述語論理型プログラムの分散処理型アーキテクチュアによる実現

田村浩一郎、田島裕昭、岡田義邦、梅山伸二
（電子技術総合研究所）

1. はじめに

述語論理型プログラムは、意味論が明確であり、また表現形式が簡明であるという点において、いわゆる知識の表現形式としてすぐれている。一方その意味から見て本質的に分散（並列）処理アーキテクチュアによる解釈に向いている。本稿では述語論理型プログラムの一形式およびその解釈を行なう分散処理アーキテクチュアを提案する。このシステムをDIALOGと呼ぶことにする。（名のいわれは後述する）。DIALOGの目的は知識ベース型パーソナルコンピュータである。

2. 述語論理型プログラム

述語論理型プログラムとは、知識表現形式として述語論理形式を用い、その表現形式が計算機によって直接解釈し実行しうるものをいう。これは主として、Kowalskiの提唱[1]による。この提唱を具現化したものがPROLOG[2]である。PROLOGの良さは、あちこちの文献にくりかえし書かれているのでそれをくりかえさないが、PROLOGに対する反発に対してコメントしておきたい。それらは、（1）プログラムが書きにくい、（2）遅い、（3）小さなプログラムしかできないのではないか、といったものである。これは、筆者らの1人が約10年程前LISPのプログラムを作りはじめた時のLISPに対する世間の反感に似ている。現在ではこれらはLISPに対してはいわれのない偏見であることがわかる。（もっともLISP自体にも様々な工夫が盛り込まれ改良されてきたが）。PROLOG（というよりもPROLOG風）言語について、（1）は慣れの問題がほとんどである。（2）はインタプリータは当然遅いが、コンパイラは場合によってはLISPより早い。（3）はたとえば、Bundy[3]のプログラムは決して「おもちゃ」ではない。しかしPROLOG自体に対する的を射た批判もある。それについては後述する。

述語論理型プログラム言語の位置付けについて述べる。できるだけ楽にプログラムをしたいという欲求はプログラム言語に対する変わらぬ欲求である。一方できるだけ効率の良いプログラムを書きたいという、多くの場合相反する欲求もある。プログラムというのは、計算機に対する意思伝達のための表現形式であるから、問題の解法（あるいはシミュレーション）の定式化とプログラムの表現形式との間の距離をできるだけせばめることがプログラムを楽にする道である。この観点から既存の言語をレベル分けすれば、（1）アセンブラ、（2）古典的高級言語――たとえば、COBOL, ALGOL, FORTRAN, あるいはLISP, APL――それに最近のオブジェクト中心的な言語[4]、（3）述語論理型言語、となる。

（2）と（3）のちがいは、前者は後者に対してより手続き的であり、後者は

前者に対してより宣言的であることによる。もう少し明確に言うならば、表現形式に内在する解釈の方向性（いわゆるコントロール）の与え方の大きいものがより手続き的であり、小さいものがより宣言的である。（３），（２），（１）の順で、ユーザーの与えられる情報が多いから、この意味で、この順に計算機を効率良く動かすプログラムの作成が可能である。しかし、当然人間の思考時間が長くなるから、広い視野で効率を問題にすると、どちらが良いかは一概には言えない。

（１）、（２）、（３）の順に、計算機側の解釈の自由度が増える。どの言語も明確な意味規則が与えられなければならないが、この規則さえ守れば、あとは計算機がその枠内で実行を効率良くするように、言語プロセッサやアーキテクチュアを考えて行けば良い。なお、（３）レベルは述語論理型に限る訳ではなく、O'Donnell［５］のように、関数の等式を用いるものもある。

（３）レベルにおいては、方向性の欠除から、分散（並列）処理アーキテクチュアによる解釈に自由度が得られ、直感的にこのアーキテクチュアが向いていると思われる。

一方、分散処理向けの言語も数多く提案されている。しかし、ほとんどはプロセスとプロセス間通信を規定するものであり、解釈の方向性およびデータの流れをユーザが強く規定する必要がある。この意味で、これらも（２）レベルに該当する。これらは効率の良さを発揮できるが、効率を追ってプログラムが複雑化した時、見通しが悪くなるという欠点を本質的に持っている。

述語論理型プログラムをプロセスおよびプロセス間通信の概念で操作的に解釈することができる。（この方法で中島［６］は並列ＰＲＯＬＯＧを提案している。）この時、述語論理型プログラムは分散処理の動作の仕様を与えていると考えることができる。実際にどう解釈し実行するかは計算機のハードとソフトにまかせることになり、この分だけユーザは楽をすることができる。また、述語論理型プログラム言語は参照透明性が（基本的には）守られているから、この意味でも分散処理アーキテクチュアのプログラム言語に適している。その一形式としてＤＩＡＬＯＧ－Ｌを提案する。その前に現在のＰＲＯＬＯＧの問題点に触れる。

３．ＰＲＯＬＯＧについて

ＰＲＯＬＯＧは逐次的処理を前提にして作られている。そこで、解釈の方向性がテキストに対して縦（句間）では上下、横（句内）では左右に固定されている。この方向性をユーザはプログラムの効率化に利用する。このため、ＰＲＯＬＯＧプログラムをいきなり非決定的に並列動作で解釈すると、予期した結果が得られない。また、基本的にＨｏｒｎ集合に限定するため，正リテラルを２個以上句内に書けず，これは、サブプログラムの母体で否定を使えないことに相当し，かなり不便である。特に並列動作をさせようと思うと，これは、場合によっては致命的でもある。わかりやすい階乗計算（ＦＡＣＴ）を例にとって説明する。

ＰＲＯＬＯＧでは、

```
+FACT(0,1).
+FACT(*N,*X)-SUBT(*N,1,*N1)
             -FACT(*N1,*Y)
```

−MULT(↑N, ↑Y, ↑X).

として階乗計算 X=n! ができる。これは、縦方向では、↑Nが0の時、+FACT(0, 1)で"引っかかって"停止するが、もし並列に解釈し実行するならば、2番目の句がどこまでもくりかえされる。これを避けるためには、

−INF(0, ↑N)　{コメント: 0<N}

を第2の句に入れておく必要がある。この例ではこの程度の修正で済んだが、多くの場合 *if* P *then* ～ *else* ～ のような仕組みを表現したい時には、*else* の部分に対して～Pの表現が欲しくなる。直観主義的に～Pに相当する命題を直接構成すれば済むことであるが、一般にこれは繁雑になる。PROLOGでこれを許さないのは、証明の完全性にこだわるからであるが、実はPROLOG自体はかなり実用的に汚染されており、インタプリータ(ないしコンパイラ)の機構が、もともとの意味論的限定を離れてひとり歩きをしている。そのため一般のPROLOGプログラムはもはや証明の完全性を保たない。良し悪しは別にして、これが現実である。この辺の事情は、現実のLISPは純粋LISPから様々な点で離れていることに似ている。

4. DIALOG-L

DIALOGシステムの中心はプログラム言語DIALOG-Lである。この言語に合わせてハードウェアのアーキテクチュアおよびその管理プログラムが設計される。ここではDIALOG-Lの基本的性格について述べる。

4.1 表現として *skolem* 標準形を基本とする。ただしPROLOGと同じくひとつの句内に正リテラルはひとつ、しかも1番左に位置させる。

例　+A−B−C

これは、もちろん

A←B∧C

と等価である。

4.2 句内の正リテラルを見出し(*handle*)と呼び、残りの負リテラルの集合を母体(*body*)と呼ぶ。母体が空のものもある。

4.3 選言に方向性をもたせない。したがって、母体内のリテラルの解釈に優先順位はない。

4.4 連言において方向性を持たない。したがって句内における解釈の優先順位はない。

4.5 負リテラルがそれに対応する(同一の述語を持つ)正リテラルを見出しとする句を「呼び出す」という方向性は持つ。引数の結合は *unification* による。見本的に *input resolution* である。

4.6 変数に入力，出力の方向性の意思表示を許す。↑Xと書けば、変数X

は、その語の計算から値が定まることを仮定する意思表示とみなし、(つまり出力)、↓Xと書けば、その時点ですでに値が定まっていると仮定する意思表示とみなす。＊Xと書けば、そのいずれかで、固定しないものとみなす。これに近い概念にHoare〔7〕の入出力記号(?と!)およびエジンバラ大のPROLOGコンパイラ〔8〕でのモード+,-があるが、それらと同一ではない。

4.7 母体内において否定表現を許す。これを、--Pのようにあらわす。これは命題論理的には当然+Pと同じであるが、ここでの意味は、アトムPがDIALOG-Lの操作的解釈内で偽であることが証明された時に限り、-Pが真となることを示す。したがって必らずしも証明の完全性は保証されない。(しかし便利である)

4.8 正リテラル+P(＊X,＊Y)について、Witginstein流〔9〕に＊Xの値と＊Yの値は等しくないものとみなす。等しい場合は、+P(＊X、＊X)とした時のみ成立する。

5. DIALOG-Lの例

5.1 並列階乗計算

```
+FACT(0,1).                                          ①
+FACT(↓N,↑X) -INF(0,↓N)
              -FACTD(1,↓N,↑X).                       ②
+FACTD(↓X,↓X,↑X).                                    ③
+FACTD(↓X,↓Y,↑Z)
              -MIDDLE(↓X,↑M,↑M1,↓Y)
              -FACTD(↓X,↓M,↑L)
              -FACTD(↓M1,↓Y,↑H)
              -MULT(↓L,↓H,↑Z).                       ④
+MIDDLE(↓X,↑M,↑M1,↓Y)
              -ADD(↓X,↓Y,↑Z)
              -DIV(↓Z,2,↑M)
              -ADD(↓M,1,↑M1).                        ⑤
```

これは、分割統治によって並列処理のうまみを活かしたプログラム例である。完全に並列演算が可能ならばlognのオーダで!が計算できる。この実現のためにデータフロー計算機を想定する向きもあろうが、それに限らないことに注意されたい。一般にDIALOGはデータフロー計算機より柔軟である。

5.2 いとこ関係

```
+いとこ(＊たろう,＊はなこ)-おやこ(＊たろうのおや,＊たろう)
                          -おやこ(＊はなこのおや,＊はなこ)
                          -きょうだい(＊たろうのおや,＊はなこのおや).
```

この“プログラム”に対して

ーいとこ(↓だれか, ↑だれかのいとこ) ①

または

ーいとこ(↑だれかのいとこ, ↓だれか) ②

または

ーいとこ(↓だれか, ↓だれかべつのひと) ③

または

ーいとこ(↑だれか, ↑だれかべつのひと) ④

または

ーいとこ(あきお, ↑あきおのいとこ) ⑤

などが適用できる。どれを選ぶかはユーザの意図による。たとえば③は「だれか」と「だれか」のいとこ関係のチェック，④はいとこ同志にあるものを知りたい，といったような意味あいである。方向づけをするとその分処理の効率向上が期待できるが融通性に乏しくなる。このトレードオフの処理はユーザにまかせる。ユーザこそがどちらを選択すべきかを一番良く知っている筈だからである。

なおプログラムにおいて，“＊たろう”，“＊はなこ”などと書いたのは，“＊”を“たとえば”と読ませるようにするとわかりやすくなることを示したかったからである。これはIBMのQBE(*Query by example*)に似ている。

6. DIALOG-M ——分散モニタ

DIALOG-Lの解釈・実行のために，多数のメモリ付きプロセッサの高密度結合システムを用いる。この構成については後述する。ここではDIALOG-Lの解釈としてプロセスおよびプロセス間通信を対応づける仕組みについて述べる。この仕組みはモニタ機構がほとんどなので，DIALOG-Mと名付ける。

DIALOG-Lで書かれたプログラムは，プロセスおよびプロセス間通信の仕様を与えているとみなすことができる。したがってその仕様を満足させるという条件のもとで，できるだけ効率の良い実行に結びつけるのがDIALOG-Mの役割である。この機構をすっきりさせるためにプロセスおよびプロセス間通信の概念を利用する。

6.1 句とプロセス

句に対して1個以上のプロセスが対応する。句はそれらのプロセスの動きを規定する記述である。同じ句に対して複数個のプロセスができるのは，5.1の例で，これを最大限並列に稼動させた場合のFACTDに対応するプロセス群を考えれば分る。

プロセスの定義は最近MilneとMilner[10]によって与えられたが，われわれの場合はそれにこだわらず，単に入出力ポートを持つ処理過程といった程度の意味であるとする。

あるプロセスにおいて，その母体内にある負リテラル(負負リテラルも含む)に対応する正リテラルを見出しとする句を記述に持つプロセスを発生させる。

こうして発生したプロセスを子プロセス、発生させたプロセスを親プロセスと呼ぶ。

6.2 プロセス間通信

親子のプロセスの間でデータのやりとりがある。これがプロセス間通信である。このデータの単位を符(token)と呼ぶ。まず親プロセス母体内の負リテラルから、子プロセスの見出しの正リテラルへ符が送られる。この符が見出しで統合できれば、それにもとづいて子プロセスが走るが、その計算過程で、親プロセスに次々と符を返す。親から子への符を指示、子から親への符を提案と呼ぶ。親プロセスは子プロセスからの提案符を総合して、自分の親に提案を返せるならば返し、あるいは他の子プロセスに新しい指示を出すべきならば出す。子プロセスからの提案が尽きたと判定できた時、親プロセスは終了する。母体のない(句を記述とする)プロセスは、親からの指示符と見出しとの付け合わせだけで終了する。

6.3 プロセス内モニタ

プロセス1個に必らずプロセス内モニタがある。この役割は

(i) 子プロセスの管理として、子プロセスの発生、起動、強制終了など。

(ii) 親からの指示符の処理、子プロセスへの指示符の作成、子プロセスからの提案符の総合による親プロセスへの提案符の作成など、符の管理

(iii) 自分自身待つべきか、あるいは終了すべきかの決定がある。

5.1の例で説明する。いま

−FACT(3, ＊X)

が与えられたとする(ユーザが端末から書き込むか、あるいは、あるプロセス内の母体中に出現したかそのいずれか)。すると+FACTを見出し(の一部)に持つ句①, ②に対応してプロセスが発生する。引数のマッチング(統合)が行なわれ、①の(句に対応する)プロセスは失敗の情報を含む符を返して終了。②のプロセスはマッチング成功により、+INF, +FACTDの見出しを持つプロセスを発生し、それぞれ指示符を送る。③のプロセスはすぐに失敗の符を提案して終了。④のプロセスは負リテラルに対応して子プロセスを発生するが、この子プロセスのうち2個のFACTDでは、MおよびM1の値がまだ与えられていないため、符を送れない。そこでこれらのプロセスは指示符待ちとなる。しかし"兄弟"MIDDLEから提案符が親プロセスに返されるとそれにより変数Mおよび M1の値が与えられ、そこで親プロセスから、先きの2個の子プロセスに指示符が送られる。④の最後の行MULTについてのプロセスも同様である。一方、FACTのプロセスで、もしINFが偽になったとすると、それを知った時点(INFからの提案符すべてをもらった時点)では"兄弟プロセス"FACTDが先きに行っているのでこれを強制停止し、親プロセスに偽の符を返す。

このようにDIALOG−Mのプロセス内モニタによって子プロセスの走行に順序付けがなされると同時に、入出力の矢印によって、盲目的にプロセスが展開されることを制御している。もし矢印がなければ、FACTDはどんどん展開され、そのあとを追って親から子に指示符が送り込まれることになる。つ

まり、無駄な先行投資が多いことになり、システムはプロセスで混雑する。

6.4 プロセッサ内モニタ

プロセスは論理的概念であるのに対して、プロセッサは物理的ないわゆるプロセッサを指す。プロセッサは有限であるから、プロセスの数がプロセッサの数を上回ることは大いにあり得る。これを解決するのがプロセッサ内モニタの役目のひとつで、プロセッサに1個ずつ張りついている。

プロセッサ内には複数個のプロセスのイメージ(プロセス内モニタ、プロセスの状態および記述)を持つ、プロセッサ内モニタはプロセス内モニタと連絡し合うが、一方他のプロセッサ内モニタとの通信も行なう。この通信は物理的なものである。いいかえれば、プロセス同志の通信は論理的なものであるのに対して、プロセッサ内モニタはこの論理的な通信を物理的に実現する仕組みを持つ。また、プロセッサ内のプロセス(複数)を管理する役割を持つ。これはふつうの多重プログラミングのモニタと同様である。

このようにプロセッサ内モニタの役割は、DIALOGの論理的アーキテクチュアとハードウェアアーキテクチュアとのインタフェースであり、その詳細はハードウェア・アーキテクチュアに依存する。しかし、おおまかに言えば以下のようになる。

(i) プロセッサ内プロセスの管理
(ii) プロセッサ間通信の管理
(iii) プロセッサ内プロセス間の通信の管理
(iv) 記述の管理(記述のコピー発生)

5.1の例で、FACTDのプロセスがプロセッサ数を上回った時、そこで計算が停止しては面白くない。しかしプロセッサ内モニタの働きによって、この場合は同一プロセッサ内にFACTDのプロセスを詰め込み、ふつうの逐次処理計算機での再帰的プログラムと同じあつかいにすることができる。これはプロセス数が増大した時、ただ処理時間が余分になるだけであり、空間のせまさのしわよせを時間にもち込むという自然な処理ができる。もちろんメモリが足りなくなればダメである。

7. DIALOG-H——ハードウェアアーキテクチュア

以上の論理的アーキテクチュアを具現化する物理的アーキテクチュアについて述べる。これをDIALOG-Hと名付ける。

単一プロセッサでも良いが、それでは効率の良さが発揮できない。これからのハードの価格低下を考慮すると、極めて多数のプロセッサ(たとえば1000個)を用いた分散処理システムがコストパフォーマンス的に見て有利であろう。このようなシステムの場合、プロセッサ間通信をどのようにするかが最大の課題である。

ハードウェア・アーキテクチュアは汎用性をねらうならば概念的に見て出来るだけ単純なものが良い。そのわく組みの中で個々の部分に工夫を凝らして行くべきである。それが現在のノイマン型コンピュータの大成功の秘訣である。

7.1 基本構成

DIALOG-Hの模式は単純である。共通バスおよび複数個のメモリ付きプロセッサだけである。プロセッサと共通バスとのインタフェース部には、通常のバス管理に加えて、連想メモリを置く。バス管理は、バスへの入出力、バッファおよびエーテル・ネット流の調停(arbitration)機構を持つ。連想メモリは、バス上のデータが、自分のプロセッサが取り込むべきものかどうかを判定するのに用いる。バス方式であるから、1対多の通信の時、最も威力を発揮する。

問題はバスによるボトルネックである。再び5.1の例を見る。通信が同時にできるならば確かにn!の計算を$\log n$のオーダでできるが同時にnのオーダの通信も最下段で行なうため、バス方式を採る限りは、実際はnのオーダになる。問題は両者の係数の比である。これが極めて小さければ、つまりバスが高速ならば、実用的には十分バランスがとれる。

通信を効率良くする理想的な機構は交換式である。しかし、プロセッサ数が大きいと、ハードも(ソフトも)複雑化し、効果的でない。またネットワークを構成する方式は線の数が増大し、そこからプロセッサ数の限界が与えられてしまう。

いずれの方式も一長一短があるが、ここでは高速バスを追求することによってバスの弱点を補いつつ、その利点を積極的に生かすことを考える。DIALOG-Hでは光バスを用いる。

7.2 光バス

光による情報伝送の最近の進歩は、ファイバーを中心にして目覚しいものがあり、現在数ギガb/s程度まで可能である。ファイバは地理的分散を埋めるためのものであり、DIALOG-Hは密結合をめざすから、ファイバーは用いない。そのかわり図1のように反射鏡を用いる。

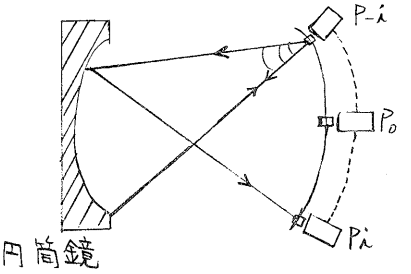


図1

P-i, ……, P0, ……, Piの各プロセッサ毎に発光・受光素子がある。あるプロセッサPjから発光すると円筒形の反射鏡によりその光は広がって、P-i, ……, Piの発光素子に向かう。このようにして、可動部分の全くない高速光バスが実現できる。

この断面を何階建てかにすることによって並列転送が可能になる。そのうちのひとつで特殊なものとしてシステム全体のクロックを作ることができる。これは、クロックを発生する発光素子1個と、それを各プロセッサで受信するためのプロセッサ毎の受光素子とで構成される。

このバス機構を良くするためには、ファンアウトを大きくするために発光素

子の効率化と受光素子の感度向上、伝送速度を向上するために発受光素子の速度の向上とが望まれる。またシステム全体を小型化するための各素子の小型化も重要である。

8. システムDIALOG

DIALOGとは、Distributed Inference Architecture on Logicの略称であり、または文字通り「対話」でもある。上に見たように、システムDIALOGの基本は、与えられた句の分解原理に基ずく分散証明にある。一方われわれの最終目標は高度に知的な「対話型」パーソナルコンピュータを実現することである。そのためには、

(i) 知識ベースの作りやすさ
(ii) コストパフォーマンス
(iii) 軽量小型

が不可欠の条件である。(i)(ii)(iii)すべてを勘案してシステムDIALOGの構成を考え、ほぼ以上に述べたようなものになった。(i)はまだしも、(ii)(iii)は現時点の技術レベルでDIALOGの有利さを決めた訳ではなく今後のマイクロエレクトロニクスおよびオプトエレクトロニクスの成長を考慮してのものである。ハードウェア技術の進歩は激しく、一方システム開発は時間がかかる。要素技術の進歩を先取りし、あるいは逆にそれらのニーズを作り出すようなシステムを設計するのも、ひとつのやり方であろう。

(i)に関して、もひとつ重要な側面、人間とのインタフェースがある。それについて次に述べる。

9. DIALOG-I——人間とのインタフェース

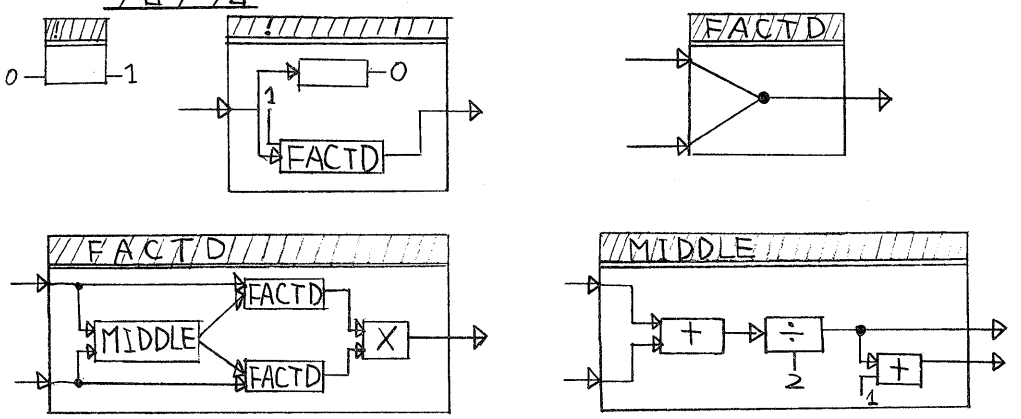
DIALOGのフロントエンド、つまり人間とのインタフェースには、ハードウェアとしては、透明タブレット密着式の平面カラーディスプレイを考える。これは図1の構成の上にそのままのせることができる。

むしろ重要なのは、いわゆる"プログラミングシステム"である。これにはPYGMALION[11]で提唱されたような図式(iconic)プログラミングを基本とする。

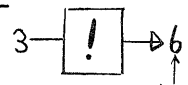
まず、DIALOG-Lの図式表現を考える。DIALOG-L自体、平明でわかりやすいが、これを2次元表記する方がよりわかりやすくなる可能性がある。わかりやすい表現記といえば、かなり短絡的に自然言語表現を持ち込む人も多いが、その方式は疑問である。自然言語は多義性に富み、論理性に欠け、しかも冗長である。この欠点を対話的に解決しようとしても、その場合は如何にもまわりくどい(プログラマことばでいえば"かったるい")。それはともかくとして話をもとにもどす。

DIALOG-Lの図式表現の例として、5-1の例および5-2の例をとりあげる。それは図2-1および図2-2のようになる。

並列階乗計算
プログラム



実行



計算機の出力

図2－1　5－1の例の図式表現

血縁関係
プログラム

実行例

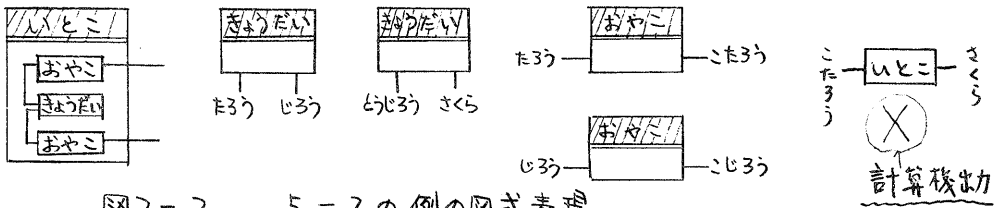


図2－2　5－2の例の図式表現

対応規則は、この例からわかることは除外すると以下のようになる。
（i）負負リテラルは、図と地を反転させたものとする。
たとえば、－－Pは、 
（ii）スコーレム関数 Cons は次のように図示する。
（Z）(X) (Y)　　ここで、Z＝Cons（X，Y）
次の例〔12〕を図示すると、図3のようになる
＋ELEM（＊X，＊X．＊Y）－P（＊X）．
＋ELEM（＊Z，＊X．＊Y）－ELEM（＊Z，＊Y）．
これは、Zが与えられたリストやにありかつ述語Pを満たすことを示す。

図3　ELEMの図式表現

このように、この図式表現は、単なるドキュメンテーションのためではなく、

ディスプレイを通して計算機との直接の情報交換に用いるものである。ディスプレイの使用方法はDynabook方式[13]をとり入れることによって、1個のディスプレイに見かけ上多くの図を詰め込むことが可能である。

以上の機能をDIALOG-Iと名付ける。Iは、Interfaceでもあり、Iconでもある。

10. おわりに

システムのDIALOGの概要について述べた。現在さらに仕様を詰めている段階である。各要素の詳細は機会をあらためて報告したい。

最後に、このプロジェクトを始める機会を与えられ、日頃適切な助言をいただいている佐藤芳平制御部長、および、様々な検討をいただいた論理システム研究室のみなさん、さらに、貴重な教示をいただいた電総研の渕、田中、横井、古川の各氏をはじめ多くの方々に謝意を表わしたい。

文献

[1] Kowalski R A: Predicate Logic as Programming Language; Proc. IFIPC, 1974

[2] van Emden M H: Programming with Resolution Logic; Machine Intelligence 8, Ellis Howard Ltd., 1977

[3] Bundy A, Byrd L, Luger G, Mellish C, Palmer M: Solving Mechanics Problems Using Meta-Level Inference; Proc. IJCAI, 1979.

[4] 島居宏次, 二木厚吉, 真野芳久: プログラミング方法論の展望; 情報処理 Vol 20, No1, 1979.

[5] O'Donnell M J: Computing in Systems Described by Equations; Lec. Notes in Comp. Scie. 58, Springer.

[6] 中島秀之: Parallel Prolog; 第21回プログラミング・シンポジウム報告集.

[7] Hoare C.A.R.: Communicating Sequential Processes; Comm. ACM Vol 21, No 8, August 1978.

[8] Warren D H D: Implementing Prolog - compiling predicate logic problems; Dept. of AI, Edinburgh, 1977.

[9] ウィトゲンシュタイン: 論理哲学論 5.531 世界の名著 70, ラッセル・ウィトゲンシュタイン・ホワイトヘッド,

中央公論社. 1980

[10] Milne G; Milner R: Concurrent Processes and Their Syntax; Journal of the ACM, vol 26, No.2, April, 1979.

[11] Smith D C: Pygmalion. A computer program to model and stimulate creative thought; Interdisciplinary Systems Research 40, Birkhäuser Verlag, 1977.

[12] Gallaire H, Lasserre C: Controlling Knowledge Deduction in a Declarative Approach; Proc. IJCAI, 1979.

[13] Teitelman W: A Display Oriented Programmer's Assistant; Proc. IJCAI, 1977.