 then
        値Vをもつ要素がなかった時の処理 ;
else
        値Vをもつ要素があった時の処理 ;
end if ;
```

A（I）＝VなるIがあれば，その添字値IをKにセットして，ループをぬける。

図２－１　Adaのプログラム例

(2) LISP

LISPの動作原理は，再帰的に定義された関数の逐次評価である。Adaではif文などで行っていた「判断」は，LISPでは「マッカーシーの条件式」と呼ばれる形式でなされる。これは，再帰的関数の評価過程の中に種々の条件（論理値を結果の値とする関数で示される）を入れて，条件によって評価する関数を変更することにより評価過程を制御するものである。

LISPは，２進木の世界の上に作られている。２進木はＳ式と呼ばれる形式で表現される。Ｓ式は，対象を表すシンボルを括孤で括ったものである。従って，再帰的関数も原則的にＳ式（２進木）を変形・合成・生成する関数である。必要な結果は，Ｓ式計算の途中結果を副作用として取り出すことにより得られる。

図２－２にLISPのプログラム例を示す。再帰的関数の評価のメカニズムの良い例であること，プログラムが小さくて済むので「ハノイの塔」を例として挙げる。なお，LISPのプログラムはプログラム自体が，それが取扱うデータの表現形式と全く同じＳ式で書かれるのであるが，図２－２ではプログラムを読み易くするためにＭ式で書いてある。Ｍ式は，マッカーシーの条件式をほぼそのままの形で書いたものである。

このプログラムはよく知られているので解説の必要はないかも知れないが，LISPになじみのない人を想定して一応解説する。

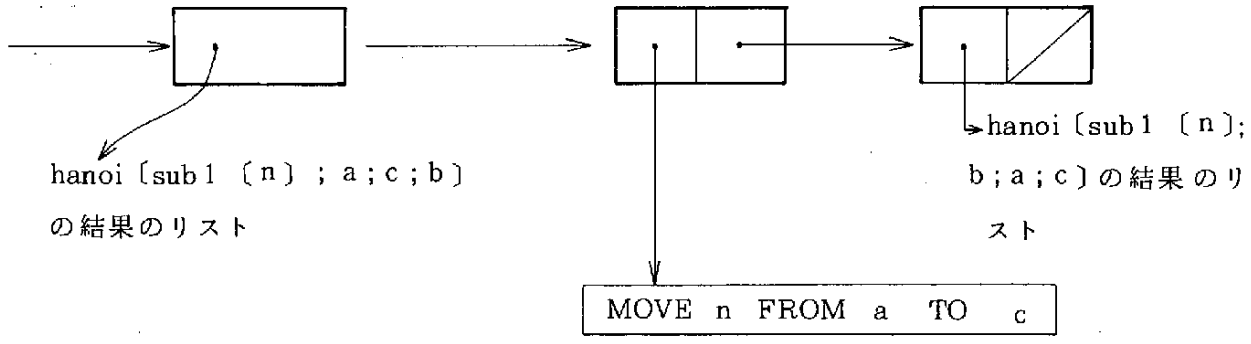
zerop〔ｎ〕は，ｎが０であるかどうか判定して真または偽を値として返す。ｔは，真を表す。append〔ｘ，ｙ〕は，リストｘの内容とリストｙの内容をつなげる。cons〔ｘ，ｙ〕は，リストｙの先頭にリストｘをそのまま追加する。list〔x_1，x_2，…，x_n〕は，x_1，x_2，…，x_nを各

```
hanoi〔n ; a ; b ; c〕
  = 〔zerop〔n〕 → nil ;
     t  →
     append〔
        hanoi〔sub1〔n〕 ; a ; c ; b〕 ;
           cons〔
              print〔
                 list〔
                    MOVE; n;FROM ; a ; TO ; c
                    〕
                 〕;
              hanoi〔sub1〔n〕 ; b ; a ; c〕
              〕
        〕
  〕.
hanoi〔3 ; A ; B ; C〕 ←――― 関数 hanoi を評価せよ，という指示。
```

関数 hanoi の定義

図２－２　LISP のプログラム例

要素とするリストを作る。print〔x〕は，x の内容を副作用として印刷する。print〔x〕の関数としての値は，x のままである。従って，図２－２のプログラムは，次のような２進木を作り出す。



hanoi〔3 ; A ; B ; C〕の評価結果をＳ式で書けば，

```
( (MOVE  1   FROM  A   TO   C)
  (MOVE  2   FROM  A   TO   B)   } hanoi〔2 ; A;C;B〕
  (MOVE  1   FROM  C   TO   B)
  (MOVE  3   FROM  A   TO   C)
```

```
(MOVE  1  FROM  B  TO  A)
(MOVE  2  FROM  B  TO  C)   hanoi〔2 ;B;A;C〕
(MOVE  1  FROM  A  TO  C)   )
```

となる。このS式が作成される過程で，print 関数によって,各リストの要素が作り出されるたびに印刷される。従って，出力結果としては，上述のS式の1番外側の括孤を除いたものが得られる。

なお，LISPには図2－1のAdaのプログラムのような処理も書けるように,配列を宣言する機能や，prog 形式と呼ばれるノイマン型言語の 「制御の流れ」や「文」の概念でプログラムを書く機能もある。prog形式の中では,飛越し文go〔ℓ〕（ℓはラベル）さえも使うことができる。しかし，これらの機能は，LISPで実用的プログラムを書くときに,再帰的関数評価という枠組では能率が悪くなるタイプの処理を，自由リストの消費を少なく効率良く書くために付け加えられた機能である。純粋LISPの観点から見れば,「不純な」機能と言えるかも知れない。

(3) Small Talk

この言語の駆動原理は，メッセージの授受である。計算の主体は対象（object）と呼ばれ，対象はその種類ごとのクラス(class) に属する。逆に言えば，まずクラスが定義されて，そのクラスに属する対象が生成される。生成された対象は，所定のメッセージを受けとり，それに対する反応としてメッセージを返す。各々の対象はメッセージの授受以外は全く関係なく独立に動作する。

Small Talk はパーソナル・コンピュータ用言語として設計されていることもあり，外観はプログラミング言語というよりはコマンド言語である。Small Talkの動作例を図2－3に示す。図2－3では，box（箱)というクラスがすでに定義されているものとしている。クラスboxに対しては，そのクラスの対象が受け取ることができるメッセージの種類として，turn（回転）やgrow（拡大）が定義されている。クラスboxの対象には，対象を識別するための名前をつける。対象が生成されると画面に表示され，turnやgrowなどのメッセージを受けとって，それに従って動作する。この例には現れていないが，前述したように，対象同志でメッセージの授受を行うこともできる。

図2－3の例には「計算」というイメージがないので，Small Talkの対象という概念の基礎になっていると考えられるactorモデルを用いて，メッセージの授受による計算（computation）を例示する。Actorは，Small Talkの対象と同じ概念と考えてよい。Actorの挙動を記述する言語としてはPLASMAがある。繰返し（iteration）により正整数の階乗を計算するactor fact - orialを PLASMAにより記述したものを図2－4に示す。

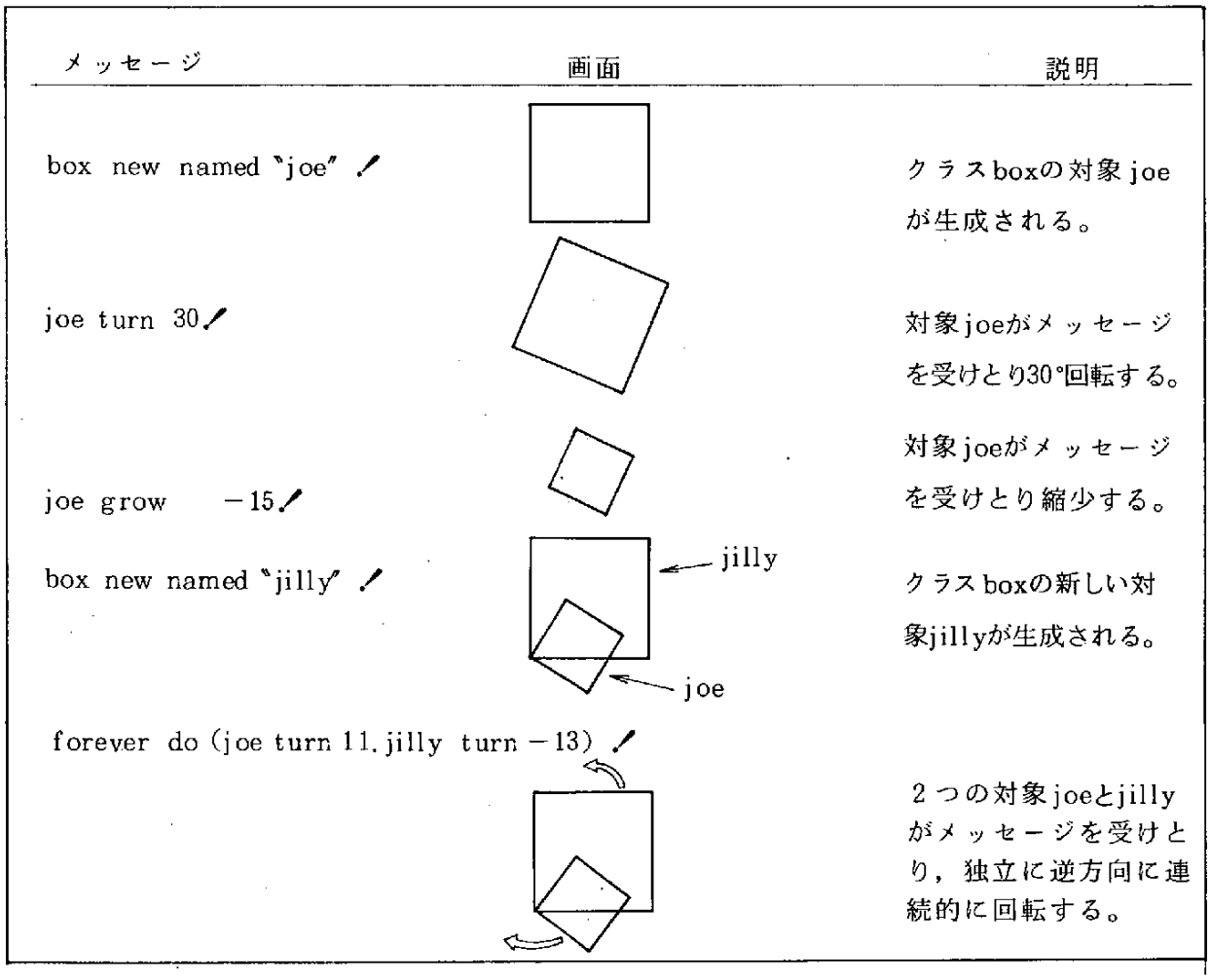


図2－3　Small Talkの動作例

```
(factorial≡
    (≡≡> (request : [= n]  (reply-to: = c ))
        ( (request : [1 n]  (reply-to: c ) ) ==>
            (loop ≡
                (≡≡> (request : [= accumulation
                                  = count]
                        (reply-to: = d ) )
                    (rules  count
                        (≡> 1
                            (d <== (reply: accumulation)))
                        (≡> (> 1 )
                            (loop <==
                                (request :
                                    [ (accumul ation * count)
                                      (count -- 1 )]
                                (reply-to: d ))))
                                                        ①
                                                        ②
                                                        ③
                                                        ④
                                                        ⑤
                                                        ⑤
                                                        ⑥
                                                        ⑦
                                                        ⑦
                                                        ⑧
                                                        ⑧
    ))))))
```

図2－4　PLASMAによるactorの記述例

図２－４の右端に付した番号に沿って各行の意味を以下に示す。

① 記述するactorの名前をfactorialとする。

② factorialが受取るメッセージは正整数ｎであり，factorial がそれに対して応答する相手は他の actor C である。

③ 以下はfactorial の動作を記述する。factorialは，メッセージ〔１ｎ〕を応答先Ｃを指定して，④以下で定義される下位のactorである loopに送る。

④ 下位actorを定義する。名前はloop。

⑤ loopが受取るメッセージはaccumulatianとcountのペアであり，その応答先はｄである。これらは引数として渡される。

⑥ メッセージの中にあるcountによる場合分け。

⑦ countが１ならば， actor ｄ にaccumulationの値をそのまま送る。

⑧ countが１より大ならば， accumulation ＊ count と count－１の２つの値のペアを，応答先actor ｄ を指定して loopに送る。

以上の計算過程を，図２－５に示す。

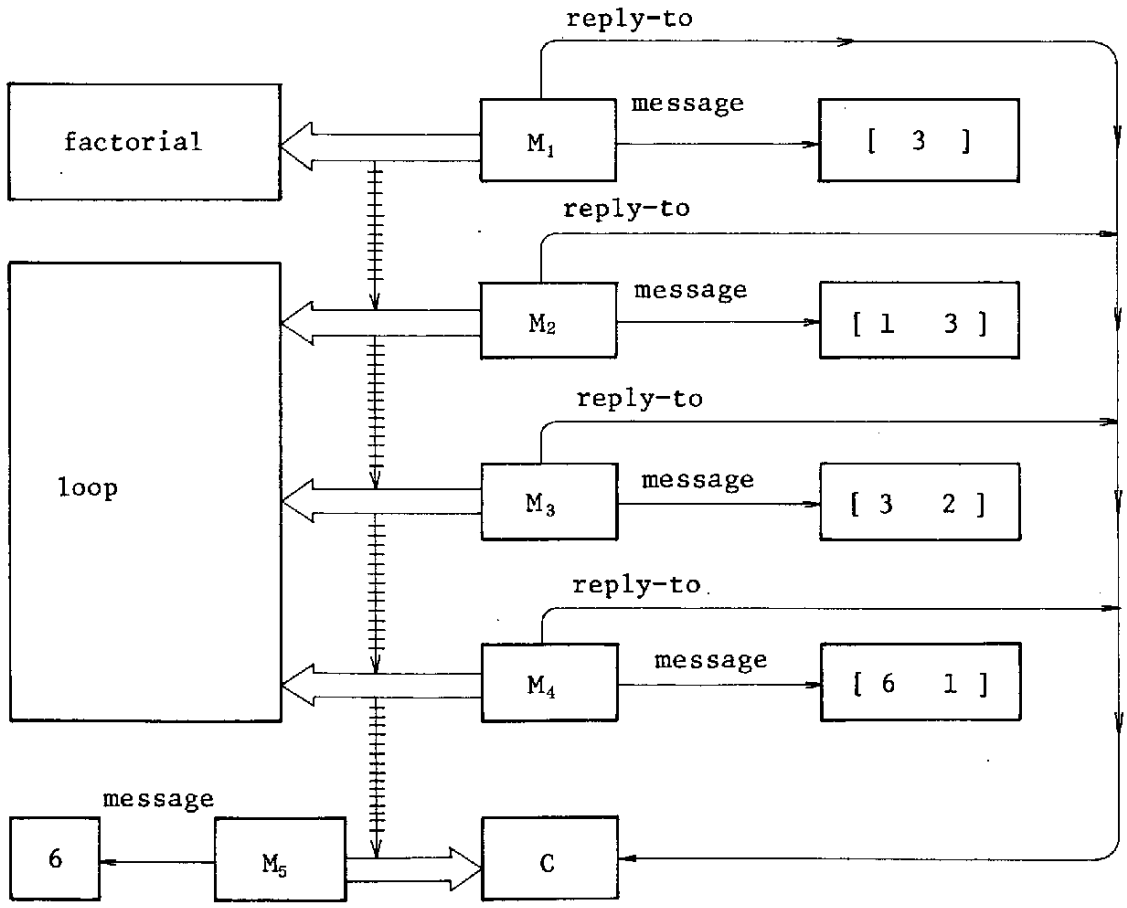


図２－５　階乗計算のイベント図

factorial , loop , C は actor である。M_1～M_5はメッセージである。⇨ は，メッセージの actorへの到着を表し，イベントと呼ばれる。⇢ は，イベント間の順序関係を示す。reply - to は，そのメッセージが到着先 actorに対して指示する応答先 actor が何であるかを示す。message は，そのメッセージの内容が何であるかを示す。

図２－５のようにして，先ずメッセージ３が factorialに送られると，factorialはaccumulationを１，countを３にしてペアを作り loopへ送る。すると loopは，countを accumtlation に掛けたものと countを１下げたものをペアにして再び loopへ送る。 最終的に，count の値が１になると，そのときの accumulationの値６が actor C に送られ，階乗の計算が終了する。

ここでは，繰り返しによる階乗の計算例を示したが，再帰的な階乗計算も actorモデル により記述できる。Actorモデルは， 極めて強力な概念的枠組を提供するものである。

(4) Iota

この言語の駆動原理は，Adaと同様に文の逐次実行である。構文的な違いはあるが，制御構造記述の概念的枠組は Adaと同系統のものである。Iotaの存在意義は，駆動原理にある訳ではなく抽象化のレベルに従った階層的なプログラムの記述とその検証ということにあるので，ここではこれ以上述べない。

(5) PROLOG

PROLOGの駆動原理は，統一化(unification)と呼ばれる一種のパターン・マッチングである。「統一化」は，一階述語論理[*1] の機械的定理証明の能率よいアルゴリズムとしてロビンソン（J. A. Robinson）によって提唱された導出原理（resolution principle)の中で用いられたものである。ロビンソンの導出原理は，それ以前のエルブラン（J. Herbran）の定理に基づく機械的定理証明の能率を飛躍的に向上した。

以下に厳密な議論は抜きにして，直観的なイメージを得るための説明を行う。まず，エルブランの定理とは，極く大雑把に言うと，『∀やヨを含む論理式は適当な変形によってヨのない形に同値変形できる。さらに，先頭の∀による量限定を除いた論理式本体は，$C_1 \wedge C_2 \wedge \cdots \wedge C_n$（ここで，各$C_i$[*2] は∨のみを含む式）の形に変形できる。そこで，もとの論理式が充分不能[*3] であるための必要十分条件は，いくつかの節C_{i1}，…，C_{im}の変数に適当に具体的な値を代入したとき$C_{i1} \wedge C_{i2} \wedge \cdots \wedge C_{im}$が充足不能になることである』と述べられる。この内容から想像されるように，エルブランの定理に基づく機械証明は，具体的な値の代入を多数回行わなければならないので能率が悪い。ロビンソンは，エルブランの定理での“具体的な値をいろいろ代入してみる”

＊１　限量記号∀（すべての），ヨ（ある）を含む論理式を扱う論理。変数，関数，述語記号も含まれる。それに対して命題論理は，命題定数だけを扱う論理で変数などを含まない。

＊２　これらのCi を節（ clause ）と言う。

＊３　直観的には「充足不能」とは，論理式中の変数にどんな具体的な値を代入しても真にならないと考えればよい。

操作を次のような論理式上のパターン・マッチングの手続きに置きかえた。これは統一アルゴリズム（unification algorithm）と呼ばれる。統一アルゴリズムを簡単な例によって次に説明する。

いくつかのパラメータを持つ命題，例えばP（x，y）があり，各パラメータに種々の値（関数などを含んでいてもよい）を代入して（このとき，前に否定～をつけてもよい）∨で結んだもの例えば，P（x，f（g（y））)∨～P（x，g（z））∨P（x，w）∨～P（x，f（w））があったとする。各命題を文字列と見て比較し，すべての命題に共通の文字が現れない位置を見分ける。その位置から始まる項をあつめる。上記の例では{f（g（y）），g（z），w，f（w）}。そこで，変数と項の組＜w，g（z）＞をとり，wをg（z）で置きかえると次が得られる。

P（x，f（g（y））∨～P（x，g（z））∨P（x，g（z））∨～P（x，f（g（z））

真中の2つの命題は～A∨Aの形をしているから除くと，

P（x，f（g（y））∨～P（x，f（g（z）））[*4]

となる。さらに，前と同様な手続きをふむと両者が不一致となる項の組＜y，z＞が得られる。yをzで置き換えれば～A∨Aの形となり，空節が導びかれる。

ロビンソンの導出原理とは，前述した節形式の命題$C_1 \wedge C_3 \wedge \cdots \wedge C_n$があったとき，それが充足不能であるための必要十分条件は，2つの節Ci，Cjに対してCi∨Cjを作った後，上の統一化アルゴリズムを適用してCiに含まれるP（…）形式の命題と，Cjに含まれる～P（…）形式の命題を消去して行くことを繰返して，最終的にCi，…，Cnから空節が導びかれることである ― というものである。

例として，p（x）→q（x）とp（a）からq（a）を導く過程を考えてみる。p（x）→q（x）は，～（p（x）∧～g（x））と同値，即ち～p（x）∨q（x）と同値であるから，次の3つの節が充足不能であることが示されればよい。結論を否定した～q（a）を加えて矛盾を導く背理法である。

（C1）～p（x）∨q（x）

（C2）　p（a）

（C3）～q（a）

まず，（C1）と（C3）で置換＜x，a＞により統一化を行い，～P(a)∨q(a)∨～q（a）から～p（a）を得る。次に～p（a）と（C2）から空節を得る。

さて，PROLOGであるが，PROLOGは先のロビンソンの導出原理を，特別な形をした節（ホーン節）に限定して適用し，その分解式を得る過程を，手続き呼び出しと解釈したものである。ホーン節とは，∨で結ばれている節の中の命題に，1つを除いてすべて～（否定）がついているものを言う。ホーン節 ～p（x）∨～q（x）∨r（x）は，～（p（x）∧q（x）∧～r（x））と同値であり，従ってp（x）∧q（x）→r（x）と同値であるから，ホーン節は→の右が唯一つの命題から成る式と言ってもよい。そのために，手続き的解釈ができるのである。

再度，例として，p（x）→q（x）とp（a）からq（a）を得る過程を考える。動作が背理

＊4　このように～A∨Aの形のものを除去した式を，分解式という。

法に基づく逆向きの推論過程と考えられることから，PROLOGでは矢印を左向き（←）に書く。

（C 1）　q（x）← p（x）

（C 2）　p（a）←

（C 3）　←q（a）

ここで，p（a）←はp（a）が常に成立することを表す。←q（a）は，q（a）から矛盾が導びかれること，即ち～q（a）が成立つことを示す。前にロビンソンの導出原理を用いて，～p（x）∨q（x）とq（a）から～p（a）を求めた過程は，PROLOGでは次のようになる。←の左になにもないホーン節[*5]からはじめ，その右にあるq（a）と統一化可能な左辺をもつ節をさがす。（C 1）がそれである。次に，統一化のための置換＜x，a＞を行って，（C 3）の右辺のq（a）を置換後の（C 1）の左辺を介して（C 1）の右辺に置き換える。すると，

←p（a）

を得る。さらに同じことを←p（a）と（C 2）の間で行えば，←の右辺もなくなり，←だけが残る。これでPROLOGの動作が終了する。しかし，これでは何も結果が得られない。計算結果を得るには，途中に適当に副作用として結果を取り出すための命題（関数と考えてよい）を入れておくのである。この事情は，LISPが基本的に2進木を計算するもので，2進木計算の過程から必要な結果を副作用として取り出すのに似ている。PROLOGのプログラムは基本的にホーン節のあつまりから成る論理式の充足不能性を証明している訳であるが，必要な結果はその証明過程の中から副作用として取り出すのである。例えば，上の例で（C 3）を次のものとする。

（C 3）←q（a），ans（x）

ここで，ansは特別な関数で，それが評価されたときにその時点の引数の値を印刷するものとする。計算が終了したときには，

← ans（a）

が残る（xは途中でaに置換されている）。その結果aが印刷され，（C 1）でのxが何に置換されたかを知ることができる。

PROLOG のプログラムと動作過程の例を図2－6及び図2－7に示す。これは，定理証明の例によく用いられるモンキー・バナナ問題を簡略化したものである。

(6)　まとめ

以上，Ada, LISP, Small Talk, PROLOG の駆動原理について触れて来たが，FGCS の目標である高度知識情報処理を念頭に置く限り，その核言語の動作そのものが人間の思考のパターンに近いことが望ましい。三段論法による演繹的推論が基本的な思考パターンの1つであることを考えれば，原理的にはPROLOGが最も知識情報処理に向いていると言える。ただし，統一化という

＊5　このような節をゴール節という。

r（x，y）　←a（x），cl（x，y）　①

cl（x，z）　←O（x，y），u（y，z）　②

u（y，z）　←mv（x，y，z）　③

a（m）　←　④

mv（m，c，b）　←　⑤

O（m，c）　←　⑥

←r（m，b）　⑦

ここで，各記号の意味は次の通り。

m：猿（monky）

b：バナナ（banana）

c：椅子（chair）

r（x，y）：xはyに手がとどく。

a（x）　：xは手先が器用である。

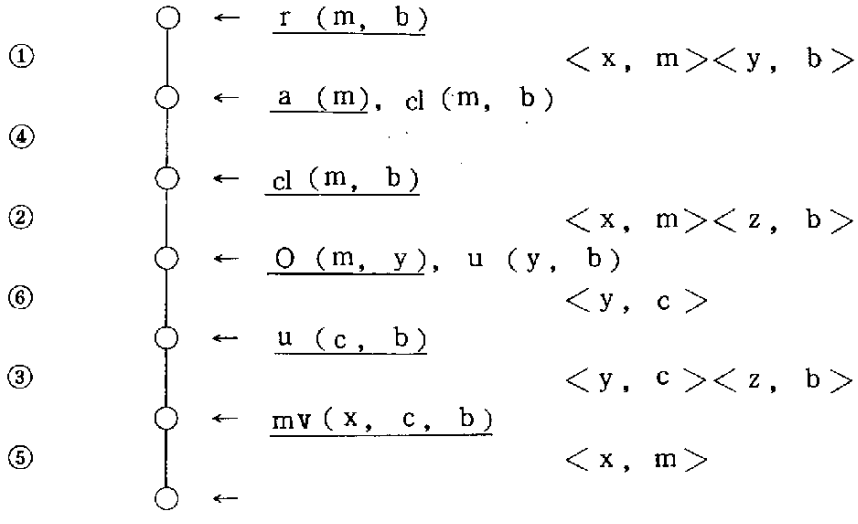
cl（x，y）：xはyの近くにいる。

O（x，y）：xはyの上に乗れる。

u（y，z）：yはzの下にある。

mv（x，y，z）：xはyをzの下に動かせる。

図2－6　モンキー・バナナ問題



ここで，左側のⓝは統一化可能なものとして選ばれた節を示す図2－6の節番号である。また，右側の＜　＞は統一化のために行われた置換を示す。

――――（アンダライン）は，統一化の対象となった命題を示す。

図2－7　モンキー・バナナ問題の動作過程

複雑な過程を経て次に実行すべき手続きを選ぶので，効率の問題を何らかの方法で解決する必要がある。

また，PROLOGにはSmall Talkのような actor 型のメッセージ授受による計算モデルを記述する機能がない。KIPS が置かれる多様な現実的環境を考えると，actor モデルの枠組の広さも核言語にあって欲しい性質であろう。

2.5 モジュラー・プログラミングのし易さ

(1) Ada

データ型の定義とそれに対する操作をまとめるカプセル化機能として，パッケージ（Package）がある。パッケージは，その仕様の記述を含むパッケージ定義（Package specification）と，その具体的実現を記述するパッケージ本体（Package body）に分かれる。また，パッケージ定義の中にはデータの型宣言が含まれるが，その具体的な形式を外部に見せないための情報隠蔽の機能として，Private typeがある。図2－8にLISPに現れるリスト処理機能を，Ada のパッケージとして書いた場合のパッケージ定義を示す。図2－9は，それに対応するパッケージ本体の記述である。

パッケージの内部には再び他のパッケージを定義してもよい。すなわち，パッケージはネストすることができる。この機能を用いて，プログラムを階層的にモジュール化することができる。しかし，同時にAdaではALGOL風のブロック構造も許しており，ブロック構造とモジュール構造（パッケージ）が自由に入れ子になることができるため，このことがプログラム構造を複雑にしてしまうのではないかという批判がある。さらに，一般にブロック構造とモジュール構造はなじみにくいのであるから，ブロック構造を廃止するあるいはモジュール内での局所的使用だけに制限するという提案もある。

このように，Adaにはモジュラー・プログラミング用機能は入っているが，機能が豊富すぎて概念的整理が不十分であると見ることができる。

(2) Iota

タイプ（type）モジュール，サイプ（Sype）モジュール，プロシージャ（Procedure）モジュールの3種類のモジュール概念をもつ。各々は，インタフェース（interface），仕様（specification），実現（realization）の3つにプログラム・テキスト上で明確に分離して書かれる。

タイプ・モジュールは，抽象データ型を定めるものである。そのインタフェース部には，そのデータ型を参照するための関数と，加えることのできる操作が，入出力の形式と共に定義される。

```
Package LISP-PROCESSING is Type LISP is private ;
  NULLIST : constant LISP ;
  function CAR (X : LISP) return LISP ;
  function CDR (X : LISP) return LISP ;
  function CONS (X, Y : LISP) return LISP ;
       ∫   ( 中略 )
private
  type LISP is access
  record
    CHOICE : constant (LIST, ATOM) ; = LIST ;
    case CHOICE of
      when LIST ⇒  CAR : LISP : = null ;
                   CDR : LISP : = null ;
      when ATOM ⇒  VALUE : INTEGER :
    end case ;
   end record ;
   NULLIST : constant LISP : = null ;
end LISP-PROCESSING
```

図２－８　リスト処理パッケージの定義

```
package body LISP-PROCESSING is
   function CAR (X : LISP) return LISP is
     begin
       if  X. CHOICE = LIST then
           return  X. CAR ;
       else
           raise  NO-LIST ;
       end if
     end CAR ;
    ∫     ( 中略 )
     function CONS (X, Y ; LISP) return LISP is
       begin
         return  new LISP (
                          CHOICE ⇒ LIST,
                          CAR ⇒ X,
                          CDR ⇒ Y   ) ;
      end CONS ;
   end LISP-PROCESSING ;
```

図２－９　リスト処理パッケージの本体

```
interface procedure  INTSEARCH
  fn    SORTED : INTARRAY ⟶ BOOL
        LOCATE : (INTARRAY, INT) ⟶ (BOOL, NN)
  end interface
  specification procedure INTSEARCH
  var   X : INTARRAY ;  M, N : NN ;  I : INT
  axiom 1 : SORTED (X) ≡
            ∀M.∀N. (0 ≦ M∧M < N∧N < HIGH (X)
                    ⊃X [M] < X [N])
        2 : SORTED (X)⊃
            (LOCATE §1 (X, I)⊃
              (0 ≦ LOCATE §2 (X, I) ∧
               LOCATE §2 (X, I) ≦ HIGH (X) ∧
               X [LOCATE §2 (X, I)] = I))
        3 : SORTED (X)⊃
            (∿LOCATE §1 (X, I)⊃
             ∀M. (0 ≦ M ∧ M ≦ HIGH (X)
              ⊃X [M] ≠ I ))
  end specification
```

図２－１０　プロシージャ・モジュールの仕様定義

```
realization procedure  INTSEACH
  fn   ↓LOCATE (X : INTARRAY, I : INT)
                return (B : BOOL, L : NN)
       vor M, N : NN
       M : = 0 ;  N : = HIGH [X] ;
  end fn
end realization
```

図２－１１　プロシージャ・モジュールの実現部

仕様部には，各々の関数や操作が満たすべき性質が公理（axiom)として記述される。実現部には関数や操作を実現するアルゴリズムが記述される。

サイプ・モジュールは，タイプ・モジュールの性質の一部をとり出して，より抽象的・一般的

な概念として定式化できるときに，それを別のデータ型として定義するもので，見かけは，タイプ・モジュールとほぼ同じである。タイプ・モジュールでは，サイプ・モジュールを参照することによって，インタフェース部や仕様部の記述を簡素にできる。

プロシージャ・モジュールは，タイプ・モジュールで定義された抽象データ型に対して行う応用的処理を関連するものをまとめて一連の関数として系統的に記述するものである。その仕様部では，やはり公理の形で（関連がある場合にはその関連も含めて）各々の関数が充すべき性質が記述される。

図２－10と図２－11に，プロシージャ・モジュールの記述例を示す。例えば，図２－10のaxiom 1では，INTARRAY（整数の配列）型のデータが“ソートされている”ことの定義が書かれている。また，axiom2と３では，LOCATE（X，I）[*6]の仕様がSORTEDを使って書かれている。なお，LOCATE§１（X，I）は，結果の第１番目の要素（論理値）を表し，LOCATE§２（X，I）は結果の第２番目の要素（位置を表す自然数）を表す。

以上のようにIotaは，カプセル化（タイプ／サイプ・モジュールによる）と情報隠蔽（インタフェース・モジュールによる）による階層的モジュール化プログラミングを，言語全般に徹底した形で実現している。

(3) Small Talk

2.4の(3)で述べたように，Small Talk でメッセージの授受を行いながら計算をすすめる主体である対象はクラスに属するが，クラスはサブクラス（subclass)を持つことができる。すなわち，あるクラスの授受するメッセージのうちまとまりを持った一部分だけをそのメッセージとして授受する別のクラスを定義できる。クラスとサブクラスの関係は，Iotaのタイプとサイプの関係に似ている。また，図２－４及び図２－５で示したactor factarialの中で再びactorとして定義されているloopも，サブクラスと考えてよい。

以上のように，Small Talk においても，授受するメッセージ群の階層的構造とクラス／サブクラスの階層的構造を同型に対応させた形でモジュール化プログラミングが可能である。

(4) PROLOG, LISP

PROLOGのプログラムは矢印式の形に書いたホーン節を並べたものであるし，LISP のプログラムは関数の定義及び引用を書き並べたものである。勿論，PROLOGの各命題やLISP の関数の間には論理的に階層を持つことがあるが，(1)～(3)に述べた言語のようにその階層を陽にプログラム・テキスト上に表現する構文はない。従って，(1)～(3)のような言わば垂直型のモジュール化機能はないと言える。

＊6　LOCATE（X，I）の機能は，次の通り。配列Xの要素を探し，値Iを持つものがあれば，その位置を値trueと共に返す。値Iを持つものがなければ，値falceを返す。

しかし，PROLOG，LISP共に機能単位を部品として，矢印式や関数定義の形でまとまり良く記述できるので，水平型のモジュール化機能とも言うべきものを備えていると言える。

2.6 推論アルゴリズムの記述のし易さ

(1) PROLOG

推論アルゴリズムとは，次のようなことを行うものである。「A，B，Cが成立つならばDが成立つ」といった規則の集合が与えられていて，それに対して「Xは成立つか？」あるいは「X，Y，Zが成立っているが，原因は何か？」というような質問に対する回答を出す。

2.4の(5)でPROLOGが定理の機械証明の研究から生まれたものであることに触れた。その際，PROLOGの動作がA→Bという規則の矢印を逆向きにたどることに他ならないことも述べた。実際図2－6のモンキー・バナナ問題は，推論そのものである。PROLOGの中に推論アルゴリズムが組み込まれていると言ってよい。最初に述べたような型の推論を必要とする問題は，PROLOGの最も得意とするところである。

一方，コンサルテーション・システムのように，専門知識が規則の集合として外部記憶などに与えられていて，その規則の集合を一定の戦略によって探索するアルゴリズムを書く場合を考える。前述の意味の「規則」は，データとしてプログラムの外にあるから，PROLOG自体の中に推論機構が内蔵されていることとは違った次元の問題になる。しかし，この場合でも規則の探索戦略を高度なものにするには，探索状況を判断したり推測したりする必要が出てくるであろう。そのような判断・推測のアルゴリズムもPROLOGで書くことができる。

(2) LISP

人工知能研究の中から生まれた言語であり，その応用である知識工学分野（コンサルテーション・システムなどもその分野に含まれるものである）でも広く使われている。実績のある言語であるから，推論アルゴリズム記述に適していることは論を待たないであろう。

しかし，推論ということに限って見た場合に，PROLOGと較べると一段抽象度の低い言語であると言える。これは，LISP が2進木を扱うものであるから当然のことであろう。

(3) Ada, Small Talk, Iota

特に推論ということを意識して設計されたものではない。

2.7 並列処理のし易さ

(1) Small Talk

2.4の(3)で述べたように，対象（object）同志はメッセージの授受以外は全く関係なく概念的に独立である。従って，各対象が並列に動作するように（適当なハードウェアの下で）インプリメントすることは容易であると考えられる。

対象の類似概念であるactorを用いた計算の例を図2－4及び図2－5に示した。そこでは，イベントの列が一系列だけであるので，計算は逐次に進まざるを得ないが，イベントの系列がいくつかの技を持つような場合には，actorは概念的に独立なので，計算を並列に進めることができる。

Small Talkは，並列処理を受け入れる自然な枠組を持つていると言ってよい。

(2) PROLOG

PROLOGも，並列処理を受け入れる自然な枠組を持っていると言ってよい。次に実行すべき節(文)として統一化可能なものを探したとき，統一化可能なものが複数個ある場合に，そこから先の計算（推論）は並列に進めることができる。

図2－12にそのような場合の例を示す。図2－13にその動作過程を示す。

```
①  factorial (0, s(o))←
②  factorial (s(x), u)  ←  factorial (x, v), times (s(x), v, u)
③← factorial (x, s(o))
```

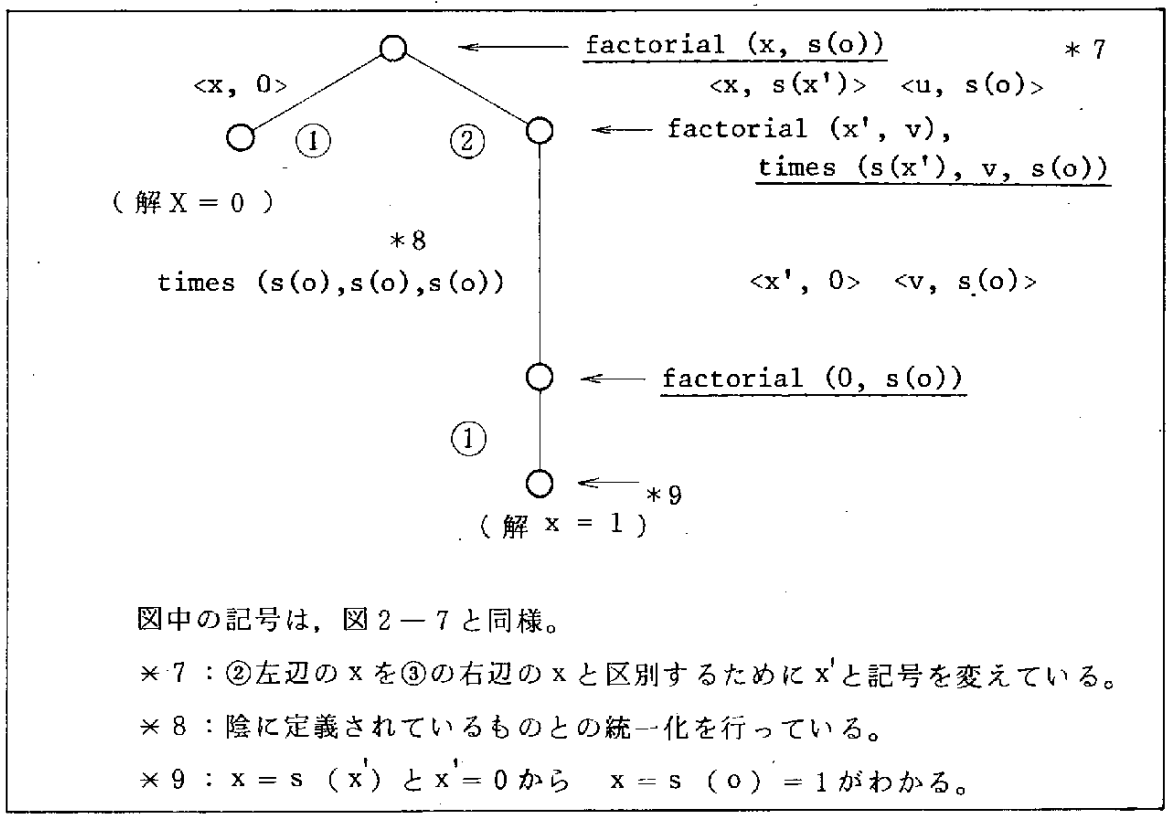
ここで，記号の意味は次の通り。

```
factorial (x, y):  x! = y
      s(x)      :  x + 1 (sはsuccessorの意味)
times (x, y, z) :  x·y = z
```

図2－12　PROLOGによる階乗の根の計算

図2－12は，x！＝1を満たすxを求めるプログラムである。ゴール節←factorial（x，s(o)）と統一化可能なものが①と②の両方であるため，図2－13の計算過程が枝分かれしている。一方の枝から解x＝0が，他方の枝から解x＝1が求まっている。この2つの過程は，並行して実行させることができる。

なお，ここで興味深いことは，図2－12のプログラムは逆向きに動かせる点である。すなわち，階乗の根ではなく，階乗を求めるプログラムとしても使える。それには，例えば2！を求めるとすると①と②はそのままにして，③を←factorial（s（s（o））， x）とすればよい。その計算過程を図2－14に示す。



図中の記号は，図２－７と同様。

＊７：②左辺のｘを③の右辺のｘと区別するためにｘ'と記号を変えている。

＊８：陰に定義されているものとの統一化を行っている。

＊９：ｘ＝ｓ（ｘ'）とｘ'＝０から　ｘ＝ｓ（ｏ）＝１がわかる。

図２－13　階乗の根の計算過程

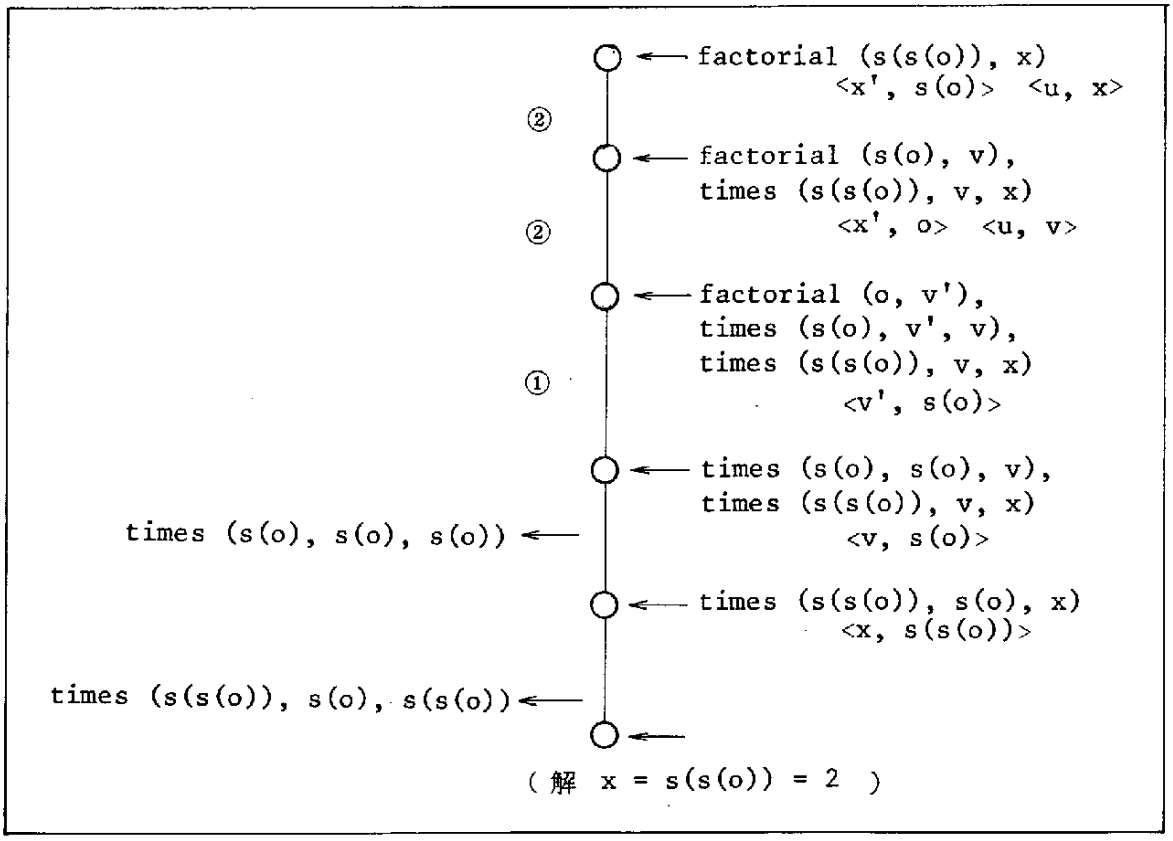


図２－１４　階乗の計算過程

(3) Ada

Adaでの並列処理は，タスクの定義・起動・呼び出しを陽にプログラム・テキストに書くことによっている。Small TalkやPROLOGでは並列処理が可能な部分を陽にプログラム・テキストに書くことはなく，並列性はプログラムの実行過程の中に内蔵されている。すなわち，(現実のSmall TalkやPROLOGには並列処理がインプリメントされている訳ではないが) 並列性を意識するのは処理系だけである。それに対して，Ada ではプログラム自身が並列性を意識する。

Adaの並列処理記述は，2.5の(1)で述べたパッケージと同じくカプセル化機能を組合せた形で行う。正確に言えば，カプセル化機能を実現するものとしてモジュール（module）があり，モジュールにはパッケージ・モジュールとタスク・モジュールがある。

図2—15にタスク・モジュールの例を示す。

```
task LINE-TO-CHAR is
  type LINE is array (1..80) of CHARACTER ;
  entry SEND-LINE (L : in LINE) ;
  entry GET-CHAR (C : out CHARACTER) ;
  end LINE-TO-CHAR ;
task body LINE-TO-CHAR is
  BUFFER : LINE ;
  begin
    loop
      accept SEND-LINE (L : in LINE) do
        BUFFER : = L ;
      end SEND-LINE ;
      for I in 1.80  loop
        accept GET-CHAR (C : out CHARACTER) do
          C : = BUFFER (I) ;
        end GET-CHAR ;
      end loop ;
  end LINE-TO-CHAR ;
```

図2—15　タスク・モジュールの例

これは，1行分のデータ領域（BUFFER）とそれに1行を送り込む操作（SEND-LINE）及びそこから1文字を取り出す操作（GET-CHAR）をカプセル化したタスク（LINE-TO-CHAR）の定義と本体である。LINE-TO-CHARタスクがinitiate文で起動されると，1行分の領域BUFFERがタスクに対応して確保される。他のタスクからSEND-LINE要求があると，送られて来たデータをBUFFER に入れてその要求の処理が終り，2つのタスクは関連を断つ。次に GET-CHAR要求があるとBUFFERから1文字取り出して要求元のタスクに返す。GET-CHAR要求を80回処理したら，始めに戻る。すなわち，要求は1回の SEND-LINEとそれに続く80回の GET-CHAR 要求という順序でしか処理されない。accept GET-CHAR に制御が達すると，他のタスクから GET-CHAR 要求があるまで待ちことによってタスク間の同期をとるのである。これを，acept と他タスクからの要求のランデブーという。

LINE-TO-CHAR タスクと他のタスクは，accept SEND-LINE又は accept GET-CHARでランデブーが起ってから，end SEND-LINE又は end GET-CHAR までの間だけ関係をもち，それ以外は並列に動作する。

2.8 データベース・アクセスのし易さ

(1) PROLOG

関係データベースの「関係（relation)」は,PROLOGのassertion（今までの記法で言えば，←の右側が何もない節，即ち常に成立つ命題を表す節）をいくつか並べることによって表現できる。関係という概念は，直積の部分集合のことである。また，assertion はその引数の間に，命題の名前で意味される関係が成立っていることを示す。つまり，1つのassertion は各々の引数が属す領域の直積の中の点を表していることになる。従って，assertion をいくつか並べると，直積の部分集合を指定したことになり，それによって関係が定義されるのである。

また，関係データベースのアクセスに用いられる集合論的な操作，和集合（union),差集合(difference),積集合（intersection）なども，集合演算と論理演算（PROLOGの文は実は論理式である）の対応によって非常に整合性よく記述される。

以上の事実に注目して，PROLOGを関係データベースの検索言語として用いる研究もなされている。

(2) Ada, LISP, Small Talk, Iota

いずれも，PROLOG のような意味での言語が本来持っているデータベースとの整合性の良さはない。ただし，LISP では長く実用に使われている言語であることから，処理系によっては高度な入出力機能を備えているものがある（例えばINTER-LISPなど)。それを用いてデータベース・アクセスをすることは可能であろう。

2.9 分散処理のし易さ

(1) Small Talk

ここでは，Small Talkというよりもその概念的枠組を考えるので，actorモデルについて考える。

分散処理の重要な性質は，各々の構成要素がシステム全体についての情報を持てず，自分の周辺の局所的情報だけに基づいて動作しなければならないという点にある。

Actor モデルは，上記の分散処理の基本的性質を内蔵している。「局所性」は，時間的前後関係に関する局所性と，情報の流れに関する局所性に分けて考えられるが，actor モデルで言えば，前者はイベントの局所性に対応し，後者はメッセージ授受の局所性に対応する。

イベントはactor へのメッセージの到着のことであるが，イベントの発生順序は一つの actor に注目した場合にだけ意味をもつ。異なる actorに関するイベント同志の順序関係には何の意味もない。actorは，お互いに独立して動作するので，違う actorにメッセージが同時に到着したかどうかというような情報は概念上必要ない。以上の意味で，イベントの時間的前後関係は局所的な概念である。

Actor 間の情報の伝達はメッセージを介してのみ行う。さらに，ある時点（勿論この「時点」は，前述のイベントの局所性によって，その actorだけに意味のある局所的時間軸の上での「時点」である）でメッセージを授受できる actorは，各actorごとに定まっている。それは局所的時間によって変化することはあるが，ある時点をとってみれば１つのactor は一定の有限個の actor としか情報を授受できない。その意味で，メッセージの授受は局所的な概念である。

以上のように，actor の上を流れる時間は局所時間であり，空間的にも actor は自分の周辺の情報しか持ち得ない。この事実に基づいて，分散処理システムの各構成要素を actorと考えれば，actor モデルによって分散処理が自然にモデル化されることは明らかであろう。

(2) Ada , LISP , Iota , PROLOG

いずれも，Small Talk（actorモデル）のような意味での分散処理との整合性の良さはない。ただし，Adaについては，2.7の(3)で述べたタスク・モジュールを使って分散処理の記述をすることができる。

2.10 検証のし易さ

(1) Iota

設計意図そのものが，プログラムを抽象化のレベルに従って階層的に記述し検証することにあるため，検証のための機能は非常に充実している。

図２－10にも見られるように，タイプ/サイプ・モジュール及びプロシージャ・モジュールのいずれについても，その仕様（specification）部では公理（axiom）として，各々の関数が満たすべき性質のすべてが記述される。公理は，Iota論理と呼ばれる論理の論理式で記述される。Iota論現は，多ソート一階述語論理（mang-sorted first order logic）の一種である。ここで,「ソート（sort)」とは大雑把にはプログラミング言語の「抽象データ型」に対応する論理上の概念と思えばよい。従って，通常の論理がデータ型の概念を持たないのに対して，多ソート一階述語論理はデータ型をも取扱える論理であると言える。

Iotaでは，階層的なプログラム開発（段階的詳細化）と検証を次のような枠組でとらえる。先ず，公理による仕様記述はIota論理の「理論（theory)」を定める。ここで，「理論」とは大雑把にソート（つまり，抽象データ型あるいはモジュール）の集合と関数の集合を組にしたものと考えてよい（厳密にはこれですべてではないが)。また,階層的プログラム開発の過程，すなわち抽象化のレベルを下げて行く過程は，「理論」の「拡大」と考える。理論Ｂが理論Ａの拡大であるとは，理論Ａのソートと関数の集合が，各々理論Ｂのソートと関数の集合に含まれることをいう。

一方，「実現（realization)」部についても，それを（Iota論理をその中に含む）多ソート論理の「理論」と考える。さらに，仕様部に対応するIota論理の「理論」から，実現部に対応する「理論」へのある準同型を定義する。この準同型は，仕様の中の公理が成立つことと，その公理の準同型による変換先が実現の「理論」で証明可能であることとが同値になるように定める。

以上の準備の下に，実現の「理論」で先の準同型による変換先の正当性の証明を，ホア論理の規則とIota 論理の規則を用いて自然演繹法と類似の技法で行う。（実際には,この過程はもっと複雑で，もとの公理の変形や「理論」の拡大などの操作が入るが，省略する）

(2) Ada, LISP, Small Talk, PROLOG

いずれも，Iotaのような意味での検証を意識して設計されてはいない。

PROLOGについては，プログラム自体が一種の論理式なので検証はやり易いとも思えるが，実行過程の戦略にバリエーションがあり，その戦略がプログラムの動作と無関係でないためか，論理プログラミングの検証はあまり研究されていないようである。

また，Small Talkに関連したactorモデルについては，それに基づく並列プログラムの形式的仕様及び検証技法が研究されている。

2.11 結論

PROLOGは，FGCSが目標としているKIPS向きの良い性質を備えている。すなわち，推論機構を内蔵しており，関係データベースや並列処理との整合性も良い。

しかし，FGCSの核言語として用いるには,現状のPROLOGでは不十分である。すなわち，抽象データ型に基づく階層的プログラム開発と検証の機能，actor モデルに基づく分散処理記述機能を

追加する必要がある。これらは，ソフトウエア危機やFGCSが置かれるであろう多様な環境（例えば分散型知識ベース・システム）を考えたとき，是非とも必要な機能であると考えられる。

なお，まとめとして表２－１に比較対象として取り上げた言語の比較表を示す。

表２－１　プログラミング言語の比較表

項目 / 言語	駆動原理	モジュラープログラミング	推論	並列	検証	分散	データベース
Ada	文の逐次実行	△	—	△	—	△	—
LISP	関数の逐次評価	△	○	—	—	—	△
Small Talk	メッセージ授受	○	—	○	—	○	—
Iota	文の逐次実行	○	—	—	○	—	—
PROLOG	統一化	△	○	○	—	—	○

〔記号の意味〕　○：適合性あり

△：部分的に適合性あり

—：核当しない（設計意図にない）

参 考 文 献

[1] Proceedings of International Conference of FGCS (Oct. 1981).

[2] Wegner, P. : Programming with Ada : an introduction by means of graduated examples, Prentice-Hall (1980).

[3] 中西 正和 : LISP 入門 ― システムとプログラミング―, 近代科学社(1977).

[4] Ingalls, D.H.H. : The Small Talk-76 Programming System Design and Implementation, in conference record of the Fifth Annual ACM symposium on POPL.

[5] Kay, A.C. : Microelectronics and the Personal Computer ; Scientific American, Vol. 237, No.3 (Sep. 1977).

[6] 米沢 明憲 : ACTOR 理論について, 情報処理, Vol. 20, No.7, (1979).

[7] Hewitt, C. : Viewing Control Structures as Patterns of Passing Messages, MIT AI Lab., AI Memo 410 (Dec. 1976).

[8] 佐藤 泰介：導出原理による定理証明, 情報処理, Vol. 22, No.11, (1981).

[9] Loveland, D.W. : Automated Theorem Proving : A Logical Basis, North-Holland (1978).

[10] Kowalski, R. : Predicate Logic as Programming Language, Information Processing 74, North-Holland (1974).

[11] Wright, D.J. : Prolog as a Relationally Complete Database Querry Language which can Handle Least Fixed Point Operators, Tech. Report No.73-80, Dep. of Comp. Sci. Univ. of Kentucky (Jun. 1980)

[12] Nakajima, R., et.al. : Hierarchical Program Specification and Verification - a Many-sorted Logical Approach, Acta Informatica 14, 135-155 (1980).

[13] 佐渡 一広, 米沢 明憲：抽象データ型言語, 情報処理, Vol. 22, No.6 (1981).

[14] 中島 玲二: Ada「批判」, 情報処理, Vol. 22, No.2 (1981).

[15] 中島 玲二: プログラム検証入門(1)～ (完), bit, Vol. 12, No.11～14 (1980).

(日立製作所 青山 明夫)

3. 知識ベース・システムの構築と展開

3. 知識ベース・システムの構築と展開

3.1 はじめに

種々の知識ベース・システムが，ここ数年来開発されている。これらを対象とする分野としては，応用人工知能というときと，知識工学と呼ぶこともあるが，いずれにせよ，高度の専門的知識を使った問題解決システムの作成を意味している。

本稿の目的は，知識ベース・システムの構築と展開についての進め方を述べたものである。

専門的知識というのは，永い経験により得られた特定領域の専門家の知識を言う。このような知識を有しないシステムは，知識ベース・システムとならないことに留意する必要がある。

というのは，知識ベース・システムの開発方法は，従来のアルゴリズムを主体とするシステム設計とはかなり異なったアプローチを必要とする。そこで，知識ベース・システム作成のための方略を整理しておくのが，ここでのねらいである。

もちろん，そのためには知識ベース・システムの開発経験を十分に経ていなければならない。残念ながら，日本の場合には，知識ベースの枠組の導入にしても，日が浅く，それほど十分な経験を積んでいるというわけでない。

3.2は，筆者の限られた範囲の，ここ数年間の経験をもとにしての，知識ベース・システムの進め方についての考察である。

ここでの視点は，次の3点にある。

① 知識ベース・システム構築の特徴

② 知識の収集方法

③ 知識ベース・システムのための道具だて

次に，これらの視点をもとに，従来の知識ベース・システムがどのようにして開発されてきたかをみようというのが，3.3の狙いである。

ついで，3.2および3.3の考察から，3.4では，どのような領域を知識ベース・システムとしていくかを，過去の研究側から考察している。

3.2 知識ベース・システム構築の進め方

(1) 知識ベース・システム構築の特徴

知識ベース・システムの基礎となる知識は，専門家のものを前提としている。このことは，知識ベースで使われる知識は，常に改良していかなければならない性格を有している。つまり，知識ベースは"開放的(Open)"な形で構築されていなければならない。

知識ベース・システムの開放性は，次の2つの意味を持っている。第1は，知識は各々が独立した単位構造となっていて，単位間には，それほど相互作用がない。例えば知識がルールの形式で

記述されているときは，独立したルールとして知識ベースにストアされる。しかがって，知識はルールの修正，追加および削除に対して開放的である。その結果，知識はルールとして，加法的に増加していく。ただし，性能は必ずしも，加法的に向上していくとは限っていない。

第2は，知識そのものが動的な性質を持つものと考えるもので，知識の単位そのものが可変的構造としておくことである。それは，知識がその他の知識と相互作用することにより，新しい知識を生成したり，また，構造的に組み変わっていくということを意味している。こうした知識の構造そのものの開放性もあるわけである。

特に，人間の知識の場合には，第2の特徴を有し，したがって，専門家となるためには永い経験を必要とするという次第である。

これらの第1，第2の開放性は，いずれにせよステップ的な精密化（Stepwise refinement）を知識ベースに加えていくことに対応している。つまり，知識ベース・システムは短期間のうちに完成されたシステムとして構築できないという特徴を持っている。

逆にみると，次のことが言える。

① 第1次の近似システムの知識ベース・システムは，ステップ的な精密化を受けるという前提で，できるだけ短期間のうちに完成すること。

② 知識ベース・システムの基礎となる知識は，変化するという開放的な性格を有しているのでできるだけ簡単な表現形成にすること。

③ 知識ベース・システムに対する知識の増加が，必ずしも，システムの性能アップにつながらないこと。

これらの特徴は，システム構築のアプローチにも結びついている。

すなわち，方策としては，

①に対しては，できるだけ早く知識ベース・システムを動くシステムとして開発せよ，そして最初から，高性能をねらうべきではない

②に対しては，できるだけ読みやすい知識の表現形式を使うこと，そして，どんどん作り変えていくことを心がける

③に対しては，専門家といっても種々のレベルがあること。もし，知識を増しても，システムの性能が変らないときは，その専門家のレベルを考える

と，それぞれが考えることができる。したがって，知識ベース・システムの構築のためには，ステップ的な精密化と，高度の，質の良い知識の収集に時間をかける必要がある。また，その点での経験が重要である。

(2) 専門的知識の役割

知識ベース・システムの基礎は，専門的知識にある。このことは，ある特定の分野の専門家がいると，高度のシステムを開発することができるのかというと，必ずしもそうは言えない。

専門家として，トップレベルにあり，かつ知識ベース・システムの作成に対して，熱心であることが必要である。

ところが，トップレベルの専門家は，常に，時間的に忙しい人達が多く，したがって，知識の収集および精密化がむずかしいと予想される。結果的に，システム設計者の周辺にいる専門家との共同作業が主となる。このことは，平均的レベルの専門家からは，平均的な知識ベース・システムしか開発できないと言える。

知識ベース・システムのユーザ側からみても，必要なのは，トップレベルの専門家が作成したシステムであり，また，それだからこそ，エキスパート・システムと呼ばれるわけである。

こうした例は，INTERNISTにおけるPittsburgh大学医学部の名医であるMyers, J. Dの役割に代表されている。それでは，トップレベルの専門家がいないと知識ベース・システムは作成できないのだろうか。そうでもないことは，いくつかの開発例からも分る。

それでは，どのように専門家と，知識ベースを開発していけばよいのだろうか，そのヒントが緑内障診断システムであるCASNETの開発方法にある。

CASNETの知識ベースは，ONET（眼科用ネットワークで，SUMEX-AIM上で作成された）により，修正された。ONETにより，Washington大学，John Hopkins大学などの5つの大学の専門家がCASNETのプロジェクトに参加し，システムの性能をチエックした。

つまり，複数の専門家の知識をうまく合成し，また，異なった意見を反映して，知識ベース・システムを精密化していくというアプローチと考えられるわけである。医療の知識ベース・システムの場合には，専門家ごとに違った事例を入力し，システムの性能を評価することがもっとも重要である。その意味で，そうした専門家との共同システムは重要である。

以上をまとめてみると，専門家との共同開発方法は，次の2点となる。

① 特定の分野の，トップレベルの専門家を探し，熱心な共同開発を進めていくこと。

② 特定分野の，複数の専門家間の意見の合意により共同開発を進めていくこと。

いずれにせよ①も②もむずかしいが，この専門家との共同開発が知識ベース・システム構築上のポイントである。

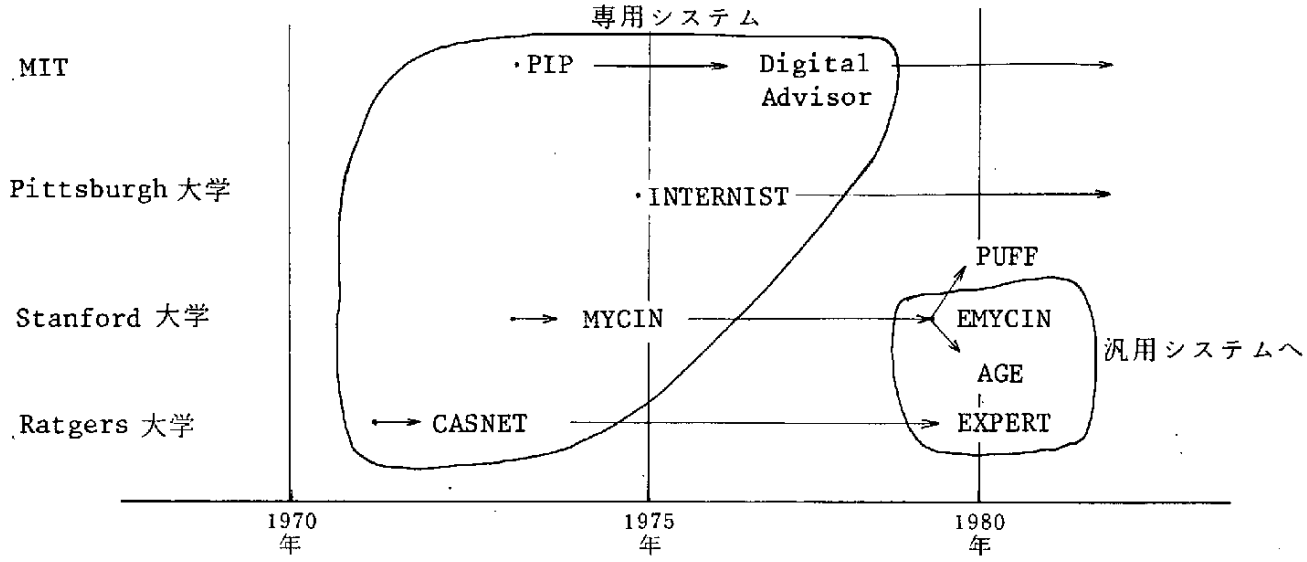
(3) 知識ベース・システムのための道具だて

知識ベース・システムを構築するためには，すぐれた専門家との共同作業という点も重要であるが，すぐに使える道具が必要である。

短期間のうちに知識ベース・システムを完成させ，システムがどの程度の性能を持つかを専門家に提示して，ステップ的な精密化を行うためにも，システム作成のための道具だては開発活動の前提となる。

こうした道具だては，どのようにして開発されてきたかをみると，通常は，固有システムから汎用システムへという発展過程を経ている。

例えば，米国における医療診断用の知識ベース・システムの開発の流れを示した図３－１をみても明らかである。感染症のコンサルテーション・システムMYCINが，EMYCINに，緑内障の診断システムCASNETがEXPERTへ，それぞれ特定システムから汎用システムに進展している。



（各々のシステムの特徴は表３－１に示されている）

図３－１　医療における知識ベース・システムの歴史的背景

そして，それぞれの特定システムの開発年月が，だいたい６年以上を経て，汎用システム化されているのも特徴である。

現段階で，第５世代コンピュータの適用を考えると，以上のような経験を繰り返すことは，時間的にも，また，コスト的にもムダと言えよう。

幸いにして，EMYCIN，EXPERTともに，コンセプト的にはプロダクション・システムの枠組で作成された道具群である。

したがって，知識ベース・システムの道具だてを作成するとすれば，各種のプロダクション・システムを開発する必要がある。

例えば，EMYCINは結果部駆動型[注)]のプロダクション・システムであり，問題の目標から逆向きにルールを適用する後戻りの制御構造を持っている。これに対し，EXPERT　は前提部駆動型のプロダクション・システムであり，データを集めて最終目標に到達していくようなルール適用で前向きの制御構造を持っている。

プロダクション・システムの使いやすさは，EXPERTタイプの前向きの制御構造のシステムの方が，MYCINタイプよりも優れている。しかし，これらのプロダクション・システムは適用領域の知識表現形式によって違ってくるので，どちらが優れたプロダクション・システムであると断定することはできない。

ただし，データ収集のしやすい対象には，前向き型のプロダクションが，また，最終目標から

注）"IF A, then B" をプロダクション・ルールとすると，Aの部分を前提部，Bの部分を結果部と呼ぶ。

みて必要なデータを収集していくには後戻り型のプロダクションが適している。

もし，知識ベース・システムを，ルールベース・システムとして位置づけていくとしたときには，その時の道具だてとしては，プロダクション・システムを用いていく方向となる。そして，実際に，動くシステムとして知識ベース・システムを作成するためにも，プロダクション・システムが第1の道具だてと考えられる。

もちろん，知識ベース・システムの作成のためには，フレームで代表される知識表現システムを道具だてと考えることもできる。プロダクション・システムに対し，フレームに第2の道具だてと言うこともできる。このことは，フレームを用いた多くの知識ベース・システムの開発状況からも明らかである。

ただし，現実的な問題を扱うフレーム型の知識ベース・システムは，それほど完成していない。やはり，知識をルール形式で表現していく方が，動く知識ベース・システムを構築していくためには強力な知識表現の枠組と言えよう。

したがって，知識ベース・システムの道具だてとしては，次のことが言える。

"道具だてとしては，プロダクション・システムに注目せよ。そして，動く知識ベース・システムを構築せよ。"

3.3　知識ベース・システムの開発過程

実際の知識ベース・システムの開発には，道具だての作成が必要である。このことは，3.2の終りで述べた通りである。

具体的には，記号処理言語のLISPを選んで，プロダクション・システムを作成していくことが道具だてを準備するひとつの方向である。ところで，こうしたシステムのプログラム技術はそれほど確立されているわけではない。

現段階では，知識ベース・システムの道具だてそのものの開発方法に新しい考え方を導入する必要がある。ひとつの考え方としては，プログラム・コードの情報ドキュメンテーション・システムそのものを開発する必要がある。

通常，プロダクション・システムを作成するプログラム言語としては，LISP が選らばれる。そして，LISPには，構造エディタのように，コーディング環境が整備され，コードの修正変更の履歴が保持されている。

したがって，道具だてを構成する要素を，コード的に変更したときには，その変更が，他のユーザにも伝達できるようなドキュメンテーション・システムが必要である。

このことは，一度作成したコードは大切に扱って，開発のダブリをさけられるように，コードの説明体系を確立することを意味している。

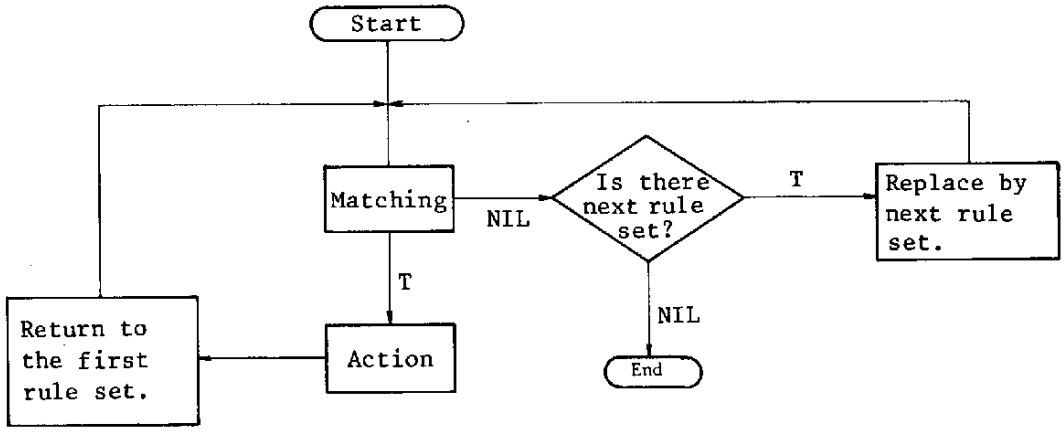
従来の，システム開発の多くは，設計と実装までには多くの時間とコストがかけられていたがシステムの文書化，プログラムのメインテナンスに対しては，それほど努力がなされていない。

プログラムの生産性を高めていくことは，過去の，有効なプログラム・コードを，いかに蓄積し，また，それらを使って，新しいシステムを作成していくような方法論が必要である。

そのひとつが，モジュラー・プログラミングであるが，その前提として，どのようなプログラム・モジュールが準備されているかが重要である。

プロダクション・システムを例にとると，どのような制御構造を，モジュラー・プログラムとして作成していくかということになる。こうしたプロダクション・システムの制御構造を示したのが図３－２である。この制御構造は，プロダクション・システムとしてみると，もっとも簡単な型式である。もちろん，この基本構造をひとつのモジュラーの単位として，さらに通常は改良されていく。

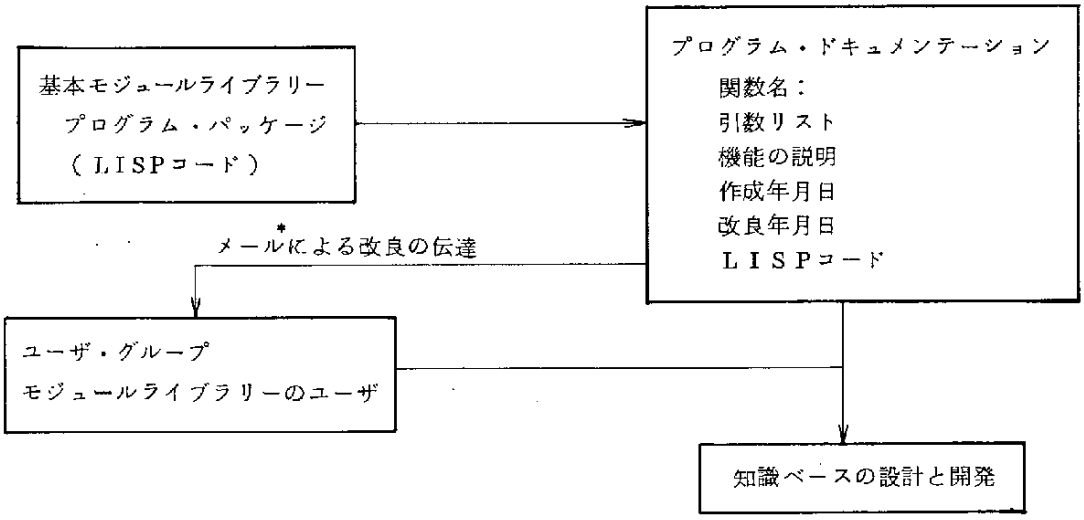
もし，こうした制御構造を PROLOGで書いたとすると，基本的には図３－２のような形の制御構造は必要でない。その意味で，PROLOGの方がコード的にも簡略化され，制御構造を意識しな




```
<PS-TOP
(LAMBDA NIL
     (*PS-TOP CONTROLS SYSTEM FLOW OF PURE PRODUCTION SYSTEM)
     (PROG (RULES MODEL RULES* DB RULE RULE-NAME)
      LOOP1
           (START)
      LOOP2
           (SETQ RULES* RULES)
      LOOP3
           <COND ((NULL RULES*)
                  (COND ((END) (GO LOOP1)) (T (RETURN NIL>
           (SETQ RULE (CAR RULES*))
           (SETQ RULE-NAME (CAR RULE))
           (*IF PATTERN-MATCH-DB RETURNS TRUE, ACTIONS OF THE MATCHED
             RULE IS ACTIVATED )
           (COND ((PATTERN-MATCH-DB (CADR RULE))
                  (ACTIVATE-ACTIONS (CAR (CDDDR RULE)))
                  (PRINT-RESULT)
                  (GO LOOP2)))
           (SETQ RULES* (CDR RULES*))
           (GO LOOP3))>
```

図３－２　単純な制御構造のプロダクション・システムのフローチャートとLISPコード

いでよい。それでも，以前に作成されたコードを，プログラム・パッケージとして効果的に使うために，種々の情報が必要である。つまり，プログラムのドキュメンテーションで，このことは図３－３のような位置づけとなろう。



（*メールというのは，メッセージ交換システムのことで通常はＴＳＳの環境に付加されている。このメールによって，プログラム変更を，ライブラリー・ユーザに伝える。）

図３－３　知識ベース・システムの作成プロセス

こうした枠組で，図３－２のモジュール構造を多様化することができる。例えば適用するルールが複数個あるときに，最良のルールを選択するルール競合の解消システムが実際には必要である。この部分については，たとえPROLOGを用いたとしても，何らかのモジュールを基本のプロダクション・システムのモジュールに付加していって，実現していかなければならない。このルール競合の解消の制御構造を示したのが，図３－４である。

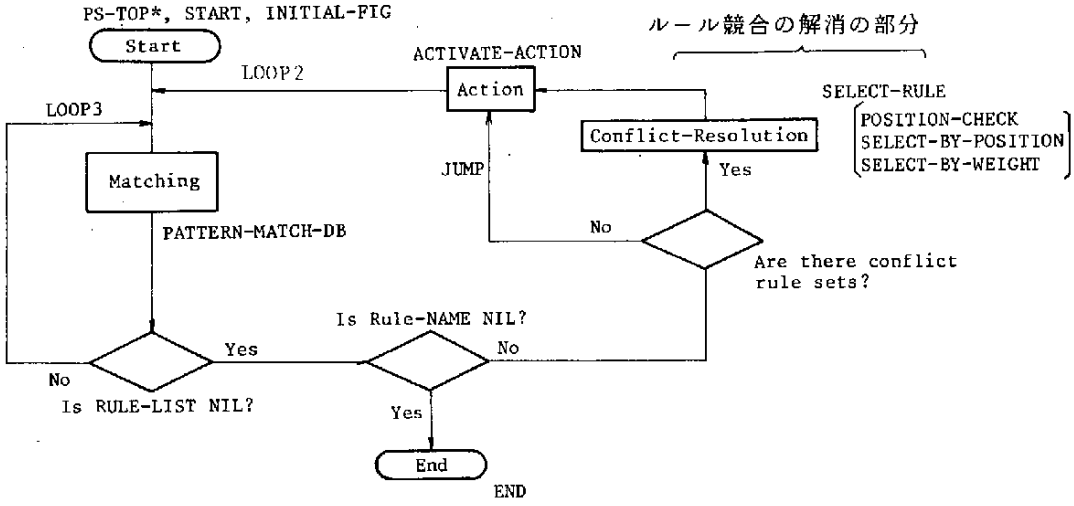


図３－４　ルール競合の解消の制御構造

このようにして，知識ベース・システムの道具としてプロダクション・システムを開発することができる。このプロセスでも明らかなように，基本モジュールがパッケージとして，どれぐらい蓄積されているかが，新しいプロダクション・システムを開発する上では重要である。おそらく，プログラムの検証は，こうしたモジュール・パッケージに対してなされ同時に，多様な文書形式が必要になってくる。

知識ベース・システムの道具だてを作成する段階としてみれば，このモジュールの蓄積と経験が必要である。

したがって，知識ベース・システムの開発経過としては，以下のアプローチをとる必要がある。

㋑ 知識ベース・システム作成のための道具を（LISP＋PROLOG）により種々のプログラム・モジュール群より組み立てていけること。

㋺ ㋑のための，プログラム・モジュール群を準備しておくこと。

㋩ ㋑のための，プログラム・モジュール群のドキュメンテーションが完備していること，そして，必要に応じて，モジュールの呼び出しが自由に行なえること。

（こうした，プログラム活動は，知的支援システムとなり，第5世代コンピュータの課題でもある。）

以上の道具だてを整備し終えた段階が，それを使っての，知識ベース・システムの作成である。これには，専門家との共同作業で，専門的知識を，道具であるプロダクション・システムに入れていくことである。

この共同作業の方式としては，単一のトップレベルの専門家でいくか，または，多数の平均的専門家の合意形式でいくかによって，知識の収集の方法が異なっている。

3.4 知識ベース・システムの現状と評価

今まで述べたような形で，多くの知識ベース・システムは開発されてきたと推察できる。ただし，知識ベース・システムの多くは，米国で開発されているので，我々が知り得るのは，完成したシステムの，表面の部分である。

知識ベース・システムが完成するまでの，生々しい部分は，外部には伝わってこないのが普通である。

ただし，3.2で述べたような道具だての作成過程は，結果的に整理できることであって，それほど一貫した形で，知識ベース・システムが構築されたとは言えないであろう。

逆に，パッケージ環境を用いた知識ベース・システムの開発が，新しい展開方法と言える。というのは，ゼロックスPARCの個人の情報環境支援システムの考え方は，明らかに，コーディング環境を整備するものであるし，また，システム設計の選択の幅を，いかにして自由度をもたせていくという方向である。

その意味では，米国の大学および研究所では，いわゆる第1世代の人工知能システムは完成し

たとみているようである。そして，第1世代のシステムから得たものは何かと言うと，知識ベース・システムを作成するためのプログラム環境はどのようなエディター，モジュール群から構成されているかということに集約されていると思われる。

どのような領域を，知識ベース・システムの対象にしていくかは，専門家の存在にもよるが，表3－1に整理した知識ベース・システムの開発現状も参考になると思われる。

表3－1は，特に，各種の診断システムが示されている。そして，基本的にはルール・ベースの知識ベースを載げている。評価というのは，実際に使われているか（実際のケースを入れた性能テストを終えていること），あるいは，仮定データによるテスト段階かどうかをみたものである。

これらの評価をみると，実用段階のシステムとなっているのは，それほど多くない。つまり，約10年間の診断システム開発のうち，スタンフォード大学のPUFFを除いては，いずれもが，現実的には，実際に適用されていないことに注目する必要がある。

むしろ，作成した知識ベース・システムを医学教育用のCAIとしていく形で，研究が進んでいる。例えば，MYCINからはGUIDONという感染症のCAIシステムが作成されている。こうした方向での，知識ベース・システムの応用が進むのが新しい方向と言えよう。

つまり，コンセプト的には知識ベース・システムによる知的アシスタント・システムであるとして，医療診断システムは，むしろ，エキスパートのシミュレータとして位置づけられると言えよう。

表３－１　診断用の知識ベース・システム

名　称	開発機関	手　法	機　能	使用言語およびコンピュータ	マンパワー	評価および特徴	文　献
PIP	MIT＋Tufts	フレーム 意味ネットワーク	腎機能 （診断）	CONNIVER MAC-LISP PDP-１０	1974 年以来開発 MITの臨床決定グループ＋Tufts医学部	人工知能を医療診断に適用するときの基本的考え方を述べる。システムはテスト段階	〔Szolovits 78〕
Digital Therapy Advisor	MIT	フレーム	心不全 （診断）	MAC-LISP PDP-10	1975年以来開発MITとTufts 医学部との共同研究	時系列データの処理と記号処理のミックス・エキスパートなみの性能を持つ。	〔Pauker 76〕
CASNET	Ratgers	意味ネットワーク	緑内障 （診断）	FORTRAN-V PDP-10	1972 年にスタート Mt. Sinai 大学 Ratgers 大学	1973 年の眼科学学会で性能評価を行なう。	〔Weiss 78〕
MYCIN	Stanford	プロダクション システム	感染症 （診断）	INTER-LISP PDP-10	1978 年スタート， 1976 年完成 計算機科学と医学部との共同研究	1979 年専門医とMICIN との性能比較が行なわれ90％診断性能を持つ。	〔Shortliffe 76〕
INTERNIST	Pittsburg	フレーム 仮説生成	内科学 （診断）	INTER-LISP PDP-10	1975年以来Pittburgの医学部との共同	対象疾患数が 500 という最大の診断システム，テスト段階	〔Pople 75〕
PUFF	Stanford	プロダクション システム	肺機能 （診断）	INTER-LISP PDP-10	1978年以来 Pacific Medical Center との共同研究	MYCIN のルール的記述で約250個のルールを持つ。 すでに，100ケース以上の事例を研究している実用システム	〔Kunz 78〕
SACON	Stanford	プロダクション システム	構造物 （診断）	INTER-LISP PDP-10	EMYCIN を使っての応用である	ルールベースのアプローチを医学以外の分野に適用したもの。テスト段階	〔Bennett 79〕
PROSPECTOR	SRI	プロダクション システム	鉱物資源 （診断）	INTER-LISP PDP-10	後もどり型のルールベースを使ったシステムである。SRI の研究で実用をめざす	鉱物資源のためのエキスパートシステムで，実用をめざしたもの。テスト段階	〔Gaschnig 80〕

3.5 おわりに

診断用の知識ベース・システムとしては，今後は医療のみならず，機械システム，構造物，化学プラント，原子炉などの工学システムに適用されていく方向が期待される。また，その意味では，第5世代コンピュータで作成される知識ベース・システムの可能性は，きわめて大きいと言えよう。

こうした可能性は，新しいプログラム言語であるPROLOGの利用と共に広がっていくと考えられる。現段階でも，〝Why PROLOG ？〟（なぜPROLOG を選んだのか？）という声が聞かれる。逆に，〝Why not PROLOG ？〟（PROLOG 以外に，新しい言語はあるのか？）と問うてみるのも重要であろう。そして，従来までの蓄積を考えつつ，まだ，未経験の領域および言語の可能性を考えていくことが，知識ベース・システムの役割と考えられる。

参 考 文 献

[Bennett 79] Bennett, J S and R S Engelmore. "SACON: A Knowledge-Based Consultant for Structural Analysis." Proceedings of the Sixth International Joint Conference on Artificial Intelligence, August 20-23, 1979, pages 47 through 49.

[Gaschnig 80] Gaschnig, J G. "Development of Uranium Exploration Models for the Prospector Consultant System." Final Report, SRI Project 7856, Artificial Intelligence Center, SRI International, Menlo Park CA, March 1980.

[Kunz 78] Kunz, J.C., Fallat, R.J., McClung, D.H., Osborn, J.J., Votteri, B.A., Nii, H.P., Aikins, J.S., Fagan, L.M., and Feigenbaum, E.A.: "A physiological rule-based system for interpreting pulmonary function test rules," Stanford Heuristic Programming Project Memo / HPP-78-19, 1978.

[Pauker 76] Pauker, S G, G A Gorry, J P Kassirer, and W B Schwartz. "Towards the Simulation of Clinical Cognition." American Journal of Medicine, volume 60, June 1976, pages 981 through 996.

[Pople 75] Pople, H E, Jr. et al. "DIALOG: A Model of Diagnostic Logic for Internal Medicine." Proceedings of the Fourth International Joint Conference on Artificial Intelligence, September 1975, pages 848 through 855.

[Shortliffe 76] Shortliffe, E H, Computer Based Medical Consultations: MYCIN. New York: Elsevier, 1976.

[Szolovits 78] Szolovits, P., & Pauker, S.G.: "Categorical and Probabilistic Reasoning in Medical Diagnosis," Artificial Intelligence 11, 1978, 115-144.

[Weiss 78] Weiss, S M, C A Kulikowski, and A Safir.: "Glaucoma Consultation by Computer." Computers in Biology and Medicine, volume 8. 1978, pages 25 through 40.

（東京理科大学　溝口文雄）

昭和 57 年 3 月発行

発行所　財団法人　日本情報処理開発協会
東京都港区芝公園３－５－８
機 械 振 興 会 館 内
ＴＥＬ　（434）8211（大代表）

印刷所　株式会社　正　　文　　社
東京都文京区本郷３－38－14
ＴＥＬ　（815）7271